

Java ME on Symbian OS



Inside the Smartphone Model

symbian

Roy Ben Hayun

Java ME on Symbian OS

Roy Ben Hayun

With

Sam Mason, Daniel Rocha, and Ivan Litovski, assisted by Sam Cartwright

Reviewed by

**Gavin Arrowsmith, Brendan Donegan, Erik Jacobson,
Martin de Jode, Mark Shackman, Jo Stichbury**

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



WILEY

A John Wiley and Sons, Ltd., Publication

Java ME on Symbian OS

Inside the Smartphone Model

Java ME on Symbian OS

Roy Ben Hayun

With

Sam Mason, Daniel Rocha, and Ivan Litovski, assisted by Sam Cartwright

Reviewed by

**Gavin Arrowsmith, Brendan Donegan, Erik Jacobson,
Martin de Jode, Mark Shackman, Jo Stichbury**

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



WILEY

A John Wiley and Sons, Ltd., Publication

Copyright © 2009

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.com

Visit our Home Page on www.wiley.com

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS, UK, without the permission in writing of the Publisher. Requests to the Publisher should be addressed to the Permissions Department, John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England, or emailed to permreq@wiley.com, or faxed to (+44) 1243 770620.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The Publisher is not associated with any product or vendor mentioned in this book.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 6045 Freemont Blvd, Mississauga, Ontario, L5R 4J3, Canada

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

Hayun, Roy Ben.

Java ME on Symbian OS : inside the smartphone model / Roy Ben Hayun, with Sam Mason ... [et al.].
p. cm.

Includes bibliographical references and index.

ISBN 978-0-470-74318-8 (pbk. : alk. paper) 1. Smartphones—Programming. 2. Java (Computer program language)

3. Symbian OS (Computer file) I. Title.

TK6570.M6H33 2009

005.26'8—dc22

2008053889

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 978-0-47074318-8

Typeset in 10/12 Optima by Laserwords Private Limited, Chennai, India

Printed and bound in Great Britain by Bell & Bain, Glasgow

This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Contents

Foreword	ix
Kicking Butt with Java Technology on Symbian OS	xv
About This Book	xvii
Author Biographies	xix
Author's Acknowledgements	xxi
Symbian Press Acknowledgements	xxiii
Part One: Introduction to Java ME and Programming Fundamentals	1
1 Introduction to Java ME, Symbian OS and Smartphones	3
1.1 2003: Rise of the Mobile	3
1.2 2008: Mobile Generation	6
1.3 Meet the Host – Symbian OS	7
1.4 What Is Java?	9
1.5 Java ME	12
1.6 Why Use Java ME on Symbian OS?	17
1.7 Java's Place in the Sun	21
1.8 Routes to Market	22

1.9	Time for a Facelift	24
1.10	Back to the Future: MSA 2.0 and MIDP 3.0	24
1.11	Summary	26
2	Fundamentals of Java ME MIDP Programming	27
2.1	Introduction to MIDP	27
2.2	Using MIDlets	28
2.3	MIDP Graphical User Interfaces API	34
2.4	Non-GUI APIs in MIDP	56
2.5	MIDP Security Model	59
2.6	Networking and General Connection Framework	69
2.7	Using the Push Registry	78
2.8	MIDP and the JTWI	81
2.9	Mobile Media API	82
2.10	Wireless Messaging API	95
2.11	Symbian OS Java ME Certification	100
2.12	Summary	101
	Part Two: Java ME on Symbian OS	103
3	Enter Java ME on Symbian OS	105
3.1	Running a MIDlet on a Symbian Smartphone	106
3.2	Which APIs Are Supported?	113
3.3	Proprietary JAD Attributes	122
3.4	Computing Capabilities of Java ME on Symbian OS	123
3.5	Java ME: Exposing the Power of Symbian OS	125
3.6	Tools for Java ME on Symbian OS	131
3.7	Java ME Management on Devices	131
3.8	Crossing to the Native Land of Symbian OS	139
3.9	Finding More Information	145
3.10	Summary	146
4	Handling Diversity	149
4.1	General Approaches to Handling Diversity	150
4.2	Detecting Diversity using Properties	151
4.3	Using Adaptive Code and Flexible Design to Handle Diversity	153
4.4	Handling JSR Fragmentation	158
4.5	Handling Transitions Between Foreground and Background	161
4.6	Supporting Diverse Input Mechanisms	162
4.7	Handling Diverse Multimedia Formats and Protocols	166
4.8	Handling Screen and Display Diversity	169

4.9	A Last Resort: NetBeans Preprocessing	173
4.10	Summary	173
5	Java ME SDKs for Symbian OS	175
5.1	Recommended Tooling Approach for Java ME on Symbian OS	176
5.2	Generic SDKs: Java ME SDK 3.0 and WTK 2.5.2	177
5.3	SDKs for the S60 3rd Edition and 5th Edition Platforms	178
5.4	SDKs for the UIQ 3 UI Platform	189
5.5	Summary	205
Part Three: MSA, DoJa and MIDP Game Development		207
6	Designing Advanced Applications with MSA	209
6.1	What Is MSA?	209
6.2	What Can I Do with MSA?	213
6.3	Spicing up Legacy MIDP Applications	222
6.4	Beyond MSA 1.1: MIDP 3.0 and MSA 2.0	227
6.5	MSA and Symbian OS	229
6.6	Summary	230
7	DoJa (Java for FOMA)	231
7.1	In the Beginning...	231
7.2	DoJa – the Basics	234
7.3	I Love JAM	238
7.4	DoJa Basic Operations Manual	240
7.5	Eclipsing DoJa	240
7.6	Dirty Hands	243
7.7	A Safe Port	250
7.8	DoJa 5.1 Profile	254
7.9	Summary	264
8	Writing MIDP Games	265
8.1	What Is a Game?	266
8.2	Building a Simple MIDP Game	273
8.3	MIDP 2.0 Game API Core Concepts	279
8.4	Building an Advanced Java Game on Symbian OS	282
8.5	Summary	303

9	Java ME Best Practices	305
9.1	Investing in the User Experience	305
9.2	Good Programming Practices	308
9.3	Streamlining the Deployment and Execution Lifecycle	322
9.4	General Tips for Symbian OS	325
9.5	Summary	327
 Part Four: Under the Hood of Java ME		 329
10	Java ME Subsystem Architecture	331
10.1	Java Applications and Symbian OS	331
10.2	Application Management Software	336
10.3	MIDP Push	337
10.4	Mean and Lean Virtual Machine	338
10.5	MIDP Implementation Layer	342
10.6	Handling Asynchronous Symbian OS Operations	346
10.7	Java-level Debugging Support	348
10.8	Performance	349
10.9	Security	350
10.10	Summary	351
 11	 Integration of Java ME and Native APIs	 353
11.1	Importance of Integration with Symbian OS Services	354
11.2	Types and Levels of Integration	356
11.3	Integration Challenges, Costs and Considerations	357
11.4	Determining the Right Integration Style	358
11.5	Examples of JSR Integration	359
11.6	Summary	384
 Appendix A: WidSets		 385
 Appendix B: SNAP Mobile		 411
 References		 433
 Index		 435

Foreword

James Coplien

It was great re-discovering through this book just how much of a nerd I am. Roy's descriptions and exercises painted amazing pictures in my head of the internal workings of my own Nokia S60 phone, to the extent that I unearthed new and useful menus previously invisible to me. While I can't promise that every casual reader will find this book suitable as a user's guide, and while I can't promise that every programmer will become an expert user of their own handset by reading the chapters that follow, I can promise the intent programmer a treasure chest of knowledge not only for understanding what goes on inside your phone but also for changing what goes on inside your phone.

Java ME on Symbian OS is a title that sounds like nerd's heaven – and it is (though it's more than that, as I'll come back to below). Though the title bears two trade names, the meat of the book proudly displays its inclusiveness across the market: Sun, Symbian, Nokia, Samsung, LG, Sony-Ericsson, Motorola . . . It's the ultimate mash-up. This is a book less about niche technologies than about a broad phenomenon. It isn't only a technological phenomenon but a sociological one; if you don't view it that way today, I believe that vision is rapidly approaching tomorrow.

To the everyday programmer, I think this book will open a new world of wonder, as history has similarly favored our industry several times in the past. Twenty years ago, someone would have thought you to be drunk if you asked him to take a picture of you with his phone. Today we have lost touch with photographic film and, to a large degree, with those ancillary devices called cameras. We used to have a shelf-full of gadgets for voice dictation on the run, digital photography, calendar management, and, yes, even communication – all of which are

converging on single small, convenient computers that are the brain-grand-grand-grand-children of Alexander Graham Bell. And those boxes need software – software such as you will find in this book, along with guidance for bringing it to life.

It was inevitable that the evolution of consumer programming should turn to the telephone. I remember the visions of unification of computers and telephones in the minds of people such as Jerry Johnson back in Bell Lab in the 1970s. We can now look back at the past 20 years and conclude that computers' primary contribution to society has been not as calculating engines in their own right, but as tiny points on an almost fully-connected network that ties together most computers in the world. The phone network is more than just a metaphor for the worldwide web: it is often its backbone. As the pundits predicted, the intelligence has evolved out of the network into its periphery, out of its bowels and into its eyes, ears, and fingers.

The intelligence hasn't stopped at phones. Next year, Nokia and RWE are expected to market a box that extends the phone network to control heating, window shades, and home security cameras. The possibilities are endless. Programmable phones will replace not only cameras and small dictation machines, but will reach ever further into both our immediate and physically remote realities. They have already made a significant foray into the video game space, taking advantage of advancing display technology, complementing our newfound connections with reality with newfound escapes into fantasy. In the not so distant future, these two worlds may blend and merge. Whereas yesterday's play was a personal Tamagotchi and today's play includes multiple Tamagotchis talking via an infrared Tamacom protocol, tomorrow will see our phones as a Tamagotchi interface whereby we feed or clean up after the flesh-and-blood Fido back home – all with a sequence of a few DTMF tones.

Roy's book takes us on a journey into a land that is more than just nerd heaven. The history of computing has drawn more and more everyday folk into programming. This trend sits squarely in the center of the vision of people such as Alan Kay and Adele Goldberg: that computers should be the sidekicks of human beings from early life, enabling children of all ages to extend their memory and processing power by manufactured, rather than biological, means. This vision of a Dynabook – a truly personal computer, an extension of self – includes information tethered to the outside world as one of its central building blocks. Because my Dynabook is part of me, I get to program it. Today, that programming means setting a date on my calendar or setting an alarm for a meeting. Those humble acts may not entail using Java, but

it's still programming: as Dan Barstow used to say, "No matter how high-level, it's still programming." To me, as a C++ programmer, Java looks almost like a scripting language. And that's not even an insult. Java has perhaps finally realized its vision of ubiquitous expression of general-purpose computation that is portable and intelligible enough that Everyperson can at least tweak the code. I know that such aesthetics are difficult to judge from the seat of a professional programmer, but we can at least say that Java code provides a glimpse of such a world. Perhaps even my neighbor, the real-estate salesman, could read some of the code in this book and understand it well enough to play with it. Or my dentist. Or my banker. None of them are professional programmers. But each has a computer on his or her desk: a machine which 40 years ago would have terrified most secretaries, and which still terrifies some executives today. Machine creatures inhabit more and more of the human ecosystem.

It's interesting to note that such programmability has played out not in desktop phones (yes, they still exist) but in hand-held portable phones. Why? Perhaps it is because only personal cellular phones are close enough to our heart – or some other part of our anatomy – to truly be part of Self. It was a revolution when people who bought the early Ataris and Amigas discovered that they could actually program them themselves. We are perhaps on the threshold of a revolution that unleashes more powerful programming into the hands of what will be over one billion owners of Symbian OS phones in two or three years. Even if great programmers arise from only one in a million of these users of small-handheld computers that take pictures, play games, and even place phone calls, it will be enough to move the world.

This book, as a nerd's text, is nonetheless a bold foray into Dynabook territory. If nothing else, it will make you realize that if you own a regular personal computer and a mobile phone, that it's within your reach – today – to write software that can be an at-hand extension of yourself any time, anywhere. It is perhaps one small step on the road to a future vision of even further integration between man and machine. Such integration must go forward with intense ethical scrutiny and social care; if we choose such a path, machines can make us even more human, humane, and social than we are today.

I leave you with a poem which, when published, was wrongly attributed to me alone. It was written by the attendees at VikingPLoP 2004 as we took the human-computer symbiosis to its limit. Think of *Java ME on Symbian OS* as an important step on this journey. Nerds, children of all ages, and everybody out there: Happy Programming!

Comfortable as blue jeans

She fits.
Part of me, yet not me;
Unplugged: oxygen is her food
And the breeze her power adaptor.

Her identity mine, yet not me:
I put her down, I take her up
like a well-worn wallet plump
with life's means and memories.
A thin mask worn in life's play
And both of us are players.
I call her Self; she does not call me User
A curse unique to programmers and other pushers.
I and my computer are one
But we own each other not.

Envision this: my new machine, a soulmate,
With whom I talk in whispers unencumbered by flesh and bone
Software begotten, not made
Of one being with its person.
She hears my voice, and perhaps my thoughts
rather than the drumming of my fingers
that burdened her forebears:
35 calcium levers linking brain to digits,
slow and tired machines of the information age.
Now, mind to mind,
there for tea and sympathy
all day, and our night
minds together process the day,
sharing dreams – yet
She slumbers not, nor sleeps.

An invisible mask the birthright of all humanity;
Her software penned by
My soul
And like my soul, her hardware
Im-mortal, invisible
Us together wise.
Transparent personæ mine, yet not me,
That with me loves, and lives, and walks life's journey –
A path both real and virtual at once
as only a True and rich life is.

With me she dies
– our immortality inscribed in
the fabric of the network.
No duality of community here
and Internet there –
The Network is the Com-
munity:

One Web of life.

Jim Coplien
Mørdrup, Denmark
2 July 2008
en.wikipedia.org/wiki/James_O._Coplien

Kicking Butt with Java Technology on Symbian OS

Mobile phones have changed the lives of billions of people and it's clear that the revolution is just beginning. The quickly evolving world of mobile technologies is an exciting place for developers to create new applications to connect people and information.

Java technology has enjoyed dizzying success in just under a decade in the mobile phone industry. As I write, more than 2.1 billion Java technology devices are deployed worldwide. At the same time, Symbian OS dominates the fast-growing smartphone market (with approximately 70%). This book is about the confluence of these two technologies and how to create your own powerful mobile applications.

Now is a great time to be a mobile developer. You can help shape the coming landscape of advanced mobile applications by harnessing the power of both Java technology and Symbian OS. Learn the technologies, let your imagination run free, and have fun!

Jonathan Knudsen

jonathanknudsen.com

Author of Kicking Butt with MIDP and MSA

About This Book

In 2001, Symbian published its first book devoted to Java on Symbian OS. Jonathan Allin's *Wireless Java for Symbian Devices* provided an in-depth exposition targeted at programming PersonalJava on Symbian OS. In 2004, Martin de Jode's *Programming Java 2 Micro Edition on Symbian OS* focused on programming MIDP, particularly MIDP 2.0 on Symbian OS. This book is the third in the series and there is no longer a need to explain what Java is. Therefore, the primary goal of this book is not to teach Java programming but to introduce you specifically to Java Platform, Micro Edition (Java ME) on Symbian OS.

This book covers various topics from different perspectives. Our approach has been to start with the basics and prerequisites that are assumed in the rest of the book, then zoom into Java ME on Symbian OS specifically and then out again to discuss a few key areas of Java ME development today. We finish the book by delving deep down to expose what's happening 'under the hood'.

The book logically divides into four main parts:

1. Introduction to Java ME and programming fundamentals

In Chapter 1, we describe the playground of Java ME on Symbian OS. It is recommended reading if you are not familiar with Java ME, Symbian OS or the smartphone industry. Chapter 2 is targeted at developers with no experience in Java ME. It sets the baseline information that is a prerequisite for reading the rest of the book, which assumes a certain level of knowledge of Java programming and, specifically, Java ME. Chapter 2 deals with the basic concepts of a MIDlet, LCDUI, MMAPI, and so on.

2. Java ME on Symbian OS

Part 2 assumes you are familiar with Java ME but are interested in learning about the Java ME platform hosted on Symbian OS. Chapter 3 explains what makes Java ME on Symbian OS different from other Java ME platforms; what makes it more powerful; and what else is unique to Symbian smartphones. Chapter 4 provides guidelines on how to handle differences between different Symbian smartphones. Chapter 5 provides an overview of development SDKs for Java ME on Symbian OS.

3. Drill down into MSA, DoJa and MIDP game development

Part 3 expands the discussion into various key areas of Java ME development. We show how to leverage the power of MSA, give an introduction to how Java evolved very differently in the Japanese market, provide a glimpse of games development and equip you with best practices for your next application.

4. Under the hood of Java ME

In Part 4, we reveal information that has rarely been exposed before. Chapter 10 explains the internal architecture of the Java ME subsystem as a Symbian OS component and Chapter 11 explains how Java ME on Symbian OS is tightly integrated with the native services.

Appendix A introduces WidSets which is a run-time environment that makes use of the Java language and the Java ME platform. Appendix B introduces SNAP Mobile for game developers.

This book contains a number of web addresses. If any of them are broken, please consult the wiki page for this book at ***developer.symbian.com/javameonsymbianos*** to find the updated location.

We welcome beginners and professionals, third-party application developers and OEM developers working inside Symbian OS. Whether you're just starting out in the exciting world of mobile development, want to learn Java ME or just like reading interesting stuff, this book is a mix of different topics that combine in the same way as a dish full of spices.

Author Biographies

Roy Ben Hayun

Roy Ben Hayun has more than 10 years' experience in various Java technologies in various software engineering roles: engineer, consultant, team lead, tech lead, and architect. He started his career in Lotus IBM, working on JDK 1.1.8, and then became a founding member of eMikolo networks, which pioneered P2P in the early days of Project JXTA. He later worked in various roles in other startup companies in Java SE, Java ME and Java EE. From 2002 until the end of 2007, he worked for Symbian, mostly in the Java group working on the design and implementation of JSRs and CLDC-HI VM tools support. In his last year at Symbian, he worked in the team supporting developers, during which time he authored a number of technical articles and white papers about run-time environments.

After JavaOne 2007, Roy joined Sun Microsystems. He is currently working as a system architect in the Engineering Services group, which leads the development, marketing and productizing of Java ME CLDC and CDC on various platforms.

Ivan Litovski

Ivan joined Symbian in 2002, working first in the Java Team and later with the System Libraries, Persistent Data Services and Product Creation Tools teams. Ivan has 11 years of professional experience with Java, embedded software, information security, networking, scalable servers and Internet protocols. He enjoys solving difficult problems and research and development. He has authored several patents and papers in peer-reviewed,

international journals. He holds a BSc in Electronics and Telecommunications from the University of Nis, Serbia and is an Accredited Symbian Developer.

Sam Mason

Sam spent about two years working on a Java-based, multi-lingual, video-on-demand system for SingTel and other Asia-Pacific Economic Cooperation (APEC) telecommunications agencies. He has spent most of the last four years working with and learning about mobile phone technologies, while picking up a couple of Java certificates and completing a Master of Information Technology (Autonomous Systems) at the University of New South Wales along the way. He was a contributing author to *Games on Symbian OS*, writing the chapters on Java ME development and DoJa. He is an Accredited Symbian Developer.

Daniel Rocha

Daniel is a software engineer with 10 years' experience in application development, having worked with web (Perl, ASP, PHP, JavaScript, JSP, Flash), enterprise (Java EE) and mobile software (Symbian C++, Java ME, Flash Lite, Python). He currently works as a Forum Nokia Technology Expert.

Author's Acknowledgements

I feel vividly that this project was an amazing experience, primarily because of the people involved. I am blessed that there are so many people to thank – editors, contributing authors, reviewers, test readers and colleagues.

Two of the people whom I had the honor to be guided by, Jo Stichbury and Jim Coplien, I met for the first time on the same day. The place was the ACCU conference in 2006 and the idea of writing the next book about Java ME for Symbian OS was too embryonic to be called 'an idea'. Jo Stichbury (who had just returned from Nokia to Symbian) knew from her own authoring experience how to deliver a book from start to finish, and I am thankful for her personal guidance, her professionalism and wisdom as a technical editor, which has always managed to find the right solution at any time and on any matter.

I am also thankful for meeting Jim Coplien, a truly visionary guy. Full of passion, equipped with his one-of-a-kind way of thinking, he has that rare long memory into the past of software engineering and a vision into the future of what will come out of all that we are currently doing.

I am also in debt to the wonderful people who have contributed from their knowledge and experience: Ivan Litovski, Sam Mason and Daniel Rocha. Ivan has been a great friend from day one in the Java group at Symbian, and is a reviewer and contributing author to this book. Sam is a passionate mobile expert and the founder of Mobile Intelligence in Australia. I have been lucky enough to work with Sam on this book and to meet and become friends at JavaOne 2008. Daniel has constantly given this project wonderful support and his contributions, from his viewpoint inside Forum Nokia, have been invaluable. He came to the rescue on many occasions, and I'm truly appreciative of his hard work and energy.

I'd also like to thank Mark Shackman for the pleasure of working with him during my four and a half years in Symbian. I am grateful for his constant insistence on the build up of logical arguments and perfect phrasing. Mark always delivers, and he manages to get the best out of hundreds (by now probably thousands!) of embryonic articles and book chapters.

Thanks also to Martin de Jode for his valuable comments from his own experience in writing the second book in this series and much gratitude to Satu McNabb for her support and for always being there to help.

I'd also like to acknowledge:

- the people at John Wiley & Sons for working with us on this book with their high standards of professionalism and quality
- my current group, Engineering Services in CSG, Sun Microsystems, which had the ability to see the future of Java from the remote distance of the 1990s and leads the way towards the next generation of Java technologies
- the Java group in Symbian, of which I had the pleasure to be a member during most of my time in London
- the Developers' Consulting team and the Developer Product Marketing team in Symbian
- the many people who took time and patience to read all the suggestions and ideas that were yet to become chapters: Arnon Klein, Assaf Yavnai, Yevgeny (Eugene) Jorov, Jonathan Knudsen, Tomas Brandalik, Erik Hellman, Per Revesjö, Magnus Helgstrand, Lars Lundqvist, David Mery, and Emil Andreas Siemes
- my parents and sister and all of my friends.

Finally, this book is for my friend, partner and wife – Gabi – and our wonderful, beloved children, Noa and Ariel.

For being blessed with everything that has happened and the things that are yet to happen.

I hope you will enjoy reading this book.

Symbian Press Acknowledgements

Symbian Press would like to thank Roy, Sam and Daniel for the hard work and energy they've put into this book. Roy piloted this project with particular passion, and the results you hold before you are testament to his efforts and those he inspired in his co-authors. We were privileged indeed to put together such a team of talented writers and developers, and they in turn were fortunate to be able to draw on an extensive and experienced support crew, including Sam Cartwright, Martin de Jode, Ivan Litovski and Mark Shackman.

We'd like to thank everyone involved in writing and reviewing the text of this book and in helping prepare the final manuscript. As usual, we're grateful to the Wiley team (Birgit, Colleen and Claire), and to Sally Tickner and Shena Deuchars.

In particular, Jo would like to thank Antony and Bruce for supporting the completion of this title.

Part One

Introduction to Java ME and Programming Fundamentals

1

Introduction to Java ME, Symbian OS and Smartphones

Symbian OS is the operating system that powers more than 70% of smartphones worldwide. In addition to providing one of the industry's most powerful native platforms for after-market applications, Symbian OS pushes the boundary further by allowing third-party software developers to work with a wide variety of mobile technologies including Symbian C++, Flash Lite, Python, POSIX-compliant C and, of course, Java ME – the focus of this book.

The Symbian OS ecosystem has flourished over the last decade and, like any success story, is made up of many parts. Handset manufacturers, such as Nokia, use Symbian OS as the foundation stone of their user-interface platforms. In Japan alone, Symbian OS powers over 30 million phones that use NTT DoCoMo's MOAP platform. The latest version, Symbian OS v9, is used in almost all of Nokia's famous Nseries (feature-focused) and Eseries (business-focused) devices.

In this chapter, we explore the Java Platform, Micro Edition (Java ME) technologies and see how they are uniquely positioned to address the on-going mobile phone software revolution. We start with a look at recent history, follow with a brief discussion of the Java language itself, the composition of Java ME and the benefits it inherits from Symbian OS. Finally, we wrap up with a look at the mobile phone market and get a glimpse of what the future holds.

1.1 2003: Rise of the Mobile

One of the problems with turning points in history is that they're often only apparent in retrospect. Don't expect to see Monday 16 June 2003 in any documentary about days that changed the world because it's not there – I checked. However, by the end of that day, a startling new

concept in mobile device hardware had been released, ushering in an era of exponential progress in mobile computing.

On that rather special Monday, Nokia released a completely new type of mobile phone. The Nokia 6600 was the most advanced mobile phone anyone had seen, and it is still remembered today as a clear indication of where the future was headed. It had a large, 16-bit color screen, a four-way joystick for navigation and games, and a VGA, 640×480 camera for video recording and photography. The phone was a 2.5 G, Internet-enabled GPRS phone running Symbian OS v7.0s with 6 MB of memory for storage, 3 MB RAM and had Bluetooth wireless technology connectivity. It was also easy to use and looked great at the time (it looks like a bit of brick today, as you can see from Figure 1.1).



Figure 1.1 Nokia 6600

This new device was clearly going to cause quite a stir, so the Nokia marketing team came up with a special phrase just for the launch – something that was prophetic and sounded really cool: Image is Power.

Less important at the time but far more relevant to us today, the Nokia 6600 was the first mobile phone to come with the very latest in Java technology – the Mobile Information Device Profile (MIDP) 2.0, which had recently been finalized. Java ME was already dominating the mobile software market; it was about to take a huge leap forward and the Nokia 6600 was the springboard. Even today, I keep my Nokia 6600 in a special place and consider it with reverence.

In 2003, while the rest of the software industry was dragging itself out of the Dot-com Bust, the constrained-device development sector was at the start of a vertical rise. Hundreds of business applications had already been written and shipped in expectation of an amazing shift in the way people did business. The momentum for mobile hardware and software

built over the years following the release of the Psion Organiser 1 in 1984 and, by 2003, millions of portable information devices had sold around the world. These included pagers, iPods and generic MP3 players, personal organizers (both the original handheld computers and the latest PDAs), *Game & Watch*¹ devices and, of course, mobile phones.

People were getting busier and more 'connected'. Business was becoming more demanding, requiring decisions faster, for higher stakes, and 24 hours per day. It was an exciting time and mobile technology, it seemed, had all the answers. Except that it didn't work out that way.

What the oracles didn't predict was that while there would be a worldwide explosion in the use of mobile phones, they would largely be used to send text messages, buy ringtones and wallpapers, and take photos or video. People didn't buy them to do business after all. They bought them as personal statements and as fashion accessories. They bought them as a convenient communications device. They also bought games. Quite a lot of games, actually. Thousands of titles were developed and sold directly to the public or shipped on devices. The market for mobile phone games quadrupled between 2002 and 2003 and it is estimated that over 50 million phones that allow after-market game installation were shipped in 2002 alone.

Fast forward five years to 2007 and there are nearly three billion mobile phone subscribers worldwide.² The mobile phone has now become the personal information accessory of choice – everyone is 'texting' using appalling new pseudo-grammars (wch is gr8 if u cn do it), crowded buses sound like call centers, the mobile games market has exploded into the public eye with huge titles, such as ***The Sims 2***,³ and in many parts of the world a new kind of digital life has emerged and become the norm. The newest devices are a far cry from the Nokia 6600. January 2007 saw the iPhone launched into the marketplace, followed closely by the Nokia N95 in March of the same year with the slogan 'It's what computers have become'. In October 2008, the T-Mobile G1 became the first smartphone to use the Android platform. Clearly innovation and demand caused many changes in just those five years!

Let's look at some of the figures to put this into perspective – in 2007, mobile handset annual sales exceeded 1 billion units for the first time. That means that more than 2.7 million phones were sold *each day*. Moreover, sales of Nokia handsets alone in 2007 were larger than the entire industry in 2001 and 2002! Figure 1.2 charts annual sales and shows how the market almost tripled over a six-year period, with an annual growth rate of nearly 20%.

¹ If you missed the 1980s, find out about the *Game & Watch* craze at en.wikipedia.org/wiki/Game_%26_Watch

² Global cell phone penetration reached 50% in November 2007, according to www.reuters.com/article/technologyNews/idUSL2917209520071129

³ www.thesims2.com

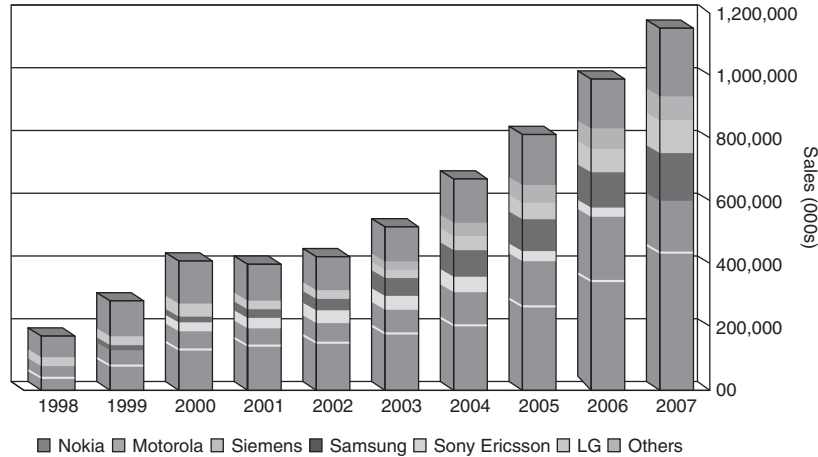


Figure 1.2 Annual number of devices sold⁴

Add five more years and it is 2012. At current growth rates, it is estimated that smartphones will comprise nearly 30% of the market (compared to 10% in 2007) and that subscriber levels worldwide will reach 3.5 billion. Just think about that number – we can’t really conceive figures like this in our minds, but this is a *really* big market. Eight out of ten devices support some form of Java-based technology – quite an opportunity for the budding technically-minded entrepreneur. If you sell every thousandth person a \$1 Java ME MIDlet, pretty soon you’ll have made over \$3 million – job done, time to retire.

1.2 2008: Mobile Generation

Our world has changed. Like all great upheavals, the ‘rise of the mobile’ has had both positive and negative effects on us. As a society, we’re irrevocably addicted to our information devices now. Everywhere we look we see iPod MP3 players, Tom-Tom personal navigation devices, mass-market feature phones, smartphones, managers who can’t get off their Blackberries, people talking while walking, people setting up meetings, messaging, listening to music, reading, using devices for entertainment, organization, romance, learning. It goes on and on.

Mobile information devices are the instruments of an unprecedented wave of fundamental social change. The very basis upon which we interact with each other has suddenly changed without notice and the results are not necessarily all good. SMS bullying is on the rise; onboard cameras have changed our expectations of personal privacy; train carriages are

⁴ java.sun.com/developer/technicalArticles/javame/mobilemarket

like mobile offices; and I've yet to go to a funeral where someone's mobile doesn't ring during the eulogy. It's fashionable to talk about people being more 'connected', as I did above, but a world full of people with iPods and earphones can feel like an increasingly isolated one. Even our basic standards of interaction are affected – people think nothing of sending text messages while talking to you and the legalities involved with using evidence from mobile phones are still being ironed out.

On the positive side though, the plethora of lifestyle applications and services available on mobile phones now definitely makes life a lot more interesting. GPS and navigation applications make it hard to get lost in most places in the world and you can send goodnight kisses to your kids via MMS from wherever you end up! Messaging is by nature asynchronous so you don't *have* to deal with inter-personal communication immediately any more, which allows you to manage your time better. And if you have a two-hour train trip to work why not listen to a podcast on some topic of interest instead of reading the newspaper?

I was thinking the other day about the phrase 'Information is power' and about mobile phones, games, Symbian OS, business applications, Nokia, the future, mobile phones (again), robots, Linux and just general technology. As sometimes happens, I had an acronym attack and realized that information is indeed POWER because it affords us Pleasure, Opportunity, Wealth, Experience and Reputation. To me, this acronym spells out five very good reasons to start learning Java ME today. If you're not yet convinced, stick with me, since the rest of this chapter, and the book, considers the use of Java technology, specifically the Java ME platform, as the candidate to address the growing hardware diversity and the challenges of the future.

1.3 Meet the Host – Symbian OS

Symbian OS is a market-leading, open operating system for mobile phones. Cumulative sales of Symbian smartphones reached 226 million units worldwide in June 2008. By the end of that month, more than 250 different models of smartphone had been created by handset manufacturers licensing Symbian OS; eight handset vendors, including the world's five leading vendors, were shipping 159 mobile phone models.⁵

On 24 June 2008, the Symbian Foundation was announced⁶ and, at the time of writing, it is on track to create an open and complete mobile software platform, which will be available for free, enabling the whole mobile ecosystem to accelerate innovation. The Symbian Foundation provides, manages and unifies Symbian OS, S60, MOAP(S) and UIQ for its members. It is committed to moving the platform

⁵ www.symbian.com/about/fast.asp

⁶ www.symbian.com/news/pr/2008/pr200810018.asp

to open source within two years of the Foundation's announcement. See www.symbianfoundation.org for more information.

There have been many operating systems available over the last decade – so what is it about Symbian OS that has enabled it to stay so far ahead of the competition? The design of Symbian OS is highly modular, which means that it is constructed from well-defined, discrete parts. Most Symbian OS components expose a Symbian C++ API⁷ and a large number of these APIs are available to third-party application developers.

At a high level, Symbian OS can be thought of as a layered model, with a flexible architecture that allows different UI layers to run as platforms on top of the core operating system. That is, Symbian OS provides a framework that supplies a common core to enable custom components to be developed on top of it. The generic UI framework of Symbian OS supplies the common behavior of the UI and supports extension by licensees who define their own look and feel. For example, the S60 platform, supplied by Nokia, sits on top of Symbian OS. S60 is the world's most popular smartphone platform, and is currently used by Nokia, LG and Samsung. Almost 180 million S60 devices had been shipped by S60 licensees by the end of June 2008.⁸

The power and configurable design of Symbian OS are the key contributing factors to its success. Today, Symbian OS devices are being sold in every market segment as you can see from the variety of hardware shown in Figure 1.3. If you want to find out more about Symbian and Symbian OS, there are a range of resources including books from Symbian Press, available from developer.symbian.com/books, and www.forum.nokia.com. If you are new to Symbian OS and interested in a slightly more detailed overview about Symbian OS, a recommended resource is the first chapter of [Babin 2007].⁹

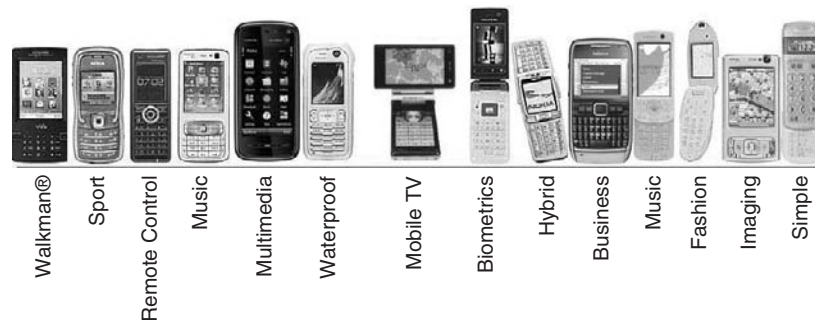


Figure 1.3 Symbian OS is everywhere

⁷ Symbian C++ is a dialect of C++ with idiomatic extensions.

⁸ www.forum.nokia.com/main/platforms/s60

⁹ The chapter can be downloaded from developer.symbian.com/main/documentation/books/books_files/dasos.

1.4 What Is Java?

Let's jump back to August 1998. The word 'Java' is synonymous with the Internet in the public mind. The most common question at the time is 'What is Java?' If you can explain that then you have a job.

The company behind Java technology is Sun Microsystems, which was founded in 1982 and today provides a diversity of software, systems, services, and microelectronics that power everything from consumer electronics, to developer tools and the world's most powerful data centers. Other than Java technology, the core brands include the Solaris operating system, the MySQL database management system, StorageTek¹⁰ and the UltraSPARC processor.

In the early 1990s, Sun formed the 'Green Team' whose job it was to prepare for the future. One of the team members was James Gosling (known as 'the father of Java'). The aim was to create a software platform that could run independently of the host platform – and the 'Write Once Run Anywhere' mantra was created. By creating a controlled virtual environment, or machine, software written on one hardware platform could run unaltered on any other hardware platform.

The Java programming language was introduced by Sun Microsystems in 1995 and is designed for use in the distributed environment of the Internet. It was designed to have familiar notation, similar to C++, but with greater simplicity. The Java programming language can be characterized by:

- Portability: Java applications are capable of executing on a variety of hardware architectures and operating systems.
- Robustness: Unlike programs written in C++, Java instructions cannot cause a system to 'crash'.
- Security: Features were designed into the language and run-time system.
- Object orientation: With the exception of primitive types, everything in Java is an object in order to support the encapsulation and message-passing paradigms of object-based software.
- Multithreading and synchronization: Java supports multithreading and has synchronization primitives built into the language.
- Simplicity and familiarity: This is, of course, relative to C++. Don't imagine that Java can be learned in a day.

Java is a technology rather than just a programming language. It is a collection of software products and specifications that provide a system

¹⁰ StorageTek is a tape-based storage solution from Sun. See www.sun.com/storagetek for more information.

for development and deployment of cross-platform applications. After you've worked with Java technologies for a while, you'll realize that they're also much more than that – they're a way of thinking about software systems and how they work together.

Java is an interpreted language. This means that program code (in a `.java` source file) is compiled into a platform-independent set of bytecodes (a `.class` file) instead of into machine-specific instructions (which is what happens to a C program). These bytecodes are then executed on any machine that hosts a Java platform, a software-only platform that runs on top of a native platform (the combination of an operating system and the underlying hardware). A Java platform is divided into two components:

- The Java virtual machine (JVM) that executes the Java language bytecode can be ported to various native platforms.
- The collection of Java APIs are ready-made software libraries that provide programming capabilities.

Together, the JVM and the APIs form the Java platform which insulates an application from the native platform.

This was a remarkable achievement but it was not achieved without cost. Support for low-level power and fine-grained control was sacrificed, as well as explicit memory management and pointers. It was almost impossible to write Java software that would corrupt memory, but programmers couldn't dispose of memory themselves (to the horror of many there was a `new` keyword but no `delete`); instead it was handled automatically by the run-time environment using the 'garbage collector'. This was part of the environment that monitored object references at run time and would decide when to reclaim allocated memory on behalf of the program.

Added to all of these new ideas was the 'applet'. This probably had the largest role in making Java a 1990s buzz word. Java applets are small programs that can run inside a web browser. Code resides on a remote server, is downloaded over the network and is executed locally inside a 'sandbox', restricting operations to those considered 'safe'. At the same time, class file verification and a well-defined security model introduced a new level of software security.

Java applets had limited use and were quickly outdated by the rise of Flash as a browser-based, client-side technology but applets had served a special role in the Java world. They demonstrated the power of this new language called Java, raised awareness of the Internet and helped create a strong worldwide Java developer base that today numbers over six million.

As time went on, Java moved to have more of a focus on desktop applications and then server applications: suddenly, everyone was talking

about Enterprise Java Beans. Following this was a big leap to smart cards and the emerging market with the rise of mobile phones.

Sun quickly realized that while there were Java solutions for all of these areas, no one set of technologies could adequately address all requirements. It was at this point that Java was separated into the four core technologies that we see in Figure 1.4.¹¹

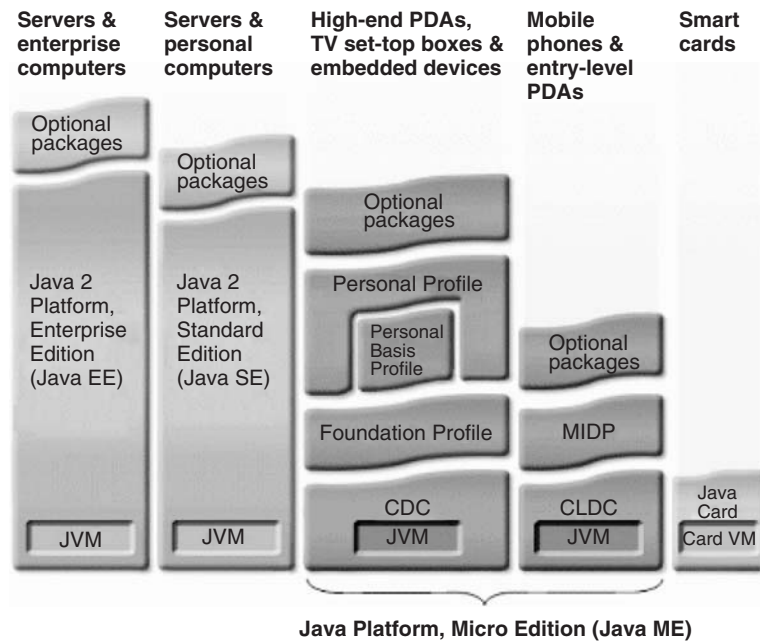


Figure 1.4 Java technologies

The Java Platform, Micro Edition (Java ME) fork was created to address the constrained-device market. Java Card is a separate technology that has nothing to do with Java ME. These days, most mobile phones include some kind of Java technology. Blackberries run a superset of Java ME, extended with RIM libraries. Google's Android platform uses a 'Java-like' platform and even Microsoft's Windows Mobile and Pocket PC support Java ME. Nokia's Series 40 phones, which form the backbone of the feature phone market, all have support for Java ME as do *all* phones running Symbian OS.¹²

If you want to know more about the other Java technologies or learn about the Java language, this is probably the wrong book for you. If that's what you want, then probably the best place to start learning about Java

¹¹ Java ME consists of two 'flavors', which is why there are five groups in Figure 1.4.

¹² Except for MOAP(S), which has DoJa; it is very similar to Java ME, but is not strictly the same.

is straight from the creator himself, James Gosling in [Arnold, Gosling and Holmes 2005]. From this point on, we're only going to talk about ME – Java ME!

1.5 Java ME

Sun launched a research and development project called Spotless in the 1990s in order to produce a JVM that could run on devices with small memory and power budgets, intermittent or non-existent connectivity, limited graphical user interfaces, divergent file system concepts (if any) and wildly varying operating systems. This could be anything from set-top boxes to Internet phones, parking meters, digital TVs, vending machines, automotive interfaces, electronic toys, personal organizers, household appliances, pagers, PDAs, high-end smartphones and mass-market feature phones.

The result of the project was the Kauai virtual machine (KVM). This was a cut-down version of the JVM that required less than 10% of the resources needed for the desktop standard edition of Java. This was achieved by targeting the size of the virtual machine and its class libraries, reducing the memory used during execution and by creating a pluggable architecture that allowed sub-systems of the KVM to be tailored to specific device architectures.

With such a diverse range of hardware targets, it was decided to adopt a more flexible model for Java ME and the final result consisted of three high-level components that together make a wide range of solutions possible. Java ME consists of:

- configurations
- profiles
- optional packages.

We'll look at each of these in detail shortly. In order to allow for future expansion and new technologies, additions to the Java ME technologies are managed through the Java Community Process¹³ in the form of Java Specification Requests known as JSRs. Each JSR is reviewed by panels of experts and public drafts allow a wide range of interested parties to contribute to the process. JSRs are usually referred to by their name and number; for example, JSR-82 is the specification request for the Bluetooth API.

A lot of functionality had to be removed from the desktop Java Standard Edition (Java SE) to make Java ME. For example, some things that are missing include:

¹³ www.jcp.org

- reflection
- thread groups and advanced thread control
- user-defined class loaders
- the Java native interface (JNI)
- automatic object finalization.

Compatibility played a large part in the design of Java ME. In order to maintain as much compatibility with Java SE as possible, the Java ME packages were divided into two logical groupings: those that were specific to Java ME (packages whose names start with `javax`) and those that were a subset of their Java SE counterparts which followed the same package-naming convention.

One of the strongest features of the Java language is its security – both at the application layer and within the virtual machine itself. The VM security layer ensures that bytecodes are valid and safe to execute. This works really well on servers and the desktop environment but not so well on mobile phones with small processors and small power and memory budgets. The solution was a hybrid approach using the concept of *pre-verification* performed offline as part of the build process on the development machine. Only after VM acceptance may the bytecodes execute on the host hardware.

1.5.1 Configurations

Configurations define the lowest common denominator for a horizontal category of devices that share similarities in terms of memory budgets, processor power and network connectivity. A configuration is the specification of a JVM and a base set of necessary class libraries which are primarily cut-down versions of their desktop counterparts. There are currently only two configurations:

- The Connected Limited Device Configuration (CLDC) was aimed specifically at mobile phones, two-way pagers and low-level PDAs.
- The Connected Device Configuration (CDC) was aimed at devices with more memory, better network connectivity and faster processors.

We should be clear, however, that the CDC profile doesn't really have a role in the mobile phone space at this time. Although high-end devices, such as Symbian OS phones, can accommodate a CDC-based Java platform, the consumer market today is based on MIDP/CLDC applications, so CDC is not within the scope of this book.

Table 1.1 CLDC 1.1 Packages

Package	Description
<code>java.io</code>	Basic classes for I/O via data streams, Readers and Writers
<code>java.lang</code>	Reduced math library, threads and system functionality, fundamental type classes
<code>java.lang.ref</code>	Support for weak references which allows you to hold a reference to an object even though it is garbage collected
<code>java.util</code>	The Vector and Hashtable classes, basic date and calendar functionality and a pseudo random number generator which is great for games
<code>javax.microedition.io</code>	CLDC specific interfaces and factories for datagrams, connection stream etc. Implementation deferred to profiles.

So far there have been two versions of the CLDC,¹⁴ the latest being CLDC 1.1. Most mobile phones shipped since 2003 contain version 1.1 of the CLDC. CLDC 1.1 (see Table 1.1) includes, amongst other updates, support for floating-point operations with the introduction of the `Float` and `Double` classes, thread naming and interrupts. The total minimum memory requirement was also lifted from 160 KB to 192 KB.

While the discussion so far has focused on mobile phones, the CLDC is not just for phones. As we'll see in the following section, it forms the foundation for a number of profiles that target completely different device families.

1.5.2 Profiles

Profiles sit on top of a configuration and allow it to be adapted and specialized for a particular class of devices in a vertical market such as mobile phones. A profile effectively defines a contract between an application and a set of devices of the same type, usually by including class libraries that are far more domain-specific than those available in any particular configuration.

Figure 1.5 shows the three profiles that currently extend the CLDC. The Mobile Information Device Profile (MIDP) is most relevant to our work here; the Information Module Profile¹⁵ (IMP) targets small devices that may have no user interface at all, such as parking meters, call boxes, and

¹⁴ See the white paper at java.sun.com/j2me/docs/pdf/CLDC-HL_whitepaper-February-2005.pdf.

¹⁵ java.sun.com/products/imp/index.jsp

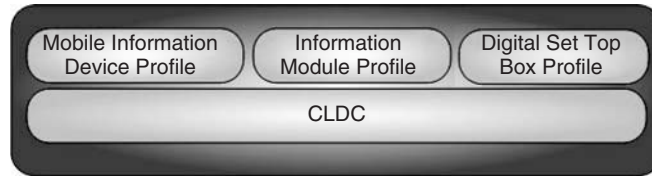


Figure 1.5 CLDC profiles (adapted from [Ortiz 2007, Figure 5])

vending machines; and the Digital Set-Top Box Profile is aimed squarely at the cable television market.

CLDC is also used by the proprietary DoJa profile for i-mode devices from NTT DoCoMo, which dominates over 50% of the Japanese market. We talk more about DoJa in Chapter 7.

Applications that use the MIDP libraries are called *MIDlets*. MIDP defines the core functionality that mobile applications can leverage (for example, the user interface (UI), local storage and network connectivity) and also specifies application lifecycle management. The original version of MIDP was started in November 1999; even though it did not specifically provide libraries for games, that didn't stop hundreds of titles being written and sold worldwide. In fact, so successful was MIDP 1.0 that work began in 2001 on defining the second version – MIDP 2.0.

The MIDP 2.0 libraries include a high-level UI API with support for standard widgets (textboxes, check boxes, buttons, etc.) and a low-level UI API supporting graphics contexts aimed mainly at games development. The low-level API includes standard routines for creating graphics primitives (lines, polygons, arcs, etc.) as well as supporting clipping, full-screen canvases, regions, image manipulation, fonts, brushes, colors, and off-screen buffers. Support for local persistence on the MIDP is provided through the record management system (RMS). The CLDC Generic Connection Framework was extended with additional connections, such as HTTP, TCP and UDP.

MIDP 2.0 also introduced timers, the Media API (a subset of the optional JSR-135 Mobile Media API) and the Game API package which enables the development of rich gaming content for mobiles.

1.5.3 Optional Packages

The optional general-purpose APIs provide a set of functionality that is not specific to any particular class of devices and have allowed the Java ME platform to evolve and embrace emerging technologies. They add extra functionality to a profile and are sometimes incorporated into the profile as the technology matures. The Mobile 3D Graphics for J2ME API (JSR-184) and the Location API (JSR-179) are good examples of optional packages. It is common to simply use the term 'JSR' when referring to

the set of packages a device may support, although configurations and profiles are actually all the product of particular JSRs.

Java ME got off to a bumpy start – the realities and pressures of business competition, largely due to the phenomenal rise of the mobile games industry, led to a wide disparity in optional JSR support across the set of available devices in the market. There was no standard set of JSRs available, some implementations didn't comply with the defined JSR requirements and some only offered partial compliance; it was extremely difficult even to find out which JSRs were supported on any particular handset. Sometimes irregularities occurred within the same family of mobile phones from the same manufacturer, with APIs missing for no obvious reason. Manufacturers would also add their own proprietary APIs to allow non-standard functionality or extra functionality that wasn't covered by the standard JSRs. While that addressed the short-term need, such libraries had to stay around for compatibility reasons long after they had been superseded.

The net result was a series of device-specific workarounds leading to complex code forks and build trees for Java ME applications – particularly games which would typically target hundreds of handsets at any one time. Game development houses wrote entire libraries trying to work across these heterogeneous environments and then had to work to recoup the investments made. This unfortunate phenomenon is known as 'fragmentation' and the 'write once, run anywhere' mantra became 'write once, debug everywhere'. This had an extremely detrimental effect on the time to market of Java ME applications and games.

1.5.4 Tackling Fragmentation

Something had to be done. The first attempt to tackle fragmentation was JSR-185, Java Technology for the Wireless Industry (JTWI), which specified a minimal level of optional API support for JTWI-compliant devices. Release 1 of the JTWI was approved in July 2003 and defined three categories of JSR: mandatory, conditionally required, and minimum configuration. A handset that declares itself to be JTWI-compliant has CLDC 1.0 or CLDC 1.1 as a minimum configuration and must support at least:

- MIDP 2.0 (JSR-118)
- Wireless Messaging API (JSR-120).

If the device allows MIDlets to access multimedia functionality via APIs, then the JTWI also specifies that the Mobile Media API (JSR-135) is required (i.e., it is conditionally required).

JTWI also defined a number of implementation parameters for these JSRs, including support for JPEG encoding, MIDI playback, the number of

record stores allowed in the RMS, Unicode support and some threading requirements. While this was a great start, more needed to be done. JTWI served as a necessary consolidation point in the evolution of the mobile industry, but there were still many JSRs in the world of Java ME.

The Mobile Services Architecture¹⁶ (JSR-248, MSA), finalized in December 2006, mandated that an MSA-compliant device is JTWI-compliant and implements a much wider range of JSRs. The specification defines two categories of MSA, known as Full MSA and MSA Subset (see Figure 1.6).

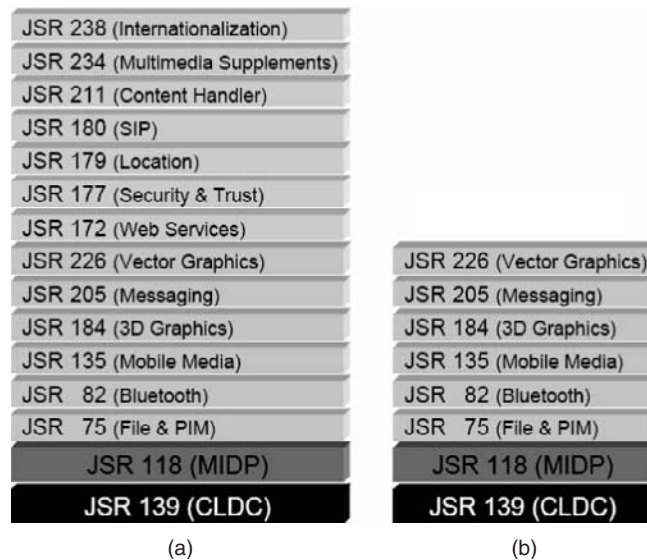


Figure 1.6 Mobile Services Architecture: a) Full MSA and b) MSA subset (adapted from [Ortiz 2007, Figure 3])

MSA is a huge step forward and the number of MSA-compliant devices available in the market is increasing all the time. Have a look at the Java ME device registry at Sun, which lists some of them, to see how MSA has raised the bar. (For more information on MSA, see Chapter 6.)

1.6 Why Use Java ME on Symbian OS?

Why not just use Symbian C++? That's a good question. To answer it, we need to look firstly at what people do with their smartphones and then how it is achieved technically.

¹⁶ jcp.org/en/jsr/detail?id=248

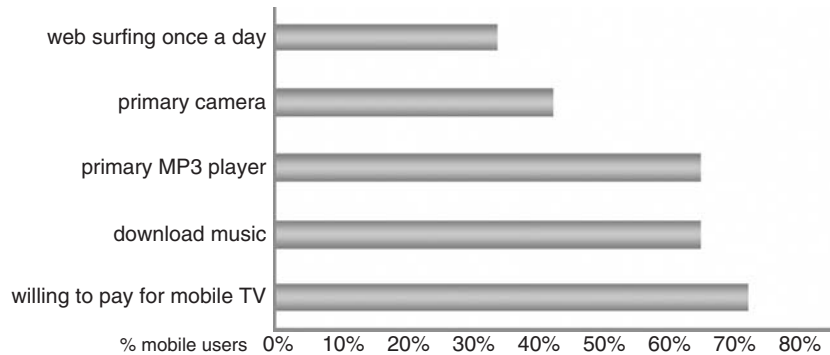


Figure 1.7 Smartphone usage¹⁷

Figure 1.7 shows the results of a survey of how people are willing to use their smartphones. In addition, if we were to ask a random sample of people in the street what they do regularly, we'd also expect them to include activities such as playing games, sending text messages, sending pictures, social networking, checking match scores, finding directions, watching videos, checking weather or traffic – and annoying other people on buses.

It is clear that there are many uses for smartphones, most of which involve downloading mobile content, such as installable applications and games, purchasing multimedia, such as music, videos, and mobile TV, or simply downloading web pages while browsing the Internet. 'Content' is the key word here. A high-end mobile platform cannot exist without content; in its absence there's nothing that compels the user to buy a high-end device. Who would buy a fully featured smartphone to make voice calls and send SMS messages only?

In theory, it is possible to restrict all installable content to that created using Symbian C++, in order to control where it comes from. That just results in a 'walled garden', which goes against the principle of Symbian OS as an open platform, because millions of talented engineers who are not well-versed in Symbian C++ cannot contribute to the platform and any existing content not written in Symbian C++ cannot be reused. That would be unduly limiting, so the Symbian strategy is to enable environments that maximize third-party developer contributions and content re-use. Apart from Java ME, the other run-time environments available include POSIX-compliant C, Flash Lite, Silverlight, Python, Ruby, Web Runtime, and WidSets.¹⁸

So a better question is "when should I use Java ME on a highly powerful and open environment, such as Symbian OS?". To work out the answer to

¹⁷ 'Welcome to the Smartphone Lifestyle' by Nigel Clifford CEO of Symbian Ltd, Symbian Smartphone Show, 2007 keynote address

¹⁸ developer.symbian.com/main/documentation/runtime-environments

that question you need to do a cost–benefit analysis which covers areas such as the available skills in your organization, functional feasibility in Java, development effort in Java compared to other options, performance considerations, and cross-platform requirements, all framed by the profile of your application.

Let’s take a quick look into how the ‘leading Java platform’ strategy translates into technical support. The Symbian implementation of the Java ME technology platform is best of breed, so we’ll run through some of the benefits that Java ME enjoys under Symbian OS.

Optimized Performance

The most common criticism leveled at Java ME (or any version of Java) is that it doesn’t execute as fast as native code because it’s an interpreted language (in principle, the JVM interprets Java bytecodes at run time). This is mentioned so often, you’d think that everyone was writing real-time satellite control systems instead of games for mobile phones. Java runs much faster than it used to: the KVM was placed in the Java museum long ago. Symbian replaced the KVM with CLDC-HI HotSpot, in which the application bytecode is compiled to the ARM instruction set on the fly. As a result, Java applications received a major performance boost that was not only measurable by objective benchmarks but also by the user – which is the *real* benchmark in most mobile applications.

No Hard Limits on Computing Resources

On most feature phones, the memory assigned to Java is fixed when the device is created, which causes a wide variety of challenges for advanced application development on these platforms. Symbian OS does not place arbitrary constraints on available memory, storage space, number of threads, number of connections, or JAR size. Within the bounds of available resources, there are no limits – the heap can grow and shrink as required.

Consistency and Variety of JSRs

Probably the greatest benefit of working with Java ME on Symbian OS is the consistency and variety of JSRs. The problems of JSR fragmentation are largely a thing of the past. The set of optional JSRs supported has grown with each new version of the operating system and the latest smartphones expose some of the very latest in technology directly to MIDlets.

With this wide variety comes both implementation consistency and the ability to leverage the best of the operating system. For example, releases of Symbian OS v8.0 and later support the full Mobile Media API (JSR-135) and provide native support for OpenGL ES 1.0 (an abstraction layer for the Mobile 3D Graphics API (JSR-184), meaning that your MIDlets automatically benefit from hardware acceleration on the handset). Symbian

Table 1.2 JSR Support on Symbian OS

	JSR	Nokia 6,600	Nokia N70	Nokia N96
Symbian OS version		v7.0 s	v8.1a	v9.3
Release Date		16 June 2003	27 April 2005	11 Feb 2008
CLDC 1.0/1.1		1.0	1.1	1.1
MIDP 2.0/2.1		2.0	2.0	2.1
Bluetooth	82	✓		✓
Mobile Media (MMAPI)	135	✓	✓	✓
Wireless Messaging	120	✓	✓	✓
Wireless Messaging	205 ¹⁹			✓
File and PIM	75		✓	✓
Mobile 3D Graphics (M3G)	184		✓	✓
JTWI	185		✓	✓
Web Services	172		✓	✓
Location API	179			✓
Security and Trust Services	177			✓
SIP API	180			✓
Scalable 2D Vector Graphics	226			✓
Advanced Media Supplements	234			✓
MSA	248			MSA Subset

OS v9 enhances a number of existing APIs and adds many more. The net result is a rich diversity of functionality optimized for performance and memory efficiency. Table 1.2 shows a snapshot of how JSR support under Symbian OS has evolved over the last few years.

Summary

When you develop using Java ME and Symbian OS, you are combining the world's leading smartphone operating system with the most successful mobile phone development platform to date. Symbian OS brings consistency, performance, reliability and variety to its Java ME offering while at the same time removing the most common limitations of other operating systems.

¹⁹ JSR-205 was an update to JSR-120 that added programmatic support for MMS.

There are an enormous number of Java ME applications already out there – far exceeding the number of native Symbian C++ applications ever built. The number of Java developers for the mobile environment is also increasing and is undoubtedly bigger than the army of Symbian C++ developers.

Having said all this in praise of Java ME, do keep in mind that there are times when you need to go native, times when you need the maximum speed, real-time performance, fine-grained control or bare-metal access to native services that are not exposed in Java. When you need low-level power, tight integration with the operating system, or advanced features that are not scoped by Java, then your best option is probably Symbian C++. For example, Symbian OS native services allow you to develop hooks into the TCP/IP stack and plug extensions to the Symbian OS Messaging component. Java ME is simply not designed for such requirements. For a further comparison of native Symbian C++ and Java ME on Symbian devices, please see [Mason and Korolev 2008].

1.7 Java's Place in the Sun

There can be no discussion of Java ME without a section about mobile games. This is such a booming and exciting industry that Symbian has already published an entire book just on mobile game development [Stichbury 2008].

Almost as soon as the first Java-enabled handsets reached the public, the mobile games explosion was under way. While MIDP 1.0 was a fantastic start, it was the release of MIDP 2.0 with its Game API that really made the difference. Suddenly, there was a `GameCanvas` class, support for basic sounds via tone generation, support for sprites and collision detection, the ability to take over the entire screen and efficient mechanisms for representing large animated backgrounds and detecting multiple keystrokes during the main game loop.

It was pretty clear that mobile games were out-selling every other type of application across all platforms and the pace was accelerating. A new type of gamer emerged who was very different from the traditional hard-core console jockeys. They didn't sit in darkened rooms eating pizza and pretending to be someone else online – instead people played games on their mobiles while waiting for a bus, walking around or just sitting in a park. In a major research project for N-Gage,²⁰ one of the key findings was that mobile games were played almost as much on the move and when waiting as when at home.

In this market, games needed to be designed to be played in short bursts, to be paused and re-started without warning, to deal with sudden

²⁰ 'Evolution of Mobile Gaming', available from Forum Nokia (www.forum.nokia.com).

loss of battery power or to be interrupted by an incoming call. Ad-hoc networks could be created with Bluetooth wireless technology for multiplayer games and, with the introduction of the Mobile 3D Graphics API (JSR-184,²¹ also known as M3G), artists and designers could create game assets using industry-standard 3D modeling tools and developers could include and animate them in 3D scenes on actual handsets. On Symbian OS phones, the M3G implementation leverages the native support for OpenGL ES that comes with the operating system.

Of course there were problems. Mobile phones, even high-end ones, have limited power, memory, processing and screen area. Floating-point operations present a significant hurdle to be overcome when trying to perform 3D graphics calculations on a phone. It was quickly apparent that JSR-184 could not provide acceptable frame rates in the absence of hardware acceleration.

Now that a few more years have passed, it's pretty common to see mobile phones with dedicated hardware for graphics and floating-point calculations and the latest Symbian OS phones natively support OpenGL ES 1.1. The introduction of OpenGL ES 2.0 will change the entire mobile gaming landscape; it has support for the programmable graphics pipeline (i.e. shaders) and there is already a JSR in progress to work with it – the Mobile 3D Graphics API 2.0 (JSR-297).²²

Realizing that mobile multiplayer game-play was a huge potential market, Nokia launched the Scalable Network Application Package (SNAP) framework²³ in 2003. Network operators love it because it generates revenue by increasing bandwidth usage. SNAP 2.1, the latest version of the SDK, has only just been released at time of writing; it is discussed in Appendix B. We also discuss general games development in more detail in Chapter 8.

1.8 Routes to Market

So you bought this book, learned all about Java ME application development, designed and built the killer Java ME mobile application and you're ready to sell – now what? How do people find and buy mobile applications? In 2006 alone, roughly \$2.5 billion dollars' worth of games were sold worldwide so let's discuss how you can grab a piece of it.

There are currently two main ways to get your applications working for you. The first way is to make a deal with a network operator or carrier for a slot on their 'deck'. The deck is a space on their mobile portal where the top mobile applications (usually games) are made available for

²¹ For a great introduction to JSR-184, check out developers.sun.com/mobility/apis/articles/3dgraphics.

²² www.jcp.org/en/jsr/detail?id=297

²³ www.forum.nokia.com/main/resources/technologies/snap_mobile

subscribers to download. The subscribers usually have some deal where they can download a certain number of applications per month and it simply gets added to their bill.

This is a really difficult path to tread. To start with, you need to be a tough negotiator with a strong sales pitch and a business plan. Secondly, operators tend to look for applications that are unique in some way so that they will sell well, and there are already thousands of titles out there so a mobile version of *Frogger* probably isn't going to cut it. Thirdly, operators tend to avoid making deals with small-timer 'unknown' developers – not only because of quality concerns but also because when something does go wrong, it is the operator that must deal with the irate customer not the developer. Lastly, in order to mitigate some of this risk, operators often insist that MIDlets be Java Verified, which can introduce some additional overhead that needs to be taken into consideration.

However, let's say you ignore all of this and go ahead anyway. The chances are pretty good that the last hurdle will tell you if your application is truly ready. The Java Verified program²⁴ is a quality assurance program run by third-party testing houses that ensure MIDlets meet well-defined and accepted sets of guidelines. MIDlets that are Java Verified can use the Java Powered logo on their splash or 'about' screens which marks the software as having passed a rigorous testing process. This conveys a level of trust in the quality of the software that is widely regarded in industry.

A better way for small developers is to work with content aggregators, companies such as www.gamemobile.co.uk, www.handango.com and www.glu.com with online portals hosting thousands of mobile applications available for sale straight to the public. The details of the process varies, but generally speaking, they host your software for free (or a very small fee) and then take a cut of each sale.

There are distinct advantages to this. The content aggregators handle marketing, billing, payments and invoicing; they are already well known, have disaster recovery plans for their server farms and are likely to have large volumes of Internet traffic which translates directly into increased sales. They also tend to be far more interested in the application than in the fact that you're a small business.

Other areas that need thinking about include a support framework for your application. This can involve taking support calls, responding to FAQs or emails, forum administration, distributing patches, releasing upgrades and, worst of all, processing refunds. We hope this won't happen too often and the best way to prevent it is to get hold of hardware and test it thoroughly. For further information about taking your application to market, we recommend 'Getting to Market', a booklet published by Symbian Press, which can be downloaded from developer.symbian.com/booklets.

²⁴ www.javaverified.com

1.9 Time for a Facelift

As mobile phones become more powerful, the old LCDUI libraries included in MIDP seem less than fresh. At the 2008 Java One Conference, it was clear that a new wave of user interface technologies will quickly dominate the market. Flash Lite is up to version 3.0 and Sony Ericsson announced the Capuchin framework, which creates a bridge between Java ME and Flash Lite. This paves the way for a very different class of constrained-device software.

An increasing number of Java ME applications use a Scalable Vector Graphics (SVG) declarative UI to provide an extremely rich user experience. It even has good performance since, like other graphics libraries, JSR-226 SVG can also take advantage of graphics hardware present on the device.

Understanding that an API alone is not sufficient to promote usage of SVG by application developers, Sun, Nokia and Ikiyo started collaborating on tools for creating SVG content. The demos at Java One convinced us that iPhone-like interfaces are both possible and practical under Java ME. So you can expect to see radical changes in mobile software interfaces over the next few years. (For more information on how to develop an SVG-based UI, please refer to Chapter 6.)

As we'll see in the next section, there is much more to come. Now that there's a clearer picture of what developers have been trying to achieve over the last few years in response to customer demand, Java ME itself is getting its largest facelift in over five years: MSA 2.0 and MIDP 3.0 are on their way.

1.10 Back to the Future: MSA 2.0 and MIDP 3.0

Technological advancements never seem to come as fast they could. It's a bit hard to see at the moment, but some people have compared our current mobile era to that of the PC and the Internet just before things started accelerating. Are we accelerating? A few years from now, the current position of Java ME may be seen as a turning point.

Not everyone has the same experiences, professional or otherwise, and so what follows is necessarily subjective. Over the last few years while working with mobile technologies on platforms that include, but are not exclusively, based on Symbian OS, clients from various industries have asked for some combination of the same few key features in their custom mobile software:

- a better UI
- better integration with the phone

- start-on-boot services that stay in the background, unseen
- application icons placed on the phone's idle screen during installation
- screen savers
- read access to the phone state – battery level, profile, power status, etc.
- the ability to open connections to remote locations while suppressing all user confirmation prompts
- a custom full-screen UI including cascading menus and input widgets
- a service that takes over the whole screen and can't be turned off (in order to create a kind of kiosk mode).

Software is usually targeted at mass-market feature phones. While most smartphone platforms natively expose some of these abilities to developers there is no general solution for Java ME. Clients don't care about published UI style guides, the operator or manufacturer's built-in protection domains, certificates or signing. Quite rightly, they just want their product to work. Clearly there's a large difference between what technophiles believe is reasonable and what the market demands.

Not all of the things mentioned in the above list will be included in MSA 2.0 and MIDP 3.0, and nor should they be. However, when the mass market is saturated with phones that use MSA 2.0 and MIDP 3.0, Java ME will enter a golden age and we'll be well down the road to a market-wide solutions framework. Such a powerful Java ME platform will raise the bar much higher and enrich the Java ME platform with more advanced JSRs. Applications will be able to do a lot more, given wider support for emerging technologies and updated APIs to extend the existing JSRs. Some of the expected highlights are:

- additional MSA 2.0 standardized JSRs
- MIDP 3.0 API enhancements
- LIBlets – class libraries that exist outside MIDlet suites and are the Java ME equivalent of a dynamically linked library; this has far-reaching consequences for code centralization and management, managing upgrades and patches and the security model
- exposure to system events and properties including changes to power status, system idle states, screen-saver events, incoming calls, back-light status, and the battery level
- inter-MIDlet communication (IMC)
- idle-screen MIDlets

- auto-start MIDlets
- screen-saver MIDlets
- built-in splash screen support in the MIDlet class and JAD file
- provisioning and OTA enhancements
- CDC as an option for MIDP.

Full details are in the MSA 2.0 and MIDP 3.0 specifications at www.jcp.org.

Not every desired feature will become available even then (e.g., there will still not be a standard mapping for all the softkeys), but when a single phone combines the power of MSA 2.0 and MIDP 3.0, we'll see some very new Java ME applications. Imagine two concurrent MIDlets communicating over IMC, using advanced JSRs and sharing libraries from an installed LIBlet. After both MIDlets exit, the idle-screen MIDlet launches and is displayed on the home screen. When we can harness power like this, Java ME will truly have found its place in the sun.

1.11 Summary

In this chapter, we've discussed how mobile phones have changed the way we look at the world, exchange information and interact with each other. Never in human history have we faced so many events that affect us so profoundly over such a short period on a daily basis. Future historians will draw a line early in the 21st century marking the end of the old world and the start of something entirely new.

Symbian OS was designed with a long-distance vision for mobile technology. That vision has proven to be practical and is the key to the success of Symbian OS within the smartphone market. Java in all its forms has successfully adapted to a constantly changing landscape for over a decade, breaking new ground and driving new industries. Hopefully by now you're convinced that Java ME is uniquely positioned to address the challenges of the future – not only in the mobile space generally but also specifically in the Symbian ecosystem.

Whether you are one of the six million existing Java developers or are going to be number six million and one; whether you are new to the Symbian ecosystem or already have seniority – we welcome you to Java ME inside the smartphone model!

In Chapter 2, we go back to basics and discuss the fundamentals of Java ME MIDP programming. If you're comfortable with the nuts and bolts of Java ME, you'll probably want to skip that material and turn to Chapter 3, which describes Java ME on Symbian OS. Either way – let's get started!

2

Fundamentals of Java ME MIDP Programming

In Chapter 1, we examined the core Mobile Information Device Profile (MIDP) functionality and outlined the CLDC and MIDP classes that form the development environment. In this chapter, we discuss the basic concepts of the MIDP application model, its component pieces and how they fit together to create a fully functional mobile application.

The goal of this chapter is to build a baseline of information to be used in the rest of the book, such as a basic knowledge of application development in MIDP, the MIDP application model, commonly used packages and classes, the packaging and deployment model, and an overview of some the most important optional APIs. The information presented here is also useful for the Symbian OS Java ME certification exam.

We then look at a basic application example which summarizes the process of creating a mobile application from end to end, and how to run it on an emulator and on a real device based on Symbian OS.

If you are an experienced Java ME developer and familiar with MIDP development, you may want to only browse quickly through this chapter, before you move on to Chapter 3, where we explore Java ME on Symbian smartphones specifically.

2.1 Introduction to MIDP

There are three types of component (see Section 1.5) that make up the Java ME environment for mobile devices such as mobile phones: configurations, profiles and optional packages.

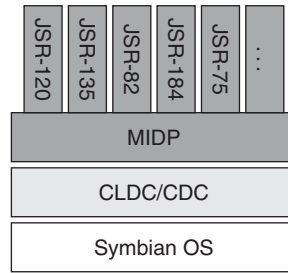


Figure 2.1 Basic Java Micro Edition environment architecture

Figure 2.1 summarizes the architecture of the Java ME environment. Note that Symbian OS is not a mandatory part of the Java ME environment, but we will consider it to be so for the scope of this book.

The MIDP application model defines what a MIDlet is, how it is packaged, and how it should behave with respect to the sometimes constrained resources of an MIDP device. MIDP provides an application framework for mobile devices based on configurations such as CLDC and CDC.

It also defines how multiple MIDlets can be packaged together as a suite, using a single distribution file called a Java Archive (JAR). Each MIDlet suite JAR file must be accompanied by a descriptor file called the JAD file, which allows the Application Management Software (AMS) on the device to identify what it is about to install.

2.2 Using MIDlets

A MIDlet is an application that executes under the MIDP. Unlike a desktop Java application, a MIDlet does not have a `main` method; instead, every such application must extend the `javax.microedition.midlet.MIDlet` class and provide meaningful implementations for its lifecycle methods. MIDlets are controlled and managed by the AMS, part of the device's operating environment. They are initialized by the AMS and then guided by it through the various changes of state of the application. We look briefly at these states next.

2.2.1 MIDlet States

A state is designed to ensure that the behavior of an application is consistent with the expectations of the end users and device manufacturer. Initialization of the application should be short; it should be possible to put an application in a non-active state; and it should also be possible to

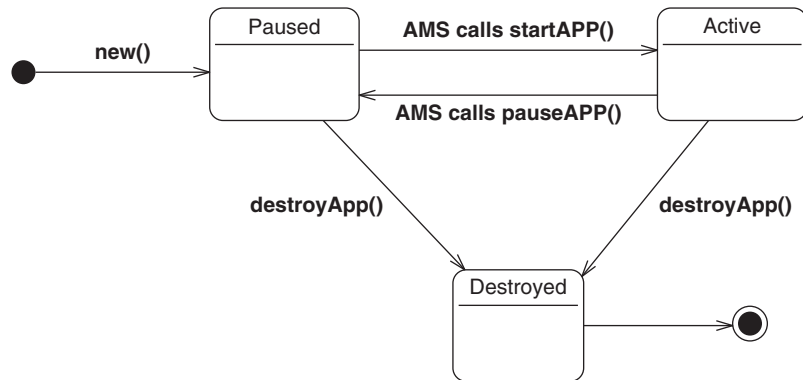


Figure 2.2 MIDlet states

destroy an application at any time. Once a MIDlet has been instantiated, it resides in one of three possible states (see Figure 2.2):

- **PAUSED**

The MIDlet has been initialized but is in a dormant state. This state is entered in one of four ways:

- after the MIDlet has been instantiated; if an exception occurs, the DESTROYED state is entered
- from the ACTIVE state, if the AMS calls the `pauseApp()` method
- from the ACTIVE state, if the `startApp()` method has been called but an exception has been thrown
- from the ACTIVE state, if the `notifyPaused()` method has been invoked and successfully returned.

During normal execution, a MIDlet may move to the PAUSED state a few times. It happens, for example, if another application is brought to the foreground or an incoming call or SMS message arrives. In these cases, the MIDlet is not active and users do not interact with it. It is, therefore, good practice to release shared resources, such as I/O and network connections, and to stop any running threads and other lengthy operations, so that they do not consume memory, processing resources, and battery power unnecessarily.

- **ACTIVE**

The MIDlet is functioning normally. This state is entered after the AMS has called the `startApp()` method. The `startApp()` method can be called on more than one occasion during the MIDlet lifecycle. When a MIDlet is in the PAUSED state, it can request to be moved into the ACTIVE state by calling the `startApp()` method.

- **DESTROYED**

The MIDlet has released all resources and terminated. This state, which can only be entered once, can be entered when the `destroyApp(boolean unconditional)` method is called by the AMS and returns successfully. If the `unconditional` argument is false, a `MIDletStateChangedException` may be thrown and the MIDlet will not move to the DESTROYED state. Otherwise, the `destroyApp()` implementation should release all resources and terminate any running threads, so as to guarantee that no resources remain blocked or using memory after the MIDlet has ceased to execute.

The MIDlet also enters the DESTROYED state when the `notifyDestroyed()` method successfully returns; the application should release all resources and terminate any running threads prior to calling `notifyDestroyed()`.

2.2.2 Developing a MIDlet

Once the source code has been written, we are ready to compile, pre-verify and package the MIDlet into a suite for deployment to a target device or a device emulator.

In this section, we create our first MIDlet in a simple but complete example of MIDlet creation, building, packaging and execution. We use a tool called the Java Wireless Toolkit (WTK) which provides a GUI that wraps the functionality of the command-line tool chain: compiler, pre-verifier and packaging tool.

The WTK (see Figure 2.3) was created by Sun to facilitate MIDP development. It can be obtained free of charge from Sun's website (java.sun.com/products/sjwtoolkit/download.html).

The WTK offers the developer support in the following areas:

- **Building and packaging:** You write the source code using your favorite text editor and the WTK takes care of compiling it, pre-verifying the class files, and packaging the resulting MIDlet suite
- **Running and monitoring:** You can directly run applications on the available mobile phone emulators or install them in a process emulating that of a real device. Your MIDlets can be analyzed by the memory monitor, network monitor and method profiler provided by the WTK.
- **MIDlet signing:** A GUI facilitates the process of signing MIDlets, creating new key stores and key pairs, and assigning them to different security domains for testing with the different API permission sets associated with those domains.

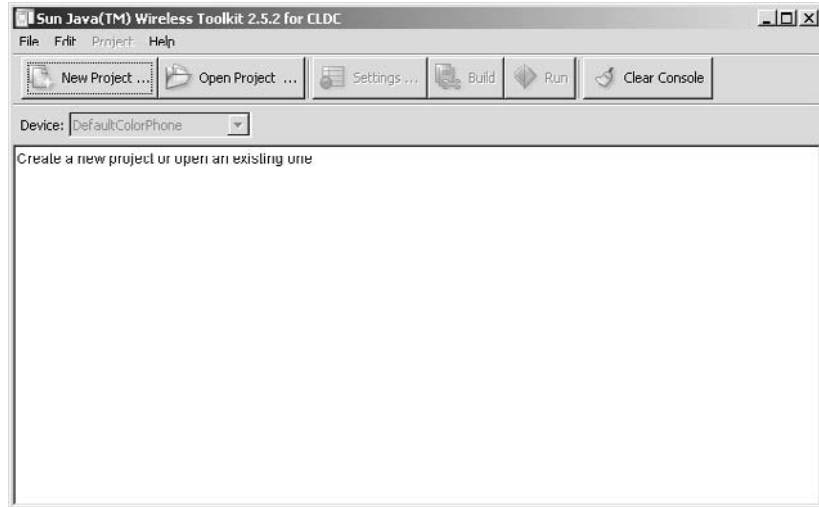


Figure 2.3 Java Wireless Toolkit

- **Example applications:** The Wireless Toolkit comes with several example applications you can use to learn more about programming with MIDP and many optional API packages.

At the time of writing, the WTK is available in production releases for Microsoft Windows XP and Linux-x86 (tested with Ubuntu 6.x). For development, you also require the Java 2 Standard Edition (Java SE) SDK of at least version 1.5.0, available from java.sun.com/javase/downloads/index.html.

Check the WTK documentation, installed by default at `C:\WTK2.5.2\index.html`, for more details on how to use the WTK's facilities for developing MIDlets. Be sure to have installed both the Java SDK and the WTK before trying out our example code.

2.2.3 Creating and Running a MIDlet using the WTK

Now that we are all set with the basic tool, let's create our first MIDlet. The first thing we do is to create a project within the WTK for our new application:

1. Go to Start Menu, Programs, Sun Java Wireless Toolkit for CLDC.
2. Choose Wireless Toolkit and click to start it up.
3. Click on New Project... and enter `Hello World` into the Project Name field and `example.HelloWorldMIDlet` into the MIDlet Class Name field.

4. Click on the Create Project button. In the following screen, change the Target Platform to JTWI and click OK.

Your project is now created, so it's time to write our first MIDlet application. Open a text editor and type in the following code:

```
package example;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Form;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;
public class HelloWorldMIDlet extends MIDlet {
    private Form form = null;

    public HelloWorldMIDlet() {
        form = new Form("HelloWorld Form");
        form.append("Hello, World!");
        form.append("This is my first MIDlet!");
    }
    public void destroyApp(boolean arg0) throws MIDletStateChangeException {}
    public void pauseApp() {}
    public void startApp() throws MIDletStateChangeException {
        Display.getDisplay(this).setCurrent(form);
    }
}
```

In the constructor, we create a new `Form` object and append some text strings to it. Don't worry about the details of the `Form` class. For now it's enough to know that `Form` is a UI class that can be used to group small numbers of other UI components. In the `startApp()` method, we use the `Display` class to define the form as the current UI component being displayed on the screen.

Build your MIDlet with the following steps:

1. Save the file, with the name `HelloWorldMIDlet.java`, in the source code folder created by your installation of the Wireless Toolkit. Usually its path is `<home>\j2mewtk\2.5.2\apps\HelloWorld\src\example`, where `<home>` is your home directory. In Windows XP systems, this value can be found in the `USERPROFILE` environment variable; in Linux, it is the same as the home folder. You must save the `.java` file in the `src\example` folder because this MIDlet is in the `example` package.
2. Switch back to the WTK main window and click **Build**. The WTK compiles and pre-verifies the `HelloWorldMIDlet` class, which is then ready to be executed.
3. Click on **Run**. You are presented with a list of MIDlets from the packaged suite, which are ready to be run. As our test suite only has one MIDlet, click on its name and it is executed on the mobile phone emulator (see Figure 2.4).

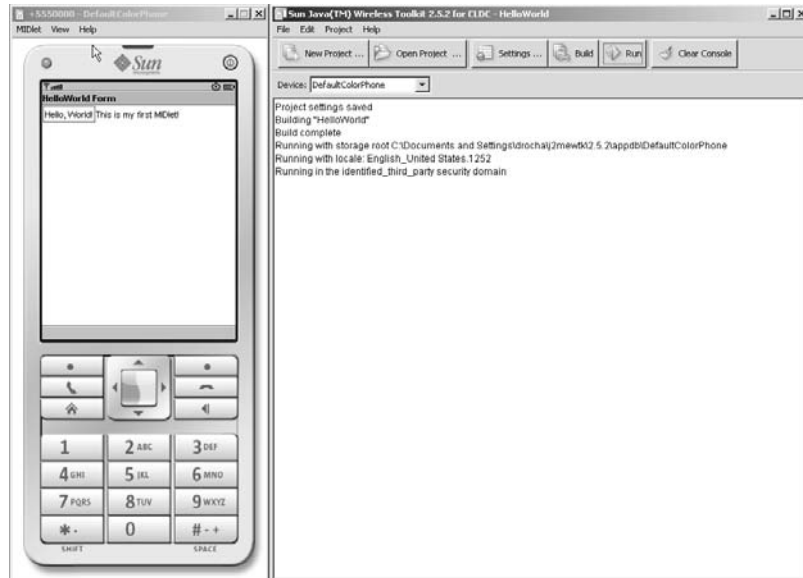


Figure 2.4 Running the HelloWorld MIDlet

2.2.4 Packaging a MIDlet

Following the above steps is sufficient to build simple MIDlets for running in the emulator. However, if you are developing a more sophisticated MIDlet that contains many classes, images, application parameters, and so on, you need to package your MIDlet into a MIDlet suite.

Packaging creates a JAR file containing all your class and resource files (such as images and sounds) and the application descriptor (JAD) file, which notifies the AMS of the contents of the JAR file.

The following attributes must be included in a JAD file:

- **MIDlet-Name:** the name of the suite that identifies the MIDlets to the user
- **MIDlet-Version:** the version number of the MIDlet suite; this is used by the AMS to identify whether this version of the MIDlet suite is already installed or whether it is an upgrade, and communicate this information to the user
- **MIDlet-Vendor:** the organization that provides the MIDlet suite
- **MIDlet-Jar-URL:** the URL from which the JAR file can be loaded, as an absolute or relative URL; the context for relative URLs is the place from where the JAD file was loaded
- **MIDlet-Jar-Size:** the number of bytes in the JAR file.

The following attributes are optional but are often useful and should be included:

- `MIDlet-n`: the name, icon and class of the *n*th MIDlet in the JAR file (separated by commas); the lowest value of *n* must be 1 and all following values must be consecutive; the name is used to identify the MIDlet to the user and must not be null; the icon refers to a PNG file in the resource directory and may be omitted; the class parameter is the name of the class extending the MIDlet class
- `MIDlet-Description`: a description of the MIDlet suite
- `MicroEdition-Configuration`: the Java ME configuration required, in the same format as the `microedition.configuration` system property, for example, 'CLDC-1.0'
- `MicroEdition-Profile`: the Java ME profiles required, in the same format as the `microedition.profiles` system property, that is 'MIDP-1.0' or 'MIDP-2.0'; if the value of the attribute is 'MIDP-2.0', the target device must implement the MIDP profile otherwise the installation will fail; MIDlets compiled against MIDP 1.0 will install successfully on a device implementing MIDP 2.0.

The following is the JAD file for our HelloWorld application:

```
MIDlet-1: HelloWorldMIDlet,,example.HelloWorldMIDlet
MIDlet-Description: Example MIDP MIDlet
MIDlet-Jar-Size: 1042
MIDlet-Jar-URL: HelloWorld.jar
MIDlet-Name: HelloWorld Midlet Suite
MIDlet-Vendor: Midlet Suite Vendor
MIDlet-Version: 1.0.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
```

You can automate the task of creating the JAD file by clicking Package/Create Package in the WTK menu. The WTK creates `HelloWorld.jar` and `HelloWorld.jad` in the `bin` folder of your project root. You can use Windows Explorer to browse to that folder and check that the package files have been correctly created. You can open the JAD file with a standard text editor to inspect its contents. Double-clicking the JAD file executes your application in the emulator (see Figure 2.4).

2.3 MIDP Graphical User Interfaces API

In this section, we take a look at the main APIs provided by the MIDP profile for GUI application development. Designing a user interface

for Java ME applications on mobile devices is quite a challenging task, because MIDP's flexibility allows it to be used in hundreds of different device models, with different form factors, hardware, screen sizes and input methods. Such wide availability makes Java ME applications attractive to a range of users (enterprises, gamers, and casual users), all of whom need to be considered when creating effective UIs for MIDlets.

There are many differences (see Table 2.1) between the hardware and software environments in which Java originated (desktop computers) and the ones found in mobile devices, such as Symbian smartphones.

Table 2.1 Differences between Java environments

Environment	Windows, Linux and Mac OS Personal Computers	Typical Symbian smartphone
Screen	1024 × 768 or bigger Landscape orientation	240 × 320 352 × 416 800 × 352 176 × 208 Landscape or portrait orientation
Input method	QWERTY keyboard Mouse	Numerical keypad QWERTY keyboard Touch screen (with stylus) Touch screen (with fingers)
Display colors	16 bit and higher, usually 32 or 64 bit	16 bit (top phones)
Processing power and memory	1 GHZ+/512 MB RAM +	Around 350 MHz

These differences (and the many others that exist) make it inappropriate to port Swing or AWT toolkits directly to mobile devices. They would suffer from poor performance (due to slower processors and smaller memory), usability problems (there cannot be multiple windows on a mobile phone) and poor input mechanism compatibility, as this varies greatly among devices.

2.3.1 LCDUI Model for User Interfaces

The LCDUI toolkit is a set of features for the implementation of user interfaces especially for MIDP-based devices. LCDUI is a generic set of UI components split into two categories, high-level and low-level.

The high-level UI components' main characteristics are a consistent API and leaving the actual UI layout and appearance to be implemented by the device environment itself. Developers can write their code against a single set of classes and objects and trust they will appear consistently in all devices implementing that set. This approach ensures that high-level UI components are portable across devices, appearing on the screen in a manner that is consistent with the device's form factor, screen size and input methods. The downside is that little control is left in the developers' hands, and some important things, such as fonts, colors and component positioning, can only be hinted at by the developer – the implementation is free to follow those hints or not.

The low-level set of UI components consists of the `Canvas` class, the `GameCanvas` subclass and the associated `Graphics` class. They provide fine-grained, pixel-by-pixel control of layout, colors, component placement, etc. The downside of using the low-level set is that the developer must implement basic UI controls (dialogs, text input fields, forms), since the `Canvas` is simply a blank canvas which can be drawn upon. The `CustomItem` class of the `javax.microedition.lcdui` package can be thought of as belonging to the low-level UI set, as it provides only basic drawing functionalities, allowing developers to specify layout and positioning of controls very precisely. However, as custom items are only used within `Forms` they are discussed in Section 2.3.2.

The Event Model

The `javax.microedition.lcdui` package implements an event model that runs across both the high- and low-level APIs. It handles such things as user interaction and calls to redraw the display. The implementation is notified of such an event and responds by making a corresponding call back to the `MIDlet`. There are four types of UI event:

- events that represent abstract commands that are part of the high-level API; the `Back`, `Select`, `Exit`, `Cancel` commands seen in Symbian OS devices generally fit this category
- low-level events that represent single key presses or releases or pointer events
- calls to the `paint()` method of the `Canvas` class
- calls to an object's `run()` method.

Callbacks are serialized and never occur in parallel. More specifically, a new callback never starts while another is running; this is true even when there is a series of events to be processed. In this case, the callbacks are processed as soon as possible after the last UI callback has returned. The implementation also guarantees that a call to `run()`, requested by

a call to `callSerially()`, is made after any pending `repaint()` requests have been satisfied. There is, however, one exception to this rule: when the `Canvas.serviceRepaints()` method is called by the MIDlet, it causes the `Canvas.paint()` method to be invoked by the implementation and then waits for it to complete. This occurs whenever the `serviceRepaints()` method is called, regardless of where the method was called from, even if that source was an event callback itself.

The Command Class

Abstract commands are used to avoid having to implement concrete command buttons; semantic representations are used instead. The commands are attached to displayable objects, such as high-level `List` or `Form` objects or low-level `Canvas` objects. The `addCommand()` method attaches a command to the displayable object. The command specifies the label, type and priority. The `CommandListener` interface then implements the actual semantics of the command. The native style of the device may prioritize where certain commands appear on the UI. For example, Exit is always placed above the right softkey on Nokia devices. There are also some device-provided operations that help contribute towards the operation of the high-level API. For example, screen objects, such as `List` and `ChoiceGroup`, have built-in events that return user input to the application for processing.

2.3.2 LCDUI High-Level API: Screen Objects

`Alert`, `List`, `TextBox`, and `Form` objects are all derived from `Screen`, itself derived from `Displayable`. Screen objects are high-level UI components that can be displayed. They provide a complete user interface, of which the specific look and feel is determined by the implementation. Only one `Screen`-derived object can be displayed at a time. Developers can control which `Screen` is displayed by using the `setCurrent()` method of the `Display` class.

This section describes the high-level API classes in a succinct manner, rather than going into every detail. To find a complete description of each featured class, please check the MIDP documentation.

Alert Object

An `Alert` object shows a message to the user, waits for a certain period and then disappears, at which point the next displayable object is shown. An `Alert` object is a way of informing the user of any errors or exceptional conditions. It may be used by the developer to inform the user that:

- an error or other condition has been reached

- a user action has been completed successfully
- an event for which the user has previously requested notification has been completed.

To emphasize these states to the user, the `AlertType` can be set to convey the context or importance of the message. For each use case described above, there's a relevant type, such as `ALARM`, `CONFIRMATION`, `ERROR` and `INFO`. These are differentiated by titles at the top of the alert screen, icons drawn on the alert, and the sound played when it is displayed.

List Object

A `List` object is a screen object that contains a list of choices for the user and is, therefore, ideal for implementing choice-based menus, which are the core user interface of most mobile devices.

TextBox Object

A `TextBox` is a `Screen` object that allows the user to enter and edit text in a separate space away from the form. It is a `Displayable` object and can be displayed on the screen in its own right. Its maximum size can be set at creation, but the number of characters displayed at any one time is unrelated to this size and is determined by the device itself.

Form Object

A `Form` object is designed to contain a small number of closely-related user interface elements. Those elements are, in general, subclasses of the `Item` class and we shall investigate them in more detail below. The `Form` object manages the traversal, scrolling and layout of the items.

Items enclosed within a form may be edited using the `append()`, `delete()`, `insert()` and `set()` methods. They are referred to by their indexes, starting at zero and ending with `size()-1`. Items are organized via a layout policy that is based around rows. The rows typically relate to the width of the screen and are constant throughout. Forms grow vertically and a scroll bar is introduced as required. If a form becomes too large, it may be better for the developer to create another screen. Users can interact with a `Form` and a `CommandListener` can be attached to capture this input using the `setCommandListener()` method. An individual `Item` can be given an `ItemCommandListener`, if a more contextual approach is required by the UI.

Item Class

This is the superclass for all items that can be added to a `Form`. Every `Item` has a label, which is a string. This label is displayed by the implementation as near as possible to the `Item`, either on the same horizontal row or above. When an `Item` is created, by default it is not owned by any container and does not have a `Command` or `ItemCommandListener`. However, default commands can be attached to an `Item`, using the `setDefaultCommand()` method, which makes the user interface more intuitive for the user. A user can then use a standard gesture, such as pressing a dedicated selection key or tapping on the item with a pointer. Symbian devices support these interfaces through `S60` and `UIQ`, respectively.

The following types are derived from `Item`.

ChoiceGroup

A group of selectable objects may be used to capture single or multiple choices in a `Form`. It is a subclass of `Item` and most of its methods are implemented via the `Choice` interface.

A `ChoiceGroup` has similar features to a `List` object, but it's meant to be placed in a `Form`, not used as a standalone `Screen` object. Its type can be either `EXCLUSIVE` (to capture one option from the group) or `MULTIPLE` (to capture many selections). As usual with high-level UI components, developers don't have control over the graphical representation of a `ChoiceGroup`. Usually, though, one of `EXCLUSIVE` type is shown as a list of radio buttons, while the `MULTIPLE` type is rendered as a list of checkboxes.

CustomItem

`CustomItem` operates in a similar way to `Canvas`: the developer can specify precisely what content appears where within its area. Some of the standard items may not give quite the required functionality, so it may be better to define home-made ones instead. The drawback to this approach is that, as well as having to draw all the contents using the item's `paint()` method, the developer has to process and manage all events, such as user input, through `keyPressed()`. Custom items may interact with either keypad- or pointer-based devices. Both are optional within the specification and the underlying implementation will signal to the item which has been implemented.

`CustomItem` also inherits from `Item`, therefore inheriting the `getMinContentWidth()` and `getPrefContentHeight()` methods, which help the implementation to determine the best fit of items within

the screen layout. If the `CustomItem` is too large for the screen dimensions, it resizes itself to within those preferred, minimum dimensions and the `CustomItem` is notified via `sizeChanged()` and `paint()` methods.

Additionally, the developer can use the `Display.getColor(int)` and `Font.getFont(int)` methods to determine the underlying properties for items already displayed in the form of which the `CustomItem` is a part, to ensure that a consistent appearance is maintained.

DateField

This is an editable component that may be placed upon a `Form` to capture and display date and time (calendar) values. The item can be added to the form with or without an initial value. If the value is not set, a call to the `getDate()` method returns `NULL`. The field can handle `DATE` values, `TIME` values, and `DATE-TIME` values.

ImageItem

The `ImageItem` is a reference to a mutable or immutable image that can be displayed on a `Form`. We look at the `Image` object in detail in Section 2.3.4. Suffice to say that the `Image` is retrieved from the MIDlet suite's JAR file in order to be displayed upon the form. This is performed by calling the following method, in this case from the root directory:

```
Image image = Image.createImage("/myImage.png");
```

An `ImageItem` can be rendered in various modes, such as `PLAIN`, `HYPERLINK`, or `BUTTON`. Please check the MIDP documentation for more details.

Gauge

This component provides a visual representation of an integer value, usually formatted as a bar graph. Gauges are used either for specifying a value between zero and a maximum value, or as a progress or activity monitor.

Spacer

This blank, non-interactive item with a definable minimum size is used for allocating flexible amounts of space between items on a form and gives the developer much more control over the appearance of a form. The minimum width and height for each spacer can be defined to provide space between items within a row or between rows of items on the form.

StringItem

This is a display-only item that can contain a string and the user cannot edit the contents. Both the label and content of the `StringItem` can, however, be changed by the application. As with `ImageItem`, its appearance can be specified at creation as one of `PLAIN`, `HYPERLINK` or `BUTTON`. The developer is able to set the text, using the `setText()` method, and its appearance, using `setFont()`.

TextField

A `TextField` is an editable text component that may be placed in a `Form`. It can be given an initial piece of text to display. It has a maximum size, set by `setSize(int size)`, and an input mask, which can be set when the item is constructed. An input mask is used to ensure that end users enter the correct data, which can reduce user frustration. The following masks can be used: `ANY`, `EMAILADDR`, `NUMERIC`, `PHONENUMBER`, `URL`, and `DECIMAL`. These constraints can be set using the `setConstraints()` method and retrieved using `getConstraints()`. The constraint settings should be used in conjunction with the following set of modifier flags using the bit-wise AND (&) operator: `PASSWORD`, `SENSITIVE`, `UNEDITABLE`, `NON_PREDICTIVE`, `INITIAL_CAPS_WORD`, `INITIAL_CAPS_SENTENCE`.

Ticker

This implements a ticker-tape object – a piece of text that runs continuously across the display. The direction and speed of the text is determined by the device. The ticker scrolls continuously and there is no interface to stop and start it. The implementation may pause it when there has been a period of inactivity on the device, in which case the ticker resumes when the user recommences interaction with the device.

2.3.3 LCDUI Interfaces

The user interface package, `javax.microedition.lcdui`, provides four interfaces that are available to both the high- and low-level APIs:

- `Choice` defines an API for user-interface components, such as `List` and `ChoiceGroup`. The contents of these components are represented by strings and images which provide a defined number of choices for the user. The user's input can be one or more choices and they are returned to the application upon selection.
- `CommandListener` is used by applications that need to receive high-level events from the implementation; the listener is attached to

a displayable object within the application using the `addCommand()` method.

- `ItemCommandListener` is a listener type for receiving notification of commands that have been invoked on `Item` objects. This provides the mechanism for associating commands with specific `Form` items, thus contextualizing user input and actions according to the current active item on the form, making it more intuitive.
- `ItemStateListener` is used by applications that need to receive events that indicate changes in the internal state of the interactive items within a form; for example, a notification is sent to the application when the set of selected values within a `ChoiceGroup` changes.

2.3.4 LCDUI Low-Level API: Canvas

The low-level API allows developers to have total control of how the user interface looks and how components are rendered on the screen. `Canvas`, the main base class for low-level UI programming, is used to exercise such fine-grained control. An application should subclass `Canvas` to create a new displayable screen object. As it is displayable, it can be used as the current display for an application just like the high-level components. Therefore a MIDlet application can have a user interface with, for example, `List`, `Form` and `Canvas` objects, which can be displayed one at a time to provide the application functionality to the users.

`Canvas` is commonly used by game developers when creating sprite animation and it also forms the basis of `GameCanvas`, which is part of the Game API (see Section 2.3.6). `Canvas` can be used in normal mode, which allows title and softkey labels to be displayed, and full-screen mode, where `Canvas` takes up as much of the display as the implementation allows. In either mode, the dimensions of the `Canvas` can be accessed using the `getWidth()` and `getHeight()` methods.

Graphics are drawn to the screen by implementing code in the abstract `paint()` method. This method must be present in the subclass and is called as part of the event model. The event model provides a series of user-input methods such as `keyPressed()` and `pointerPressed()`, depending upon the device's data input implementation. The `paint(Graphics g)` method passes in a `Graphics` object, which is used to draw to the display.

The `Graphics` object provides a simple 2D, geometric-rendering capability, which can be used to draw strings, characters, images, shapes, etc. For more details, please check the MIDP documentation.

Such methods as `keyPressed()` and `pointerPressed()` represent the interface methods for the `CommandListener`. When a key is pressed, it returns a key code to the command listener. These key codes are

mapped to keys on the keypad. The key code values are unique for each hardware key, unless keys are obvious synonyms for one another. These codes are equal to the Unicode encoding for the character representing the key. Examples of these are `KEY_NUM0`, `KEY_NUM1`, `KEY_STAR`, and `KEY_POUND`. The problem with these key codes is that they are not necessarily portable across devices: other keys may be present on the keypad and may form a distinct list from those described previously. It is therefore better, and more portable, to use game actions instead. Each key code can be mapped to a game action using the `getGameAction(int keyCode)` method. This translates the key code into constants such as `LEFT`, `RIGHT`, `FIRE`, `GAME_A` and `GAME_B`. Codes can be translated back to key codes by using `getKeyCode(int gameAction)`. Apart from making the application portable across devices, these game actions are mapped in such a way as to suit gamers. For example, the `LEFT`, `RIGHT`, `UP` and `DOWN` game actions might be mapped to the 4, 6, 2 and 8 keys on the keypad, making game-play instantly intuitive.

A simple Canvas class might look like this:

```
import javax.microedition.lcdui.*;

public class SimpleCanvas extends Canvas {
    public void paint(Graphics g) {
        // set color context to be black
        g.setColor(255, 255, 255);
        // draw a black filled rectangle
        g.fillRect(0, 0, getWidth(), getHeight());
        // set color context to be white
        g.setColor(0, 0, 0);
        // draw a string in the top left corner of the display
        g.drawString("This is some white text", 0, 0, g.TOP | g.LEFT);
    }
}
```

2.3.5 Putting It All Together: `UIExampleMIDlet`

We use our new knowledge of the high- and low-level APIs to build a showcase of most LCDUI components: `Form`, `Item` (`ImageItem`, `StringItem`, `ChoiceGroup`, `DateField` and `Gauge`), `Ticker`, `List`, `TextBox`, and `Canvas`. We also use the `Command` class and its notification interface, `CommandListener`, to show how you can handle these abstract events to produce concrete behavior in your application.

The application also shows how to build a menu using `Commands` and how to create high- and low-level UI components and switch between them, emulating what we would have in a real-world application. For the sake of simplicity and readability, we skip some code sections which are common knowledge among Java programmers, such as the list of imported packages and empty method implementations. This allows us

to focus instead on the behavior of the UI components we have described in so much detail.

To get started, open the WTK and create a new project, called `UIExampleMIDlet`. Our application has two classes in the `example` package: `UIExampleMIDlet` (the main class) and `MenuCanvas`.

UIExampleMIDlet Class

```

Class UIExampleMIDlet
package example;
/*
Import list omitted for readability
*/
public class UIExampleMIDlet extends MIDlet implements CommandListener {
    private Form form = null;
    private List list = null;
    private TextBox textBox = null;
    private Canvas canvas = null;
    private TextField textField = null;
    private DateField dateField = null;
    private Alert alert = null;
    private ChoiceGroup choiceGroup = null;
    private StringItem stringItem = null;
    private Image image = null;
    private ImageItem imageItem = null;
    private Command backCommand = null;
    private Command exitCommand = null;
    private Ticker ticker = null;
    private Gauge gauge = null;

    public UIExampleMIDlet() throws IOException {
        // creating commands
        backCommand = new Command("Back", Command.BACK, 0);
        exitCommand = new Command("Exit", Command.EXIT, 0);

        // creating Form and its items
        form = new Form("My Form");
        textField = new TextField("MyTextField", "", 20, TextField.ANY);
        dateField = new DateField("MyDateField", DateField.DATE_TIME);
        stringItem = new StringItem("MyStringItem", "My value");
        choiceGroup = new ChoiceGroup("MyChoiceGroup", Choice.MULTIPLE,
            new String[] { "Value 1", "Value 2", "Value 3"}, null);
        image = Image.createImage("/test.png");
        imageItem = new ImageItem("MyImage", image, ImageItem.LAYOUT_CENTER,
            "No Images");
        gauge = new Gauge("My Gauge", true, 100, 1);

        form.append(textField);
        form.append(dateField);
        form.append(stringItem);
        form.append(imageItem);
        form.append(choiceGroup);
        form.append(gauge);
        form.addCommand(backCommand);
        form.setCommandListener(this);
    }
}

```

```

// creating List
list = new List("MyList",Choice.IMPLICIT,
               new String[] {"List item 1","List item 2",
                             "List item 3"},null);

list.addCommand(backCommand);
list.setCommandListener(this);

// creating textBox
textBox = new TextBox("MyTextBox","",256,TextField.ANY);
textBox.addCommand(backCommand);
textBox.setCommandListener(this);

// creating main canvas
canvas = new MenuCanvas();
canvas.addCommand(exitCommand);
canvas.setCommandListener(this);
// creating ticker
ticker = new Ticker("My ticker is running....");
canvas.setTicker(ticker);

// creating alert
alert = new Alert("Message","This is a message!",null,AlertType.INFO);
}

public void commandAction(Command c, Displayable d) {
    if(c == exitCommand) {
        notifyDestroyed();
    }

    if(c == backCommand) {
        setDisplayable(canvas);
    }

    if(d == canvas) {
        if(c.getLabel().equals("Form")) {
            setDisplayable(form);
        }
        else if(c.getLabel().equals("List")) {
            setDisplayable(list);
        }
        else if(c.getLabel().equals("TextBox")) {
            setDisplayable(textBox);
        }
        else if(c.getLabel().equals("Canvas")) {
            setDisplayable(canvas);
        }
        else if(c.getLabel().equals("Alert")) {
            setDisplayable(alert);
        }
    }
}

private void setDisplayable(Displayable d) {
    Display.getDisplay(this).setCurrent(d);
}

protected void startApp() throws MIDletStateChangeException {
    setDisplayable(canvas);
}

```



```

protected void destroyApp(boolean unconditional)
    throws MIDletStateChangeException {
    // empty implementation, as there are no resources to be released
}
protected void pauseApp() {
    // empty implementation, as there are no resources to be released
}
}

```

All of our instance variables, references to the UI components in our application, are declared and initialized to NULL. In the class constructor, we assign each variable to a newly-created object of the proper class. First, we create a couple of commands (Back and Exit) that will be used to handle the high-level user actions defined by the device implementation, assigned to the UI components and passed to us via the `commandAction(Command, Displayable)` method:

```

backCommand = new Command("Back", Command.BACK, 0);
exitCommand = new Command("Exit", Command.EXIT, 0);

```

Following this, we begin to create the actual UI components. For example, let's see how a new Form object is constructed:

```

form = new Form("My Form");
textField = new TextField("MyTextField", "", 20, TextField.ANY);
dateField = new DateField("MyDateField", DateField.DATE_TIME);
stringItem = new StringItem("MyStringItem", "My value");
choiceGroup = new ChoiceGroup("MyChoiceGroup", Choice.MULTIPLE,
    new String[] {"Value 1", "Value 2", "Value 3"}, null);
image = Image.createImage("/test.png");
imageItem = new ImageItem("MyImage", image, ImageItem.LAYOUT_CENTER,
    "No Images");
gauge = new Gauge("My Gauge", true, 100, 1);

form.append(textField);
form.append(dateField);
form.append(stringItem);
form.append(imageItem);
form.append(choiceGroup);
form.append(gauge);
form.addCommand(backCommand);
form.setCommandListener(this);

```

As outlined before, a Form is a flexible UI component that can hold other components, instances of the Item class, and show them on the screen. Users expect Form components to present information that is related in meaning, such as settings for an application. Doing otherwise can be confusing and result in bad user interaction with the application.

After instantiating a Form, we create its items (textField, dateField, stringItem, choiceGroup, imageItem and Gauge) and use

the `Form.append()` method to add them to the `Form` components list. They are then displayed with the `Form`. One important detail: the `Image` object we use to create the `ImageItem` retrieves a PNG image from the root of the JAR file. In order to replicate this behavior in your own project, just copy a file called `test.png` to the `res` folder of the project.

After creating and appending the items, we add the `Back` command to the `Form`; the emulator's implementation maps it to one of the softkeys. Finally, we set the `CommandListener` to be our own `MIDlet`, which implements the `commandAction()` method (defined in the `CommandListener` interface) to handle user actions.

The rest of the constructor creates the other UI components, adds the `Back` command to them and sets the `MIDlet` itself as their event listener. The following lines create an instance of `MenuCanvas` and add the `Exit` command to it:

```
canvas = new MenuCanvas();
canvas.addCommand(exitCommand);
canvas.setCommandListener(this);
```

This canvas is the main screen for our application.

The `commandAction()` method reacts to the events of the `Back` and `Exit` buttons and the `MenuCanvas`, changing screens according to the user's choices. The `startApp()` method simply sets the `MenuCanvas` instance as the main display, using the `setDisplayable()` utility.

MenuCanvas Class

```
package example;
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Graphics;
public class MenuCanvas extends Canvas {
    private Command[] options = new Command[] {
        new Command("Form",Command.OK,1),
        new Command("List",Command.OK,2),
        new Command("Canvas",Command.OK,3),
        new Command("TextBox",Command.OK,4),
        new Command("Alert",Command.OK,4)
    };

    public MenuCanvas() {
        for(int i=0;i<options.length;i++) {
            this.addCommand(options[i]);
        }
    }

    protected void paint(Graphics g) {
        g.setColor(0,0,0);
        g.fillRect(0,0,getWidth(),getHeight());
    }
}
```

```

g.setColor(255,255,255);
g.drawString("This is a canvas.",0,0,Graphics.TOP | Graphics.LEFT);
g.drawString("Check Options menu",0,20, Graphics.TOP | Graphics.LEFT);
g.drawString("for more UI components",0,40,
            Graphics.TOP | Graphics.LEFT);
}
}

```

The options instance variable holds an array of Commands which serves as the main menu of the application. Each command has a label, a positioning guide (Command.OK) and a priority value. In the constructor, we loop through the array, adding all the commands to the Canvas so they are displayed on the screen. The abstract paint(Graphics) method is implemented so that the Canvas can be drawn upon, using the Graphics object passed in as a parameter. In this case, we set the context color to black, paint a screen-sized rectangle, set the color to white and draw some strings to instruct the user how to use the application.

Save UIExampleMIDlet in a file called UIExampleMIDlet.java and MenuCanvas in a file called MenuCanvas.java in your project's src/example folder, then build and package the project using the WTK. Figure 2.5 shows the application running on the WTK emulator.



Figure 2.5 UIExampleMIDlet application running on the WTK emulator: a) the MenuCanvas and b) the Form

2.3.6 Game API

The MIDP specification supports easy game development through the use of the `javax.microedition.lcdui.game` package. It contains the following classes:

- `GameCanvas`
- `LayerManager`
- `Layer`
- `Sprite`
- `TiledLayer`.

The aim of the API is to facilitate richer gaming content through a set of APIs that provides useful functionality.

GameCanvas

A basic game user interface class that extends `javax.microedition.lcdui.Canvas`, `GameCanvas` provides an offscreen buffer as part of the implementation even if the underlying device doesn't support double buffering. The `Graphics` object obtained from the `getGraphics()` method is used to draw to the screen buffer. The contents of the screen buffer can then be rendered to the display synchronously by calling the `flushGraphics()` method. The `GameCanvas` class also provides the ability to query key states and return an integer value in which each bit represents the state of a specific key on the device:

```
public int getKeyStates();
```

If the bit representing a key is set to 1, then this key has been pressed at least once since the last invocation of the method. The returned integer can be ANDed against a set of predefined constants, each representing a specific key, by having the appropriate bit set (support for the last four values is optional).

We use the `GameCanvas` class members that describe key presses (e.g., `GameCanvas.FIRE_PRESSED`) to ascertain the state of a key in the manner shown below:

```
if ( getKeyStates() & Game.Canvas.FIRE_PRESSED != 0 ) {  
    // FIRE key is down or has been pressed - take appropriate action  
}
```

TiledLayer

The abstract `Layer` class is the parent class of `TiledLayer` and `Sprite`. A `TiledLayer` consists of a grid of cells each of which can be filled with an image tile. An instance of `TiledLayer` is created by invoking the constructor:

```
public TiledLayer(int columns, int rows, Image image, int tileWidth,  
                  int tileHeight);
```

The `columns` and `rows` arguments represent the number of columns and rows in the grid. The `tileWidth` and `tileHeight` arguments represent the width and height of a single tile in pixels. The `image` argument represents the image used to create the set of tiles that populate the `TiledLayer`. Naturally, the dimension of the image in pixels must be an integral multiple of the dimension of an individual tile. The use of `TiledLayer` is best illustrated with an example.

One of the principal uses of `TiledLayer` is the creation of large scrolling backgrounds from relatively few tiles. Consider the example in Figure 2.6, which shows a big set of tiles of equal dimensions that we can arrange in different ways to create a `TiledLayer` background using selected tiles. You can for example reuse tile 1 to create a large lake or tiles 1, 5 and 16 to create a strip of land surrounded by a small

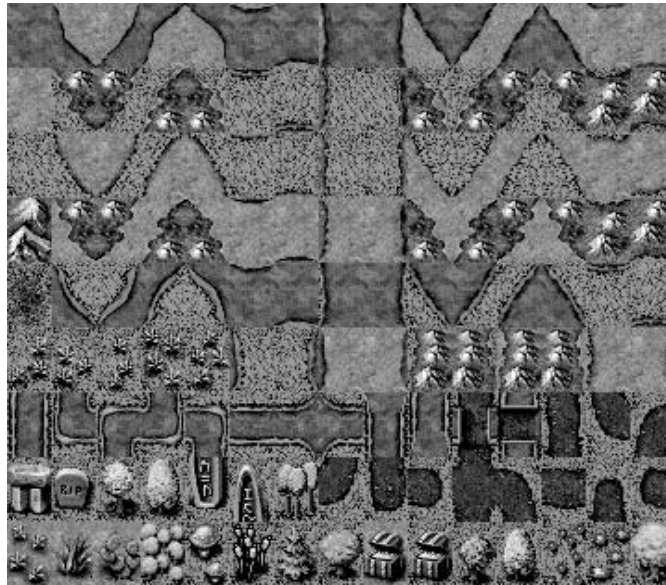


Figure 2.6 Image for use by a `TiledLayer`



Figure 2.7 Image split into 135 tiles

lake. The dimensions of the image are 360×315 pixels and each tile's dimension is 24×35 pixels; that gives us a total of 135 tiles which can be treated as a 15×9 array (see Figure 2.7) and manipulated to create our background.

We put `TiledLayer` scrolling backgrounds into use later on, with a `LayerManagerDemo` example.

Sprite

A `Sprite` is a basic visual element suitable for creating animations and consists of an image composed of several smaller images (frames). The `Sprite` can be rendered as one of the frames. By rendering different frames in a sequence, a `Sprite` provides animation. Let us consider a simple example. Figure 2.8 consists of 12 frames of the same width and height. By displaying some of the frames in a sequence, we can produce an animation.

Here's how to create and animate a sprite based on the image in Figure 2.8. The code is abbreviated for clarity and we will see a more complete example in the `LayerManagerDemo` MIDlet:

```
// create image for sprite
Image image = Image.createImage("/example_sprite.png");
// create and position sprite
guy = new Sprite(image, SPRITE_WIDTH, SPRITE_HEIGHT);
guy.setPosition(spritePositionX, spritePositionY);
// paint sprite on the screen
public void paint(Graphics g) {
    g.setColor(255, 255, 255);
    g.fillRect(0, 0, getWidth(), getHeight());
    guy.paint(g);
}
// loop through all the frames
while (running) {
    repaint();
    guy.nextFrame();
    ...
}
```



Figure 2.8 A Sprite image consisting of 12 frames

In addition to various transformations such as rotation and mirroring, the `Sprite` class also provides collision detection, which is essential for games as it allows the developer to detect when the sprite collides with another element and prevent the hero from crossing a solid wall or obstacle. Both pixel-level and bounding-rectangle collisions are available.

LayerManager

As the name implies, the `LayerManager` manages a series of `Layer` objects. `Sprite` and `TiledLayer` both extend `Layer`. More specifically, a `LayerManager` controls the rendering of `Layer` objects. It maintains an ordered list so that they are rendered according to their z-values (in standard computer graphics terminology). We add a `Layer` to the list using the method:

```
public void append(Layer l);
```

The first layer appended has index zero and the lowest z-value – that is, it appears closest to the user (viewer). Subsequent layers have successively

greater z-values and indices. Alternatively, we can add a layer at a specific index using the method:

```
public void insert(Layer l, int index);
```

To remove a layer from the list we use the method:

```
public void remove(Layer l);
```

We position a layer in the `LayerManager`'s coordinate system using the `setPosition()` method. The contents of `LayerManager` are not rendered in their entirety; instead, a view window is rendered using the `paint()` method of the `LayerManager`:

```
public void paint(Graphics g, int x, int y);
```

The `x` and `y` arguments are used to position the view window on the displayable object (`Canvas` or `GameCanvas`) upon which the `LayerManager` is rendered. The size of the view window is set using this method:

```
public void setViewWindow(int x, int y, int width, int height);
```

The `x` and `y` values determine the position of the top left corner of the rectangular view window in the coordinate system of the `LayerManager`. The `width` and `height` arguments determine the width and height of the view window and are usually set to a size appropriate for the device's screen. By varying the `x` and `y` coordinates we can pan through the contents of the `LayerManager`.

LayerManagerDemo Example

Our `LayerManagerDemo` example summarizes all Game API concepts seen so far. It illustrates the use of a `Sprite` (for animating the hero), a `TiledLayer` (for background construction), a `LayerManager` (for scrolling the background), and a `GameCanvas` (for drawing it all). The source code is too large to be included in this book, so it is available for download from the website. We highlight here some parts that show how to use the Game API components.

In the constructor, we load the image resources used by our `TiledLayer` and the `Sprite` (see Figures 2.6 and 2.8, respectively). We create our game hero, using the `Sprite` constructor, then set the frame sequence, which allows us to loop only to the frames that interest us. In

this case, we want the hero to walk downwards, so we choose to use only frames 6, 7 and 8:

```
guy = new Sprite(guyImage, SPRITE_WIDTH, SPRITE_HEIGHT);  
guy.setFrameSequence(new int[] {6, 7, 8});
```

We then create the background for the game scene, by constructing a new `TiledLayer` from the source image (see Figure 2.9), and use the `fillLayer()` method and the cells array to create a terrain over which our hero will walk.

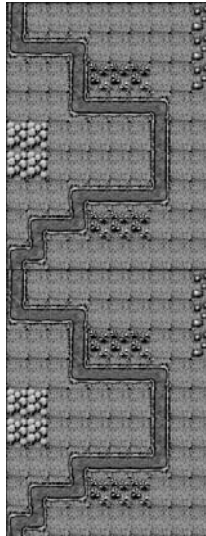


Figure 2.9 Background image for the `LayerManagerDemo` MIDlet

We create a `LayerManager` to manage both the background `TiledLayer` and the `Sprite` (both inherit from `Layer`). We append the `Sprite` then the background layer, to ensure that the hero is shown over the background and not the other way around, since the hero has a smaller z-index. This is how it's done in code:

```
// creating LayerManager  
manager = new LayerManager();  
manager.append(guy);  
manager.append(background);  
manager.setViewWindow(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
```

We also set the view window to match the size of the screen so we can see it in its entirety, and not just a portion of it. Try playing with the values for the view window and you get partial views of the background

that you can use for other games ideas. The last line of code in the constructor just sets the `GameCanvas` to be in full-screen mode.

Since your animation runs permanently, it's good practice to put it in another thread, so we don't block the current thread, which serves UI application requests. This frees the application to continue processing events. To do this, we make our `GameCanvas` `Runnable`, implement the game animation loop within the `run()` method, and use the `start()` and `stop()` methods to start and stop the animation:

```
public void start() {
    running = true;
    Thread t = new Thread(this);
    t.start();
}

public void stop() {
    running = false;
}
```

Using `t.start()` triggers the `run()` method, which is then responsible for the game animation. This includes managing background scrolling, animating the sprite, drawing everything to the screen, and resetting the background scrolling when needed, so our hero won't fall off the screen:

```
public void run() {
    Graphics graphics = this.getGraphics();
    // graphics.setColor(255, 255, 255);

    while(running) {
        layerY -= 1;
        background.setPosition(0, layerY);
        guy.setPosition(getWidth()/2 - guy.getWidth()/2,
                        getHeight()/2 - guy.getHeight()/2);
        manager.paint(graphics, 0, 0);
        flushGraphics();
        guy.nextFrame();

        try {
            Thread.sleep(20);
        }
        catch(InterruptedException e) {
            break;
        }

        if(layerY == -(background.getHeight() / 2)) {
            layerY = 0;
        }
    }
}
```

The source code, JAR and JAD files for the `LayerManagerDemo` MIDlet are available from the book's website.¹ I strongly encourage you

¹ developer.symbian.com/javameonsymbianos

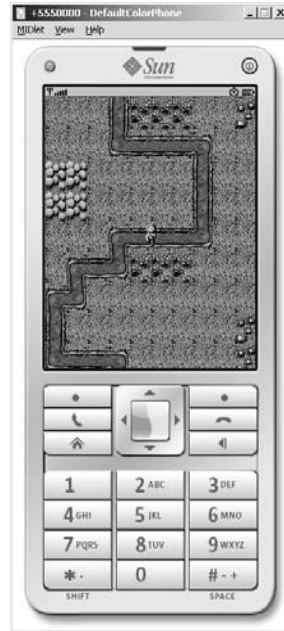


Figure 2.10 LayerManagerDemo MIDlet

to at least run the example application with the WTK emulator (see Figure 2.10) so you can see for yourself how these techniques combine to generate the illusion of motion that is the basis of all games and get motivated to create your own games using this skeleton code.

This concludes our introduction to the Game API of MIDP 2.0. We come back to this subject in much more detail in Chapter 8, where you develop a full, multimedia-rich game application expanded to allow multiple player, interconnected gaming.

2.4 Non-GUI APIs in MIDP

In this section, we cover in more detail the MIDP APIs not related to GUI development. Focus here is given to the persistent data, Media, Networking and Push Registry APIs.

2.4.1 Record Management System

MIDP provides a simple record-based persistent storage mechanism known as the Record Management System (RMS). It's a package that

allows the MIDlet application to store persistent data within a controlled environment, while maintaining system security. It provides a simple, non-volatile data store for MIDlets while they are not running. The classes making up the RMS are contained in the `javax.microedition.rms` package.

Essentially, the RMS is a very small, basic database. It stores binary data in a `RecordStore` within a `RecordStore`. MIDlets can add, remove and update the records in a `RecordStore`. The persistent data storage location is implementation-dependent and is not exposed to the MIDlet. A `RecordStore` is, by default, accessible across all MIDlets within a suite, and MIDP extends access to MIDlets with the correct access permissions from other MIDlet suites. When the parent MIDlet suite is removed from the device, its record stores are also removed, regardless of whether a MIDlet in another suite is making use of them.

Here is a short example of how to use RMS for writing and reading persistent data from the device:

```
// saving data to RMS
public int saveToStore(byte[] data) {
    int recordID = 0;
    try {
        RecordStore store = RecordStore.openRecordStore("ImageStore", true);
        recordID = store.addRecord(data, 0, data.length);
        store.closeRecordStore();
    }
    catch(RecordStoreException rse) {
        rse.printStackTrace();
    }
    return recordID;
}

// reading data from RMS
public byte[] loadFromStore(String storeName, int recordID) {
    byte[] data = null;
    try {
        RecordStore store = RecordStore.openRecordStore("ImageStore", false);
        data = store.getRecord(recordID);
        store.closeRecordStore();
    }
    catch(RecordStoreException rse) {
        rse.printStackTrace();
    }
    return data;
}
```

For more about RMS, including searching and sorting records in a `RecordStore`, refer to the MIDP documentation. Also download the `RMSWriter` and `RMSReader` example applications from this book's website; they provide a fairly complete example of how to write, read and share `RecordStores`.

2.4.2 Media API

MIDP includes the Media API which provides limited, audio-only multimedia. It is a subset of the optional and much richer JSR-135 Mobile Media API, which currently ships on most Symbian OS phones.

The MIDP Media API provides support for tone generation and audio playback of WAV files if the latter is supported by the underlying hardware. Since MIDP is targeted at the widest possible range of devices, not just feature-rich smartphones, the aim of the Media API is to provide a lowest common denominator of functionality suitable for the capabilities of all MIDP devices. However, with the wide availability of JSR-135 MMAPI on Symbian OS devices, we look at both the Media and Mobile Media APIs in detail later in this chapter, as the latter is much richer in functionality and provides additional opportunities for development of media-centric Java applications.

2.4.3 Networking – Generic Connection Framework

CLDC has defined a streamlined approach to networking, known as the Generic Connection Framework (GCF). The framework seeks to provide a consistent interface for every network connection between the MIDP classes and the underlying network protocols. It doesn't matter what kind of connection is being opened; the interface remains the same. For instance, if you're opening a socket or an HTTP connection, you are still going to use the same `Connector.open()` method. MIDP has support for many protocols, although HTTP and HTTPS are mandatory. Java ME on Symbian OS also supports other optional protocols, such as sockets, server sockets and datagrams. Due to the implications of the MIDP security model on networking APIs, these are discussed in more detail in Section 2.6.

2.4.4 Push Registry

The Push Registry API allows MIDlets to be launched in response to incoming network connections. Many applications, particularly messaging applications, need to be continuously listening for incoming messages. To achieve this in the past, a Java application would have had to be continually running in the background. Although the listening Java application may itself be small, it would still require an instance of the virtual machine to be running, thus appropriating some of the mobile phone's scarce resources. The JSR-118 group recognized the need for an alternative, more resource-effective solution for MIDP and so introduced the Push Registry. The Push Registry API is also affected by MIDP 2.0's security model, therefore we discuss it in Section 2.7.

2.5 MIDP Security Model

The MIDP security model is built on two concepts:

- Trusted MIDlet suites are those whose origin and integrity can be trusted by the device on the basis of some objective criterion.
- Protected APIs are APIs to which access is restricted, with the level of access being determined by the permissions (Allowed or User) allocated to the API.

A protection domain defines a set of permissions which grant, or potentially grant, access to an associated set of protected APIs. An installed MIDlet suite is bound to a protection domain, thereby determining its access to protected APIs.

An MIDP device must support at least one protection domain, the untrusted domain, and may support several protection domains, although a given MIDlet suite can only be bound to one protection domain. The set of protection domains supported by an implementation defines the security policy. If installed, an unsigned MIDlet suite is always bound to the untrusted domain, in which access to protected APIs may be denied or require explicit user permission. Since a requirement of the MIDP specification is that a MIDlet suite written to the MIDP 1.0 specification runs unaltered in an MIDP environment, MIDP 1.0 MIDlets are automatically treated as untrusted.

2.5.1 The X.509 PKI

The mechanism for identifying and verifying that a signed MIDlet suite should be bound to a trusted domain is not mandated by the MIDP specification but is left to the manufacturer of the device and other stakeholders with an interest in the security of the device, for example, network operators. The specification does, however, define how the X.509 Public Key Infrastructure (PKI) can be used to identify and verify a signed MIDlet suite.

The PKI is a system for managing the creation and distribution of digital certificates. At the heart of the PKI lies the system of public key cryptography. Public key cryptography involves the creation of a key pair consisting of a private key and a public key. The creator of the key pair keeps the private key secret, but can freely distribute the public key. Public and private key pairs have two principal uses: they enable secure communication using cryptography and authentication using digital signatures. In the first case, someone wishing to communicate with the holder of the private key uses the public key to encrypt the communication. The encrypted communication is secure since it can only be decrypted by the holder of the private key.

In the current context, however, we are more interested in the second use of public–private key pairs: enabling authentication using digital signatures. A digital signature is an electronic analogy of a conventional signature. It authenticates the source of the document and verifies that the document has not been tampered with in transit. Signing a document is a two-stage process: a message digest is created that is a unique representation of the contents of the document; the message digest is then encrypted using the private key of the sender (see Figure 2.11).

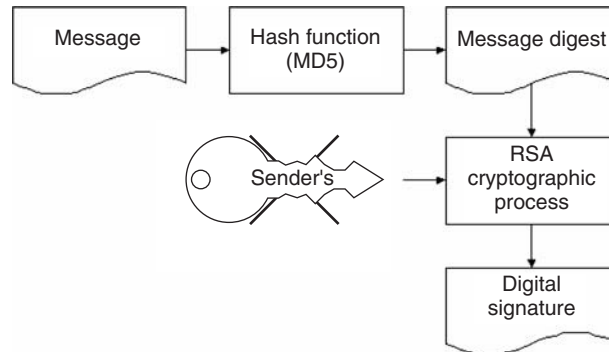


Figure 2.11 Encryption process in a nutshell

The receiver of the document then uses the public key of the sender to decrypt the message digest, creates a digest of the received contents, and checks that it matches the decrypted digest that accompanied the document. Hence, a digital signature is used to verify that a document was sent by the holder of the private key, not some third party masquerading as the sender, and that the contents have not been tampered with in transit. This raises the issue of key management and how the receiver of a public key can verify the source of the public key. For instance, if I receive a digitally signed JAR file, I need the public key of the signer to verify the signature, but how do I verify the source of the public key? The public key itself is just a series of numbers, with no clue as to the identity of the owner. I need to have confidence that a public key purporting to belong to a legitimate organization does in fact originate from that organization and has not been distributed by an impostor, enabling the impostor to masquerade as the legitimate organization, signing files using the private key of a bogus key pair. The solution is to distribute the public key in the form of a certificate from a trusted certificate authority (CA).

A certificate authority distributes a certificate that contains details of a person's or organization's identity, the public key belonging to that person or organization, and the identity of the issuing CA. The CA vouches that the public key contained in the certificate does indeed belong to the person or organization identified on the certificate. To verify that the

certificate was issued by the CA, the certificate is digitally signed by the CA using its private key. The format of certificates used in X509.PKI is known as the X509 format.

Of course, this raises the question of how the recipient of the certificate verifies the digital signature contained therein. This is resolved using root certificates or root keys. The root certificate contains details of the identity of the CA and the public key of the CA (the root key) and is signed by the CA itself (self-signed). For mobile phones that support one or more trusted protection domains, one or more certificates ship with the device, placed on the phone by the manufacturer or embedded in the WIM/SIM card by the network operator. Each certificate is associated with a trusted protection domain, so that a signed MIDlet that is authenticated against a certificate is bound to the protection domain associated with that certificate.

2.5.2 Certification Paths

In practice, the authentication of a signed file using the root certificate may be more complex than the simplified approach described above. The PKI allows for a hierarchy of certificate authorities (see Figure 2.12)

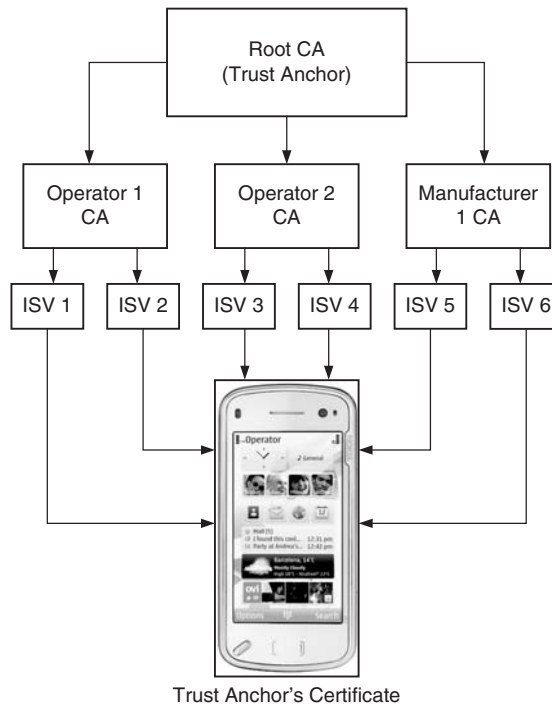


Figure 2.12 Applications from a variety of independent software vendors (ISVs) signed by various CAs and authenticated by a single trust root

whose validity can be traced back to a root certification authority, the uppermost CA in the hierarchy, also known as the trust anchor. In this case the root certificate on the device (the trust root) belongs to the root certification authority in the hierarchy (the trust anchor) which directly or indirectly validates all the other CAs in the certification path. The certificate supplied with the signed JAR file does not need to be validated (signed) by the trust anchor whose certificate is supplied with the device, as long as a valid certification path can be established between the certificate accompanying the signed JAR file and the root CA. It is not actually necessary for a device to have various self-signed top-level certificates from CAs, manufacturers and operators installed. In practice, it only needs access to one or more certificates which are known to be trustworthy, for example, because they are in ROM or secure storage on a WIM/SIM, or because the user has decided that they are.

These certificates act as trust roots. If the authentication of an arbitrary certificate chains back to a trust root known to the device, and the trust root is also identified as being suitable for authenticating certificates being used for a given purpose, for example, code-signing, website identification, and so on, then the arbitrary certificate is considered to have been authenticated.

2.5.3 Signing a MIDlet Suite

To sign a MIDlet suite, a supplier must create a public–private key pair and sign the MIDlet JAR file with the private key. The JAR file is signed using the RSA-SHA1 algorithm. The resulting signature is encoded in Base64 format and inserted into the application descriptor as the following attribute:

```
MIDlet-Jar-RSA-SHA1: <base64 encoding of JAR signature>
```

The supplier must obtain a suitable MIDlet suite code-signing certificate from an appropriate source, for example, the developer program of a device manufacturer or network operator, containing the identity of the supplier and the supplier's public key. The certificate is incorporated into the MIDlet suite's application descriptor (JAD) file.

In the case of a certification path, we need to include all the necessary certificates required to validate the JAR file. Furthermore, a MIDlet suite may include several certification paths in the application descriptor file (if, for example, the MIDlet suite supplier wishes to target several device types, each with a different root certificate). In Figure 2.13, we need to include certificates containing the public keys belonging to CA 1,

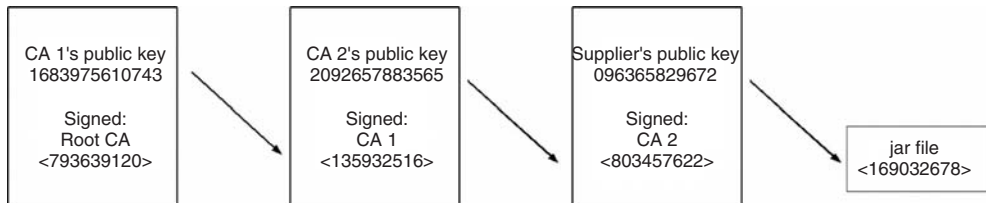


Figure 2.13 Certification path

CA 2 and the Supplier. The root certification authority's certificate (the root certificate) is available on the device. Using the root certification authority's public key, we can validate CA 1's public key. This is then used to validate CA 2's public key, which is then used to validate the Supplier's public key. The Supplier's public key is then used to verify the origin and integrity of the JAR file. The MIDP specification defines an application descriptor attribute of the following format:

```
MIDlet-Certificate-<n>-<m>: <base64 encoding of a certificate>
```

Here <n> represents the certification path and has a value of 1 for the first certification path, with each additional certification path adding 1 to the previous value (i.e. 1, 2, 3, ...). There may be several certification paths, each leading to a different root CA. <m> has a value of 1 for the certificate belonging to the signer of the JAR file and a value 1 greater than the previous value for each intermediate certificate in the certification path.

For the example shown in Figure 2.13, with just one certification path, the relevant descriptor attribute entries would have the following content:

```
MIDlet-Certificate-1-1: <base64 encoding of Supplier's certificate>
MIDlet-Certificate-1-2: <base64 encoding of CA 2's certificate>
MIDlet-Certificate-1-3: <base64 encoding of CA 1's certificate>
```

2.5.4 Authenticating a Signed MIDlet Suite

Before a MIDlet suite is installed, the Application Management Software (AMS) checks for the presence of the MIDlet-Jar-RSA-SHA1 attribute in the application descriptor and, if it is present, attempts to authenticate the JAR file by verifying the signer certificate. If it is not possible to successfully authenticate a signed MIDlet suite, it is not installed. If the MIDlet suite descriptor file does not include the MIDlet-Jar-RSA-SHA1 attribute, then the MIDlet can only be installed as untrusted.

2.5.5 Authorization Model

A signed MIDlet suite containing MIDlets which access protected APIs must explicitly request the required permissions. The MIDP specification defines two new attributes, `MIDlet-Permissions` and `MIDlet-Permissions-Opt`, for this purpose. Critical permissions (those that are required for MIDP access to protected APIs that are essential to the operation of MIDlets) must be listed under the `MIDlet-Permissions` attribute. Non-critical permissions (those required to access protected APIs without which the MIDlets can run in a restricted mode) should be listed under the `MIDlet-Permissions-Opt` attribute.

The `MIDlet-Permissions` and `MIDlet-Permissions-Opt` attributes may appear in the JAD file or the manifest of a signed MIDlet suite, or in both, in which case their respective values in each must be identical, but only the values in the manifest are 'protected' by the signature of the JAR file.

It is important to note that a MIDlet suite that has been installed as trusted is not granted any permission it has not explicitly requested in either the `MIDlet-Permissions` or `MIDlet-Permissions-Opt` attributes, irrespective of whether it would be granted were it to be requested.

The naming scheme for permissions is similar to that for Java package names. The exact name of a permission to access an API or function is defined in the specification for that API. For instance, an entry requesting permission to open HTTP and secure HTTP connections would be as follows:

```
MIDlet-Permissions: javax.microedition.io.Connector.http,  
                  javax.microedition.io.Connector.https
```

The successful authorization of a trusted MIDlet suite requires that the requested critical permissions are recognized by the device (for instance, in the case of optional APIs) and are granted, or potentially granted, in the protection domain to which the MIDlet suite would be bound, were it to be installed. If either of these requirements cannot be satisfied, the MIDlet suite is not installed.

2.5.6 Protection Domains

A protection domain is a set of permissions determining access to protected APIs or functions. A permission is either `Allowed`, in which case MIDlets in MIDlet suites bound to this protection domain have automatic access to this API, or `User`, in which case permission to access the protected API or function is requested from the user, who can then grant or

deny access. In the case of User permissions, there are three interaction modes:

- **Blanket** – as long as the MIDlet suite is installed, it has this permission unless the user explicitly revokes it.
- **Session** – user authorization is requested the first time the API is invoked and it is in force while the MIDlet is running.
- **Oneshot** – user authorization is requested each time the API is invoked.

The protection domains for a given device are defined in a security policy file. A sample security policy file is shown below:

```
alias: net_access
javax.microedition.io.Connector.http,
javax.microedition.io.Connector.https,
javax.microedition.io.Connector.datagram,
javax.microedition.io.Connector.datagramreceiver,
javax.microedition.io.Connector.socket,
javax.microedition.io.Connector.serversocket,
javax.microedition.io.Connector.ssl
domain: Untrusted
session (oneshot): net_access
oneshot (oneshot): javax.microedition.io.Connector.sms.send
oneshot (oneshot): javax.microedition.io.Connector.sms.receive
session (oneshot): javax.microedition.io.PushRegistry
domain: Symbian
allow: net_access
allow: javax.microedition.io.Connector.sms.send
allow: javax.microedition.io.Connector.sms.receive
allow: javax.microedition.io.PushRegistry
```

User permissions may offer several interaction modes, the user being able to select the level of access. For instance, the following line indicates that the API or functions defined under the `net_access` alias have User permission with either `session` or `oneshot` interaction modes, the latter being the default:

```
session (oneshot): net_access
```

2.5.7 The Security Model in Practice

In this section, we go through the steps involved in producing a signed MIDlet suite. We shall illustrate this process using the tools provided by the WTK. The basic steps in producing a signed MIDlet suite are listed below:

1. Obtain (or generate) a public–private key pair.

2. Associate the key pair with a code-signing certificate from a recommended CA.
3. Sign the MIDlet suite and incorporate the certificate into the JAD file.

To sign a MIDlet suite, the supplier of the suite needs to obtain a public–private key pair either by generating a new key pair or importing an existing key pair. The WTK provides tools for doing this; they can be accessed by opening your project and choosing the Project/Sign option from the main panel. Clicking the Sign button brings up the panel shown in Figure 2.14. To generate the key pair, click on Keystore, then New Key Pair, enter the appropriate details and click the Create button (see Figure 2.15).



Figure 2.14 Sign MIDlet Suite view of the WTK

A new key pair is generated and added to the WTK key store. The newly-generated public key is incorporated into a self-signed certificate. We use this to obtain a suitable MIDlet suite code-signing certificate from an appropriate source (such as a recommended Certification Authority, for instance, Verisign or Thawte) that can be authenticated by a root certificate that ships with the device or is contained in the WIM/SIM card. Application developers and suppliers should contact the relevant developer program of the device manufacturer or network operator to ascertain the appropriate CA.

We can then generate a Certificate Signing Request (CSR) using our self-signed certificate and the Generate CSR option in the WTK (Figure 2.16). This generates a file containing the CSR that can be saved to a convenient location. The contents of the CSR can then be copied into an email to the recommended CA, requesting a code-signing certificate.



Figure 2.15 Creating a new key pair

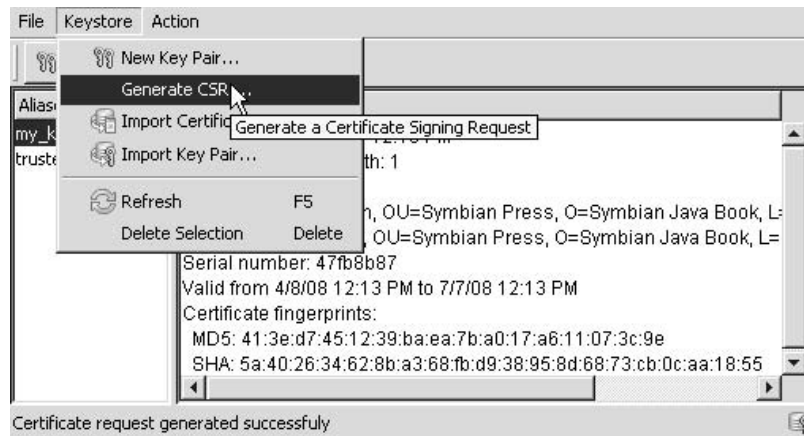


Figure 2.16 Generating a Certificate Signing Request

When we have received the certificate from the recommended CA, we need to associate this with our key pair. The Import Certificate option of the WTK associates the certificate with our key pair, identified by its alias and held in the key store. If the public key that we provided in the CSR, and now contained in the certificate, matches the public key of the key pair held in the key store, we should be notified accordingly and are now ready to sign our MIDlet suite. To sign the MIDlet suite we simply select Sign MIDlet Suite from the Action menu and choose the JAD file

belonging to the MIDlet suite we wish to sign. The MIDlet suite is now ready for deployment on the target device.

The WTK also offers additional functionality to test out signed MIDlet suites. There is a default trusted key pair (and an associated root certificate) that can be used to install and bind a signed MIDlet suite to a trusted protection domain within the emulator environment. This then allows MIDlets in the signed suite to run within the WTK environment as trusted without obtaining and importing a certificate from a CA. This feature is particularly useful for ensuring that the appropriate permissions to access protected APIs have been requested in the JAD file. It is important to remember that this feature of the WTK only works in the test environment. For real devices, you must sign your MIDlet suite with a valid certificate received from a Trusted CA.

2.5.8 Untrusted MIDlets

An untrusted MIDlet suite is an unsigned MIDlet suite. It is, therefore, installed and bound to the untrusted protection domain. Untrusted MIDlets execute within a restricted environment where access to protected APIs or functions may be prohibited or allowed only with explicit user permission, depending on the security policy in force on the device. To ensure compatibility with MIDlets developed according to the MIDP 1.0 specification, the MIDP specification demands that the untrusted domain must allow unrestricted access to the following APIs:

- `javax.microedition.rms`
- `javax.microedition.midlet`
- `javax.microedition.lcdui`
- `javax.microedition.lcdui.game`
- `javax.microedition.media`
- `javax.microedition.media.control`

Furthermore, the specification requires that the following APIs can be accessed with explicit permission of the user:

- `javax.microedition.io.HttpConnection`
- `javax.microedition.io.HttpsConnection`

The full list of permissions for the untrusted domain is device-specific, however the MIDP specification does provide a Recommended Security Policy Document for GSM/UMTS Compliant Devices as an addendum (with some clarifications added in the JTWI Final Release Policy for

Untrusted MIDlet Suites). Finally, if a signed MIDlet fails authentication or authorization, it does not run as an untrusted MIDlet, but rather is not installed by the AMS. For more information on the security model, see the MIDP specification.

2.5.9 Recommended Security Policy

The Recommended Security Policy defines a set of three protection domains (Manufacturer, Operator and Trusted Third Party) to which trusted MIDlet suites can be bound (and which a compliant device may support).

For a trusted domain to be enabled, there must be a certificate on the device, or on a WIM/SIM, identified as a trust root for MIDlet suites in that domain, i.e. if a signed MIDlet suite can be authenticated using that trust root it will be bound to that domain. For example, to enable the Manufacturer protection domain, the manufacturer must place a certificate on the device. This is identified as the trust root for the Manufacturer domain. A signed MIDlet suite will be bound to the Operator domain if it can be authenticated using a certificate found on the WIM/SIM and identified as a trust root for the Operator domain. A signed MIDlet suite will be bound to the Trusted Third Party protection domain if it can be authenticated using a certificate found on the device or on a WIM/SIM and identified as a trust root for the Trusted Third Party protection domain. Verisign and Thawte code-signing certificates usually fit the latter situation.

As already mentioned, the recommended security policy is not a mandatory requirement for an MIDP 2.0-compliant device. An implementation does not have to support the RSP in order to install signed MIDlet suites; it simply has to implement the MIDP security model and support at least one trusted protection domain.

2.6 Networking and General Connection Framework

Now that we have considered in detail the MIDP Security Model, let's learn more about the networking APIs in the specification, provided through the Generic Connection Framework (GCF).

Symbian's implementation of MIDP complies with the specification, providing implementations of the following protocols:

- HTTP
- HTTPS
- Sockets

- Server sockets
- Secure sockets
- Datagrams
- Serial port access.

In the remainder of this chapter, we focus on the networking protocols from the GCF. The local connectivity protocol (serial port access) is similar in nature to the network protocols, but its connection string depends on the target handset model, therefore you must study it separately.

2.6.1 HTTP and HTTPS Support

To open an HTTP connection we use the `Connector.open()` method with a URL of the form **`www.myserver.com`**. Code to open an `HttpConnection` and obtain an `InputStream` would look something like this:

```
try{
    String url = "www.myserver.com";
    HttpConnection conn = (HttpConnection)Connector.open(url);
    InputStream is = conn.openInputStream();
    ...
    conn.close()
}
catch(IOException ioe){...}
```

Under the MIDP security model, untrusted MIDlets can open an HTTP connection only with explicit user confirmation. Signed MIDlets that require access to an HTTP connection must explicitly request permission:

```
MIDlet-Permissions: javax.microedition.io.Connector.http, ...
```

Opening an HTTPS connection follows the same pattern as a normal HTTP connection, with the exception that we pass in a connection URL of the form **`https://www.mysecureserver.com`** and cast the returned instance to an `HttpsConnection` object, as in the following example:

```
try{
    String url = "https://www.mysecureserver.com";
    HttpsConnection hc = (HttpsConnection)Connector.open(url);
    InputStream is = hc.openInputStream();
    ...
    hc.close()
}
```

```
...
}
catch(IOException ioe){...}
```

The security policy related to access permissions by trusted and untrusted MIDlets is the same as the one for HTTP connections, except that the permission setting has a different name:

```
MIDlet-Permissions: javax.microedition.io.Connector.https, ...
```

2.6.2 Socket and Server Socket Support

MIDP makes support for socket connections a recommended practice. Socket connections come in two forms: client connections, in which a socket connection is opened to another host; and server connections in which the system listens on a particular port for incoming connections from other hosts. The connections are specified using Universal Resource Identifiers (URI). You should be familiar with the syntax of a URI from web browsing. They have the format `<string1>://<string2>` where `<string1>` identifies the communication protocol to be used (e.g., `http`) and `<string2>` provides specific details about the connection. The protocol may be one of those supported by the Generic Connection Framework (see Section 2.5.2).

To open a client socket connection to another host, we pass a URI of the following form to the connector's `open()` method:

```
socket://www.symbian.com:5000
```

The host may be specified as a fully qualified hostname or IP address and the port number refers to the connection endpoint on the remote peer. Some sample code is shown below:

```
SocketConnection sc = null;
OutputStream out = null;
try{
    sc = (SocketConnection)Connector.open ("socket://200.251.191.10:7900");
    ...
    out = c.openOutputStream();
    ...
}
catch(IOException ioe){...}
```

A server socket connection is used to listen for inbound socket connections. To obtain a server socket connection, we can pass a URI in either of the following forms to the connector's `open()` method:

```
socket://:79
socket://
```

In the first case, the system listens for incoming connections on port 79 (of the local host). In the latter case, the system allocates an available port for the incoming connections.

```
ServerSocketConnection ssc = null;
InputStream is = null;
try{
    ssc = (ServerSocketConnection)Connector.open("socket://:1234");
    SocketConnection sc = (SocketConnection)ssc.acceptAndOpen();
    ...
    is = sc.openInputStream();
    ...
}
catch(IOException ioe){...}
```

The `ServerSocketConnection` interface extends the `StreamConnectionNotifier` interface. To obtain a connection object for an incoming connection the `acceptAndOpen()` method must be called on the `ServerSocketConnection` instance. An inbound socket connection results in the call to the `acceptAndOpen()` method, returning a `StreamConnection` object which can be cast to a `SocketConnection` as desired.

A signed MIDlet suite containing MIDlets which open socket connections must explicitly request the appropriate permissions:

```
MIDlet-Permissions: javax.microedition.io.Connector.socket,
                    javax.microedition.io.Connector.serversocket, ...
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, these permissions, then the MIDlet suite is installed and the MIDlets it contains will be able to open socket connections. This can be done either automatically or with user permission, depending upon the security policy in effect on the device for the protection domain to which the MIDlet suite has been bound.

Whether MIDlets in untrusted MIDlet suites can open socket connections depends on the security policy relating to the untrusted domain in force on the device.

2.6.3 Secure Socket Support

Secure socket connections are client socket connections over SSL. To open a secure socket connection we pass in a hostname (or IP address)

and port number to the connector's `open()` method using the following URI syntax:

```
ssl://hostname:port
```

We can then use the secure socket connection in the same manner as a normal socket connection, for example:

```
try{
    SecureConnection sc = (SecureConnection)
                        Connector.open("ssl://www.secureserver.com:443");
    ...
    OutputStream out = sc.openOutputStream();
    ...
    InputStream in = sc.openInputStream();
    ...
}
catch(IOException ioe){...}
```

A signed MIDlet suite that contains MIDlets which open secure connections must explicitly request permission:

```
MIDlet-Permissions: javax.microedition.io.Connector.ssl, ...
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, this permission, the MIDlet suite can be installed and the MIDlets it contains will be able to open secure connections. This can be done automatically or with user permission, depending on the security policy in effect. Whether untrusted MIDlets can open secure connections depends on the permissions granted in the untrusted protection domain.

2.6.4 Datagram Support

Symbian's MIDP implementation includes support for sending and receiving UDP datagrams. A datagram connection can be opened in client or server mode. Client mode is for sending datagrams to a remote device. To open a client-mode datagram connection we use the following URI format:

```
datagram://localhost:1234
```

Here the port number indicates the port on the target device to which the datagram will be sent. Sample code for sending a datagram is shown below:

```
String message = "Hello!";
```

```
byte[] payload = message.toString();
try{
    UDPDatagramConnection conn = null;
    conn = (UDPDatagramConnection)
        Connector.open("datagram://localhost:1234");
    Datagram datagram = conn.newDatagram(payload, payload.length);
    conn.send(datagram);
}
catch(IOException ioe){...}
```

Server mode connections are for receiving (and replying to) incoming datagrams. To open a datagram connection in server mode, we use a URI of the following form:

```
datagram://:1234
```

The port number in this case refers to the port on which the local device is listening for incoming datagrams. Sample code for receiving incoming datagrams is given below:

```
try{
    UDPDatagramConnection dconn = null;
    dconn = (UDPDatagramConnection)Connector.open("datagram://:1234");
    Datagram dg = dconn.newDatagram(300);
    while(true){
        dconn.receive(dg);
        byte[] data = dg.getData();
        ...
    }
}
catch(IOException ioe){...}
```

A signed MIDlet suite which contains MIDlets that open datagram connections must explicitly request permission to open client connections or server connections:

```
MIDlet-Permissions: javax.microedition.io.Connector.datagram,
                    javax.microedition.io.Connector.datagramreceiver, ...
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, the requested permissions, the MIDlet suite can be installed and the MIDlets it contains will be able to open datagram connections. This can be done automatically or with user permission, depending on the security policy in effect. Whether untrusted MIDlets can open datagram connections depends on permissions granted to MIDlet suites bound to the untrusted protection domain.

2.6.5 Security Policy for Network Connections

The connections discussed above are part of the Net Access function group (see the RSP addendum to the MIDP specification). On the Nokia N95, for example, an untrusted MIDlet can access the Net Access function group with User permission (explicit confirmation required from the user). Figure 2.17 shows an example of an unsigned MIDlet, Google's Gmail, and the available permission options on a Nokia N95. This policy varies from licensee to licensee so you must check with the manufacturer of your target devices which settings apply for existing security domains.



Figure 2.17 Permission options for an unsigned MIDlet on a Nokia N95

2.6.6 NetworkDemo MIDlet

We finish this section with a simple example using MIDP's `javax.microedition.io.HttpConnection` to interrogate a web server. The `NetworkDemo` MIDlet connects to a web server via `HttpConnection`, and reads and displays all the headers and the first 256 characters of the response itself. Let's take a look at the core network functionality exploited in this MIDlet. The connection work is all done in the `ClientConnection` class:

```
public class ClientConnection extends Thread {
    private static final int TEXT_FIELD_SIZE = 256;
    private NetworkDemoMIDlet midlet = null;
    private String url = null;
```

```

public ClientConnection(NetworkDemoMIDlet midlet) {
    this.midlet = midlet;
}
public void sendMessage(String url) {
    this.url = url;
    start();
}
public void run() {
    try{
        HttpURLConnection conn = (HttpURLConnection)Connector.open(url);
        int responseCode = conn.getResponseCode();

        midlet.append("Response code","" + responseCode);
        int i = 0;
        String headerKey = null;
        while((headerKey = conn.getHeaderFieldKey(i++)) != null) {
            midlet.append(headerKey,conn.getHeaderField(headerKey));
        }

        InputStream in = conn.openInputStream();
        StringBuffer buffer = new StringBuffer(TEXT_FIELD_SIZE);
        int ch;
        int read = 0;

        while ( (ch = in.read()) != -1 && read < TEXT_FIELD_SIZE) {
            buffer.append((char)ch);
            read++;
        }

        midlet.append("HTML Response" ,buffer.toString());
        conn.close();
        conn = null;
    }
    catch(Exception e){
        e.printStackTrace();
    }
}
}

```

The `url` parameter of the `sendMessage()` method has the following form:

```
http://www.symbian.com:80
```

The `sendMessage()` method creates a request and then starts a new `Thread` to create the connection, send the request and read the response. Let us look at the contents of the thread's `run()` method in more detail:

```

HttpURLConnection conn = (HttpURLConnection)Connector.open(url);
int responseCode = conn.getResponseCode();
midlet.append("Response code","" + responseCode);
int i = 0;

```

```
String headerKey = null;
while((headerKey = conn.getHeaderFieldKey(i++)) != null) {
    midlet.append(headerKey, conn.getHeaderField(headerKey));
}
```

An `HttpConnection` is opened using the URI given by the user and is used first to retrieve the HTTP response code (200, in case of success). We also loop through the HTTP response methods until they are all read and append them to the MIDlet's Form object as a `StringItem`.

Having read the HTTP readers, it is time to read the actual HTML content. We open an `InputStream` from the `HttpConnection`, start reading the input and save it to a `StringBuffer` of `TEXT_FIELD_SIZE` length. Once we reach this value in number of bytes read, we leave the loop, as we don't want to pollute the user's screen with too much raw HTML.

```
InputStream in = conn.openInputStream();
StringBuffer buffer = new StringBuffer(TEXT_FIELD_SIZE);
int ch;
int read = 0;
while ( (ch = in.read()) != -1 && read < TEXT_FIELD_SIZE) {
    buffer.append((char)ch);
    read++;
}
```

After reading `TEXT_FIELD_SIZE` bytes of HTML, we append them to the MIDlet's Form so they can be shown on the screen along with all the other information we retrieved. Figure 2.18 shows the `NetworkDemo` MIDlet running on a Nokia N95. The full source code, JAD and JAR files for the `NetworkDemo` MIDlet are available for download from this book's website.

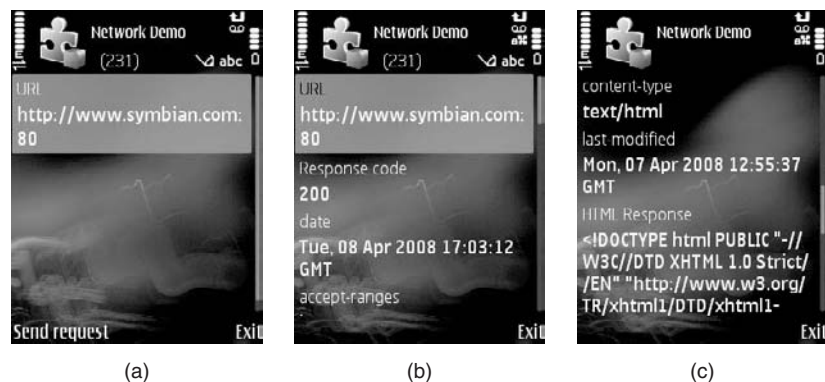


Figure 2.18 NetworkDemo MIDlet running on a Nokia N95: a) connecting to the URL, b) receiving the headers, and c) reading the HTML

2.7 Using the Push Registry

The Push Registry API is encapsulated in the `javax.microedition.io.PushRegistry` class. It maintains a list of inbound connections that have been previously registered by installed MIDlets. A MIDlet registers an incoming connection with the push registry either statically at installation via an entry in the JAD file or dynamically (programmatically) via the `registerConnection()` method.

When a MIDlet is running, it handles all the incoming connections (whether registered with the push registry or not). When the MIDlet is not running, the AMS launches the MIDlet in response to an incoming connection previously registered by it, by invoking the `startApp()` method. The AMS then hands off the connection to the MIDlet which is responsible for opening the appropriate connection and handling the communication. In the case of static registration, the MIDlet registers its interest in incoming connections in the JAD file, in the following format:

```
MIDlet-Push-<n>: <ConnectionURL>, <MIDletClassName>, <AllowedSender>
```

The `<ConnectionURL>` field specifies the protocol and port for the connection end point in the same URI syntax used by the argument to the `Connector.open()` method that is used by the MIDlet to process the incoming connection. Examples of `<ConnectionURL>` entries might be:

```
sms://:1234  
socket://:1234
```

The `<MIDletClassName>` field contains the package-qualified name of the class that extends `javax.microedition.midlet.MIDlet`. This would be the name of the MIDlet class as listed in the application descriptor or manifest file under the `MIDlet-<n>` entry. The `<AllowedSender>` field acts as a filter indicating that the AMS should only respond to incoming connections from a specific sender. For the SMS protocol, the `<AllowedSender>` entry is the phone number of the required sender. For a server socket connection endpoint, the `<AllowedSender>` entry would be an IP address (in both cases, note that the sender port number is not included in the filter). The `<AllowedSender>` syntax supports two wildcard characters: `*` matches any string including an empty string and `?` matches any character. Hence the following would be valid entries for the `<AllowedSender>` field:

```
*  
129.70.40.*  
129.70.40.23?
```

The first entry indicates any IP address, the second entry allows the last three digits of the IP address to take any value, while the last entry allows only the last digit to have any value. So the full entry for the MIDlet-Push-<n> attribute in a JAD file may look something like this:

```
MIDlet-Push-1: sms://:1234, com.symbian.devnet.ChatMIDlet, *  
MIDlet-Push-2: socket://:3000, com.symbian.devnet.ChatMIDlet, 129.70.40.*
```

If the request for a static connection registration cannot be fulfilled then the AMS must not install the MIDlet. Examples of when a registration request might fail include the requested protocol not being supported by the device, or the requested port number being already allocated to another application.

To register a dynamic connection with the AMS we use the static `registerConnection()` method of `PushRegistry`:

```
PushRegistry.registerConnection("sms://:1234",  
                                "com.symbian.devnet.ChatMIDlet", "");
```

The arguments take precisely the same format as those used to make up the MIDlet-Push-<n> entry in a JAD or manifest. Upon registration, the dynamic connection behaves in an identical manner to a static connection registered via the application descriptor. To un-register a dynamic connection, the static Boolean `unregisterConnection()` method of `PushRegistry` is used:

```
boolean result = PushRegistry.unregisterConnection(("sms://:1234");
```

If the dynamic connection is successfully unregistered, a value of `true` is returned. The AMS responds to input activity on a registered connection by launching the corresponding MIDlet (assuming that the MIDlet is not already running). The MIDlet responds to the incoming connection by launching a thread to handle the incoming data in the `startApp()` method. Using a separate thread is recommended practice for avoiding conflicts between blocking I/O operations and normal user interaction events. For a MIDlet registered for incoming SMS messages, the `startApp()` method might look something like this:

```

public void startApp() {
    // List of active connections.
    String[] connections = PushRegistry.listConnections(true);
    for (int i=0; i < connections.length; i++) {
        if(connections[i].equals("sms://:1234")){
            new Thread(){
                public void run(){
                    Receiver.openReceiver();
                }
            }.start();
        }
    }
    ...
}

```

One other use of the push registry should be mentioned before we leave this topic. The `PushRegistry` class provides a method which allows a running MIDlet to register itself or another MIDlet in the same suite for activation at a given time:

```

public static long registerAlarm(String midlet, long time)

```

The `midlet` argument is the class name of the MIDlet to be launched at the time specified by the `time` argument. The launch time is specified in milliseconds since January 1, 1970, 00:00:00 GMT. The push registry may contain only one outstanding activation time entry per MIDlet in each installed MIDlet suite. If a previous activation entry is registered, it is replaced by the current invocation and the previous value is returned. If no previous wakeup time has been set, zero is returned.

The `PushRegistry` is a protected API; a signed MIDlet suite which registers connections statically or contains MIDlets which register connections or alarms must explicitly request permission:

```

MIDlet-Permissions: javax.microedition.io.PushRegistry, ...

```

Note that a signed MIDlet suite must also explicitly request the permissions necessary to open the connection types of any connections it wishes to register. If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, the requested permission, the MIDlet suite can be installed and the MIDlets it contains can register and deregister connections and alarms, either automatically or with user permission, depending on the security policy in effect.

Untrusted MIDlets do not require a `MIDlet-Permissions` entry. Whether access is granted to the Push Registry API depends on the security policy for the untrusted protection domain in effect on the device.

On the Nokia N95, untrusted MIDlet suites can use the Push Registry APIs (Application Auto-Start function group) with user permission. The default User permission is set to Session ('Ask first time'). It can be changed to 'Not allowed' or 'Ask every time'.

2.8 MIDP and the JTWI

The Java Technology for the Wireless Industry (JTWI) initiative is part of the Java Community Process (JSR-185) and its expert group has as its goal the task of defining an industry-standard Java platform for mobile phones, by reducing the need for proprietary APIs and providing a clear specification that phone manufacturers, network operators and developers can target. The JTWI specification concerns three main areas:

- It provides a minimum set of APIs (JSRs) that a compliant device should support.
- It defines which optional features within these component JSRs must be implemented on a JTWI-compliant device.
- It provides clarification of component JSR specifications, where appropriate.

2.8.1 Component JSRs of the JTWI

The JTWI defines three categories of JSR that fall under the specification: mandatory, conditionally required and minimum configuration. The following mandatory JSRs must be implemented as part of a Java platform that is compliant with JTWI:

- MIDP (JSR-118)
- Wireless Messaging API (JSR-120).

The Mobile Media API (JSR-135) is conditionally required in the JTWI. It must be present if the device exposes multimedia APIs (e.g., audio or video playback or recording) to Java applications. The minimum configuration required for JTWI compliance is CLDC 1.0 (JSR-30). Since CLDC 1.1 is a superset of CLDC 1.0 it may be used instead, in which case it supersedes the requirement for CLDC 1.0. Today most of the Symbian OS devices in the market support CLDC 1.1.

2.8.2 JTWI Specification Requirements

As mentioned earlier, the JTWI specification makes additional requirements on the implementation of the component JSRs. For full details,

consult the JTWI specification available from the Java Community Process (JCP) website (<http://jcp.org>).

- CLDC 1.0/1.1 must allow a MIDlet suite to create a minimum of 10 running threads and must support Unicode characters.
- MIDP 2.0 must allow creation of at least five independent record stores, must support the JPEG image format and must provide a mechanism for selecting a phone number from the device's phone-book when the user is editing a `TextField` or `TextBox` with the `PHONENUMBER` constraint.
- GSM/UMTS phones must support SMS protocol push handling within the Push Registry.
- MMA must support MIDI playback (with volume control), JPEG encoding for video snapshots and the Tone Sequence file format.

The JTWI specification also clarifies aspects of the MIDP recommended security policy for GSM/UMTS devices relating to untrusted domains.

2.8.3 Symbian and the JTWI

Symbian supports and endorses the efforts of the JTWI and is a member of the JSR-185 expert group. Releases of Symbian OS from version 9.1 provide implementations of the mandatory JSRs (Wireless Messaging API and Mobile Media API) and configurations (CLDC 1.0/1.1 and MIDP 2.0) specified by the JTWI. They also provide many other JSR implementations, about which you can find information on Symbian's website.

In 2003, JTWI was still in its draft phase, therefore only a couple of Symbian OS phones supported it, and only partially. At the time of writing (2008), the vast majority of Symbian devices supports the full range of JTWI specification plus many optional APIs. Developers can therefore rely on JSR-185 support on Symbian devices for their JTWI-based applications.

2.9 Mobile Media API

The Media API is fairly limited, giving support only to basic audio types. Smartphones these days have many more multimedia capabilities. Therefore it is necessary to expose all this multimedia functionality, such as tone generation, photo capture, and video playback and capture, to Java ME applications, so developers can create truly compelling multimedia applications attractive to end users.

This was achieved through the creation of JSR-135 – Mobile Media API (MMAPI). MMAPI is indicated for a JTWI-compliant device. Note that

it does not form part of the MIDP 2.0 specification; its specification in JSR-135 is presided over by an expert group.

Due to the wide scope encompassed by the word ‘multimedia’, MMAPI is very flexible and modular; it supports many different devices with different multimedia capabilities in a simple, understandable way for the developer. There are essentially three media types which can be handled within the framework: audio, video and generated tones. Any or all of these may be supported in a particular MMAPI implementation. Three tasks can be performed in relation to audio and video:

- playing – stored video or audio content is recovered from a file at a specified URI (or perhaps stored locally) and displayed onscreen or sent to a speaker
- capturing – a video or audio stream is obtained directly from hardware (camera or microphone) associated with the device
- recording – video or audio content which is being played or captured is sent to a specified URI or cached locally and made available for ‘re-playing’.

Implementations are not required to support all three tasks, although clearly it is not possible to support recording without supporting either playing or capturing. Further, in the context of a mobile phone, there is little point in supporting capture without the ability to play. So the practical options for audio and video are:

- only playing is supported
- playing and capturing are supported, but not recording
- playing and recording are supported, but not capturing
- all three are supported.

As we shall see, Symbian OS phones typically support all three functions. There are further choices in terms of supported formats and the ability to manipulate data streams or playback.

2.9.1 MMAPI Architecture

At the heart of MMAPI is the concept of a media player. Players are obtained from a factory or manager which also serves to associate them with a particular media stream. While the player allows basic start and stop capability for the playback, fine-grained manipulation, such as capturing and recording, is achieved through various kinds of controls, which are typically obtained from a player. Also of interest is the concept

of a player listener, which allows you to track the progress of players being initialized, started or stopped.

The following classes or interfaces embody the above basic concepts and are the fundamental elements of the core package `javax.microedition.media`:

- `Player`
- `Manager`
- `Control`
- `PlayerListener`.

In fact, of these only `Manager` is a concrete class; the others are all interfaces. The `Control` interface exists merely as a marker for the set of (more concrete) sub-interfaces which are defined in the related package `javax.microedition.media.control`. The functionality provided by the latter is so diverse that nothing is in common, and the `Control` interface, remarkably, defines no methods!

MMAPI also allows you the flexibility to define your own protocols for downloading or obtaining the media content to be played. This involves defining concrete implementations of the abstract `DataSource` class. As this is a specialist topic beyond the scope of this chapter, we shall not say anything further. The MMAPI specification document contains further details.

2.9.2 Obtaining Media Content

The elements of the Mobile Media API work together as shown in Figure 2.19 (the `PlayerListener` has been omitted and we shall return to it in a later section).

A `Player` is typically created using the following factory method of the `javax.microedition.media.Manager` class:

```
public static Player createPlayer(String locator)
```

The method's argument is a media locator, a string representing a URI that provides details about the media content being obtained. The details are specified using the well-documented Augmented Backus–Naur Format (ABNF) syntax. Table 2.2 lists some examples. MMAPI supports both communication and local protocols, which allow retrieval of the desired media from its source.

There are other variants of the `createPlayer()` method that take different parameters, such as an `InputStream` and a MIME type (for media stored locally) or a custom `DataSource` implemented by the

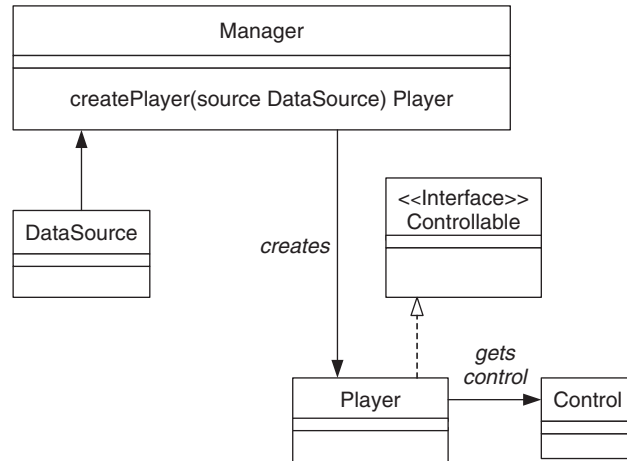


Figure 2.19 The basic architecture of the Mobile Media API

Table 2.2 Examples of Media Locator

Locator	Purpose	Examples
capture://	Retrieving live media data from hardware	capture://audio – captures microphone audio capture://video – captures camera output capture://audio_video – captures audio and video simultaneously (for some devices, only capture://video is necessary)
device://	Configuring players for tone sequences or MIDI data	device://tone – tone device device://midi – midi device
rtsp:// rtp://	Media streaming, where playback starts before download is complete	rtsp://streamer.why.org/encoder/live.rm
http://	Fetching media stored on a web server	http://www.yourhost.com/myfile.mp3

developer. Please refer to the SDK documentation for more details on obtaining media using the player creation methods of the `Manager` class.

2.9.3 Playing Media Content

Because of the complexity of what a `Player` does, there are necessarily several stages (or states) it has to go through before it can play (see Figure 2.20). As we have just seen, the first stage is the creation of a `Player` object via the `Manager` factory class. This involves the creation of a `DataSource` object, which provides a standard interface to the media content. Unless you are working with a custom `DataSource`, the creation is done on your behalf. It's worth noticing that creating the `Player` object does not initialize the data transfer, which begins at the next stage.

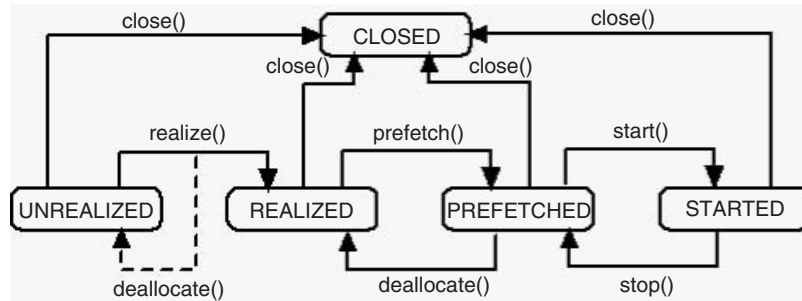


Figure 2.20 Lifecycle of a `Player` object

On creation, the `Player` is in the `UNREALIZED` state. Calling the `realize()` method causes the `Player` to initiate data transfer, for example, communicating with a server or a file system. Peer classes to marshal the data on the native side are typically instantiated at this point. When this method returns, the player is in the `REALIZED` state. Calling `prefetch()` causes the `Player` to acquire the scarce and exclusive resources it needs to play the media, such as access to the phone's audio device. It may have to wait for another application to release these resources before it can move to the `PREFETCHED` state. Once in this state, the `Player` is ready to start. A call to its `start()` method initiates playing and moves it on to the `STARTED` state. In order to interrogate the state of the `Player`, the `getState()` method of the `Player` class is provided.

In many cases, of course, clients of `MMAPI` will not be interested in the fine distinctions of which resources are acquired by which methods. The good news is that you are free to ignore them if you wish: a call to `start()` on a `Player` in any state other than `CLOSED` results in any

intermediate calls needed to `realize()` or `prefetch()` being made implicitly. Of course, the price you pay is less fine-grained control of exception handling.

The matching methods to stop the `Player` are `close()`, `deallocate()` and `stop()`. As with the `start()` method, the `close()` method encompasses the other two, so they need not be invoked on a `Player` directly. You should be aware, however, that reaching the end of the media results in the `Player` returning to the `PREFETCHED` state, as though the `stop()` method had been called. The good thing about this is that you can then conveniently replay the media by calling `start()` again. However, you must call the `close()` method explicitly to recover all the resources associated with realization and prefetching and to set to `NULL` all references to your `Player` so the garbage collector can dispose of it. (You do want to dispose of it, since a closed `Player` cannot be reused!)

In playing media content, it is often useful to work with one or more `Control` objects that allow you to control media processing. They are obtained from an implementer of the `Controllable` interface, in most cases a `Player`, using one of the following methods:

```
Control getControl(String controlType);  
Control[] getControls();
```

A media player of a given type may support a variety of controls. The string passed in determines the name of the interface implemented by the returned control, which is typically one of the pre-defined types in the `javax.microedition.media.control` subpackage:

- `FramePositioningControl`
- `GUIControl`
- `MetaDataControl`
- `MIDIControl`
- `PitchControl`
- `RateControl`
- `TempoControl`
- `RecordControl`
- `StopTimeControl`
- `ToneControl`
- `VideoControl`
- `VolumeControl`.

If the type of control you want is not available, a `NULL` value is returned (which you should always check for). You will also need to cast the control appropriately before using it:

```
VolumeControl volC = (VolumeControl) player.getControl("VolumeControl");
if (volC != null)
    volC.setVolume(50);
```

The availability of support for controls depends on a number of factors, such as the media type and the phone model. Only `ToneControl` and `VolumeControl` are available as part of the MIDP audio subset. The remainder are specific to MMAPI. You can check which controls are supported by a certain `Player` by calling its `getControls()` method, which returns an array containing all available controls. You can then use `instanceof` to ascertain whether the control you want is available:

```
Control[] controls = player.getControls();
for (int i = 0; i < controls.length; i++) {
    if (controls[i] instanceof VolumeControl) {
        VolumeControl volC = (VolumeControl) controls[i];
        volC.setVolume(50);
    }
    if (controls[i] instanceof VideoControl) {
        VideoControl vidC = (VideoControl) controls[i];
        vidC.setDisplayFullScreen(true);
    }
}
// allow controls to be garbage collected
controls = null;
```

Note that `getControl()` and `getControls()` cannot be invoked on a `Player` in the `UNREALIZED` or `CLOSED` states; doing so will cause an `IllegalStateException` to be thrown.

Aside from using a `Player`, there is a further option to play simple tones or tone sequences directly using the static `Manager.playTone()` method. However, you normally want the additional flexibility provided by working with a `Player` (configured for tones) and a `ToneControl` (see Section 2.9.8).

The `PlayerListener` interface provides a `playerUpdate()` method for receiving asynchronous events from a `Player`. Any user-defined class may implement this interface and then register the `PlayerListener` using the `addPlayerListener()` method. The `PlayerListener` listens for a range of standard pre-defined events including `STARTED`, `STOPPED` and `END_OF_MEDIA`. For a list of all the standard events refer to the MMAPI specification.

2.9.4 Working with Audio Content

In this section, we demonstrate how to play audio files with code from a simple Audio Player MIDlet. We focus on the actual MMAPI code used to play the audio file.

```
// code abbreviated for clarity
public void run(){
    try {
        player = Manager.createPlayer(url);
        player.addPlayerListener(controller);
        player.realize();
        player.prefetch();
        volumeControl = (VolumeControl)player.getControl("VolumeControl");

        if (volumeControl != null) {
            volumeControl.setLevel(50);
        }
        else {
            controller.removeVolumeControl();
        }
        startPlayer();
    }
    catch (IOException ioe) {
        // catch exception connecting to the resource
    }
    catch (MediaException me) {
        // unable to create player for given MIME type
    }
}
```

A `Player` is created and a `PlayerListener` is registered with it. The controller reference serves two purposes: to facilitate callbacks to the UI indicating the progress of the initialization (this has been omitted for clarity); and to act as the `PlayerListener` which will be notified of `Player` events. The `Player` is then moved through its states. In this example, we obtain a `VolumeControl` (if the implementation supports this feature) although it is not essential for simple audio playback. The volume range provided is from 0–100. Here we set the volume level midway and start the `Player` so the audio file content can be heard from the phone's speakers.

Closing the player is a straightforward task:

```
public void closePlayer(){
    if (player != null){
        player.close();
    }
    player = null;
    volumeControl = null;
}
```

Now let us consider the `playerUpdate()` method mandated by the `PlayerListener` interface:

```
public void playerUpdate(Player p, String event, Object eventData) {  
    if (event == PlayerListener.STARTED) {  
        // react to the Player being started  
    }  
    else if (event == PlayerListener.END_OF_MEDIA){  
        // react to reaching the end of media.  
    }  
    else if (event == PlayerListener.VOLUME_CHANGED) {  
        // react to volume being changed  
    }  
}
```

In this example, three types of `PlayerListener` event are processed: `STARTED`, `END_OF_MEDIA` and `VOLUME_CHANGED`. For a complete list of events supported by `PlayerListener`, please check its documentation.

The full source code and JAR and JAD files for the Audio Player MIDlet can be downloaded from this book's website.

2.9.5 Working with Video Content

We now illustrate how to play a video with code highlights taken from a simple Video Player MIDlet (see Figure 2.21). The architecture of the Video Player MIDlet is very similar to that of the Audio Player. The



Figure 2.21 Video Player MIDlet running on a Nokia S60 device

VideoCanvas renders the video playback and the other classes fulfill very similar roles to their equivalents in the Audio Player MIDlet.

The resource file name is tested to ascertain its format (MPEG for the WTK emulator and 3GPP for real phones) and the appropriate MIME type. A new thread is then launched to perform the essential initialization required to play the video content. The `run()` method, mandated by the `Runnable` interface, contains the initialization of the `Player`:

```
public void run(){
    try {
        InputStream in = getClass().getResourceAsStream("/" + resource);
        player = Manager.createPlayer(in, mimeType);
        player.addPlayerListener(controller);
        player.realize();
        player.prefetch();
        videoControl = (VideoControl)player.getControl("VideoControl");
        if (videoControl != null) {
            videoControl.initDisplayMode(
                videoControl.USE_DIRECT_VIDEO, canvas);
            int cHeight = canvas.getHeight();
            int cWidth = canvas.getWidth();
            videoControl.setDisplaySize(cWidth, cHeight);
            videoControl.setVisible(true);
            startPlayer();
        }
        else {
            controller.showAlert("Error!", "Unable to get Video Control");
            closePlayer();
        }
    }
    catch (IOException ioe) {
        controller.showAlert("Unable to access resource", ioe.getMessage());
        closePlayer();
    }
    catch (MediaException me) {
        controller.showAlert("Unable to create player",
            me.getMessage());
        closePlayer();
    }
}
```

An `InputStream` is obtained from the resource file and is used to create the `Player` instance. A `PlayerListener` (the controller) is registered with the `Player` in order to receive callbacks. The `prefetch()` and `realize()` methods are then called on the `Player` instance. Once the player is in the `PREFETCHED` state, we are ready to render the video content. First we must obtain a `VideoControl` by calling `getControl()` on the `Player` and casting it down appropriately.

The `initDisplayMode()` method is used to initialize the video mode that determines how the video is displayed. This method takes an integer mode value as its first argument with one of two predefined values: `USE_GUI_PRIMITIVE` or `USE_DIRECT_VIDEO`. In the case of MIDP

implementations (supporting the LCDUI), `USE_GUI_PRIMITIVE` results in an instance of a `javax.microedition.lcdui.Item` being returned, and it can be added to a `Form` in the same way as any other `Item` subclass. `USE_DIRECT_VIDEO` mode can only be used with implementations that support the LCDUI (such as Symbian OS) and a second argument of type `javax.microedition.lcdui.Canvas` (or a subclass) must be supplied. This is the approach adopted in the example code above. Methods of `VideoControl` can be used to manipulate the size and the location of the video with respect to the canvas where it is displayed. Since we are using direct video as the display mode, it is necessary to call `setVisible(true)` in order for the video to be displayed. Finally, we start the rendering of the video with the `startPlayer()` method.

The other methods of the `VideoPlayer` class are the same as their namesakes in the `AudioPlayer` class of the Audio Player MIDlet.

The `VideoCanvas` class, where the video is shown, is very simple, as the MMAPI implementation takes care of rendering the video file correctly on the canvas:

```
public class VideoCanvas extends Canvas{
    // code omitted for brevity

    // Paints background color
    public void paint(Graphics g){
        g.setColor(128, 128, 128);
        g.fillRect(0, 0, getWidth(), getHeight());
    }
}
```

The important point to note is that the `paint()` method plays no part in rendering the video. This is performed directly by the `VideoControl`. The full source code and JAR and JAD files for the Video Player MIDlet can be downloaded from this book's website.

2.9.6 Capturing Images

`VideoControl` is also used to capture images from a camera. In this case, rather than specifying a file (and MIME type) as the data source, we specify `capture://video`. Other than that, the setting up of the video player and control proceeds pretty much as in the Video Player MIDlet in Section 2.9.5.

The following code, which performs the necessary initialization of a video player and a control, is reproduced from the `VideoPlayer` class in the Video Player MIDlet example:

```
// Creates a VideoPlayer and gets an associated VideoControl
public void createPlayer() throws ApplicationException {
    try {
        player = Manager.createPlayer("capture://video");
        player.realize();
        // Sets VideoControl to the current display.
        videoControl = (VideoControl)(player.getControl("VideoControl"));
        if (videoControl == null) {
            discardPlayer();
        }
        else {
            videoControl.initDisplayMode(VideoControl.USE_DIRECT_VIDEO, canvas);
            int cWidth = Canvas.getWidth();
            int cHeight = Canvas.getHeight();
            int dWidth = 160;
            int dHeight = 120;
            videoControl.setDisplaySize(dWidth, dHeight);
            videoControl.setDisplayLocation((cWidth - dWidth)/2,
                                           (cHeight - dHeight)/2);
        }
    }
}
```

By setting the canvas to be the current one in the display, we can use it as a viewfinder for the camera. When we are ready to take a picture, we simply call `getSnapshot(null)` on the `VideoControl`:

```
public byte[] takeSnapshot() throws ApplicationException {
    byte[] pngImage = null;
    if (videoControl == null) {
        throw new ApplicationException
            ("Unable to capture photo: VideoControl null");
    }
    try {
        pngImage = videoControl.getSnapshot(null);
    }
    catch(MediaException me) {
        throw new ApplicationException("Unable to capture photo",me);
    }
    return pngImage;
}
```

It should be noted that, if a security policy is in operation, user permission may be requested through an intermediate dialog, which may interfere with the photography!

2.9.7 Generating Tones

MMAPI also supports tone generation. Generating a single tone is simply achieved using the following method of the `Manager` class:

```
public static void playTone(int note, int duration, int volume)
    throws MediaException
```


The note is passed as an integer value in the range 0–127; `ToneControl.C4 = 60` represents middle C. Adding or subtracting 1 increases or lowers the pitch by a semitone. The duration is specified in milliseconds and the volume is an integer value on the scale 0–100.

To play a sequence of tones it is more appropriate to create a `Player` and use it to obtain a `ToneControl`:

```
byte[] toneSequence = { ToneControl.C4, ToneControl.C4 + 2,
                        ToneControl.C4 + 4, ...};

try{
    Player player = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
    player.realize();
    ToneControl control = (ToneControl)player.getControl("ToneControl");
    control.setSequence(toneSequence);
    player.start();
}
catch (IOException ioe) { }
catch (MediaException me) { // handle }
```

A tone sequence is specified as a list of tone–duration pairs and user-defined sequence blocks, using ABNF syntax (refer to the MMAPI specification for more detail). The list is packaged as a byte array and passed to the `ToneControl` using the `setSequence()` method. To play the sequence, we simply invoke the `start()` method of the `Player`. A more sophisticated example can be found in the documentation of `ToneControl` in the MMAPI specification.

2.9.8 MMAPI on Symbian OS Phones

If you know which of the Symbian OS platforms you are targeting with a MIDlet, you can craft your code to conform to the cited capabilities. However, in practice it is more likely that you will want to write portable code which can run on several or all of the platforms, or indeed on non-Symbian OS phones with MMAPI capability. In this case, your applications need to be able to work out the supported capabilities dynamically and make use of what is available, or fail gracefully (for example, by removing certain options from menus), if the capability you want is just not available.

2.9.9 MMAPI and the MIDP Security Model

For reasons of privacy, the following Mobile Media API calls are restricted under the MIDP security model (see the *Mobile Media API Specification 1.1 Maintenance Release* at jcp.org):

- `RecordControl.setRecordLocation(String locator)`

- `RecordControl.setRecordStream(OutputStream stream)`
- `VideoControl.getSnapshot(String type).`

A signed MIDlet suite which contains MIDlets that make use of these APIs must explicitly request the appropriate permission in the JAD file or manifest:

```
MIDlet-Permissions: javax.microedition.media.control.RecordControl, ...
```

or:

```
MIDlet-Permissions:  
    javax.microedition.media.control.VideoControl.getSnapshot, ...
```

These protected APIs are part of the Multimedia Recording function group as defined by the *Recommended Security Policy for GSM/UMTS Compliant Devices* addendum to the MIDP specification. It must also be remembered that if a MIDlet in a signed MIDlet suite makes use of a protected API of the `javax.microedition.io` package, for instance to fetch media content over HTTP, then explicit permission to access that API must be requested, even if it is fetched implicitly, perhaps by calling:

```
Manager.createPlayer("http://www.myserver.com/video.3gp")
```

Whether MIDlets in an untrusted MIDlet suite can use the protected APIs of the MMAPI depends on the security policy relating to the untrusted domain in force on the device. Under the *JTWI Security Policy for GSM/UMTS Compliant Devices*, MIDlets in an untrusted MIDlet suite can access the Multimedia Recording function group APIs with explicit permission from the user. The default user permission setting is `oneshot` (ask every time).

2.10 Wireless Messaging API

The Wireless Messaging API (JSR-120) is an optional API targeted at devices supporting the Generic Connection Framework defined in the CLDC. The Wireless Messaging API (WMA) specification defines APIs for sending and receiving SMS messages and receiving CBS messages. At the time of writing, the current release of the Wireless Messaging API is version 1.1. This contains minor modifications to the 1.0 specification to enable the API to be compatible with MIDP 2.0.

The WMA is a compact API containing just two packages:

- `javax.microedition.io` contains the platform network interfaces modified for use on platforms supporting wireless messaging connection, in particular an implementation of the `Connector` class for creating new `MessageConnection` objects.
- `javax.wireless.messaging` defines APIs which allow applications to send and receive wireless messages. It defines a base interface, `Message`, from which `BinaryMessage` and `TextMessage` both derive. It also defines a `MessageConnection` interface, which provides the basic functionality for sending and receiving messages, and a `MessageListener` interface for listening to incoming messages.

In this section, we consider sending and receiving SMS messages. We then go on to show how to use the Push Registry API to register an incoming SMS connection with a MIDlet.

2.10.1 Sending Messages

Sending an SMS message using the WMA could not be simpler, as the code paragraph below shows:

```
String address = "sms://+447111222333";
MessageConnection smsconn = null;
try {
    smsconn = (MessageConnection)Connector.open(address);
    TextMessage txtMessage = (TextMessage)
        smsconn.newMessage(MessageConnection.TEXT_MESSAGE);
    txtMessage.setPayloadText("Hello World");
    smsconn.send(txtMessage);
    smsconn.close();
}
catch (Exception e) {
    // handle
}
```

The URL address syntax for a client-mode connection has the following possible formats:

```
sms://+447111222333
sms://+447111222333:1234
```

The first format above is used to open a connection for sending a normal SMS message, which will be received in an inbox. The second format is used to open a connection to send an SMS message to a Java application listening on the specified port.

2.10.2 Receiving Messages

Receiving a message is, again, straightforward:

```
MessageConnection smsconn = null;
Message msg = null;
String receivedMessage = null;
String senderAddress = null;
try {
    conn = (MessageConnection) Connector.open("sms://:1234");
    msg = smsconn.receive();
    ...
    // get sender's address for replying
    senderAddress = msg.getAddress();
    if (msg instanceof TextMessage) {
        // extract text message
        receivedMessage = ((TextMessage)msg).getPayloadText();
        // do something with message
        ...
    }
}
catch (IOException ioe) {
    ioe.printStackTrace();
}
```

We open a server mode `MessageConnection` by passing in a URL of the following syntax:

```
sms://:1234
```

We retrieve the message by invoking the following method on the `MessageConnection` instance.

```
public Message receive()
```

The address of the message sender can be obtained using the following method of the `Message` interface:

```
public String getAddress()
```

A server mode connection can be used to reply to incoming messages, by making use of the `setAddress()` method of the `Message` interface. In the case of a text message, we cast the `Message` object appropriately and then retrieve its contents with the `TextMessage` interface, using the method below:

```
public String getPayloadText()
```

If the message is an instance of `BinaryMessage`, then the corresponding `getPayloadData()` method returns a byte array. In practice, of course, we need the receiving application to listen for incoming messages and invoke the `receive()` method upon receipt. We achieve this by implementing a `MessageListener` interface for notification of incoming messages. The `MessageListener` mandates one method, which is called on registered listeners by the system when an incoming message arrives:

```
public void notifyIncomingMessage(MessageConnection conn)
```

The `MessageConnection` interface supplies the following method to register a listener:

```
public void setMessageListener(MessageListener l)
```

For more details on the use of the `MessageListener` interface, please download the source code, JAD and JAR files for the `SMSSChat` MIDlet from this book's website.

2.10.3 WMA and the Push Registry

When implemented in conjunction with MIDP 2.0, the Wireless Messaging API can take advantage of the push registry technology. A MIDlet suite lists the server connections it wishes to register in its JAD file, or manifest, by specifying the protocol and port for the connection end point. The entry has the following format:

```
MIDlet-Push-<n>: <ConnectionURL>, <MIDletClassName>, <AllowedSender>
```

In this example, the entry in the JAD file would be as follows:

```
MIDlet-Push-1: sms://:1234, SMSMIDlet, *
```

The `<AllowedSender>` field acts as a filter indicating that the AMS should only respond to incoming connections from a specific sender. For the SMS protocol, the `<AllowedSender>` entry is the phone number of the required sender (note the sender port number is not included in the filter). Here the wildcard character `'*` indicates 'respond to any sender'. The AMS responds to an incoming SMS directed to the specified `MessageConnection` by launching the corresponding MIDlet

(assuming it is not already running). The MIDlet should then respond by immediately handling the incoming message in the `startApp()` method. As before, the message should be processed in a separate thread to avoid conflicts between blocking I/O operations and normal user interaction events.

2.10.4 WMA and the MIDP Security Model

A signed MIDlet suite that contains MIDlets which open and use SMS connections must explicitly request the following permissions:

- `javax.microedition.io.Connector.sms` – needed to open an SMS connection
- `javax.wireless.messaging.sms.send` – needed to send an SMS
- `javax.wireless.messaging.sms.receive` – needed to receive an SMS

```
MIDlet-Permissions: javax.microedition.io.Connector.sms,
                   javax.wireless.messaging.sms.send
```

or:

```
MIDlet-Permissions: javax.microedition.io.Connector.sms,
                   javax.wireless.messaging.sms.send,
                   javax.wireless.messaging.sms.receive
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, the requested permissions, the MIDlet suite can be installed and the MIDlets it contains can open SMS connections and send and receive SMS messages. This can be done automatically or with explicit user permission, depending upon the security policy in effect.

Whether MIDlets in untrusted MIDlet suites can access the WMA depends on the security policy relating to the untrusted domain in force on the device. In line with the *Recommended Security Policy for GSM/UMTS Compliant Devices* addendum to the MIDP specification and the *JTWI Security Policy for GSM/UMTS Compliant Devices*, a messaging function group permission of `oneshot` requires explicit user permission to send an SMS message, but allows blanket permission (permission is granted until the MIDlet suite is uninstalled or the user changes the function group permission) to receive SMS messages.

2.10.5 WMA on Symbian OS Phones

Most Symbian OS phones in the marketplace today, even older ones such as the Nokia 3650, implement Wireless Messaging API (JSR-120). WMA has a new version (defined in JSR-205) available on most modern devices as well; its main benefit is to add Multimedia Messaging System (MMS) capabilities to the previous version of WMA. However, we won't mention it here further since only JSR-120 is mandatory on JTWI-compliant devices. WMA is a mandatory part of another umbrella specification, Mobile Service Architecture (JSR-248), which is covered in Chapter 6.

2.11 Symbian OS Java ME Certification

Majinate (www.majinate.com) is a company from the UK which specializes in building and testing competence in application development for Symbian OS. It operates the Accredited Symbian Developer (ASD) program on behalf of Symbian and the Accredited S60 Developer program on behalf of Nokia.

In addition to the certification exams for native application developers, Majinate also offers a test for Java ME developers working on the Symbian OS platform. The Symbian OS: J2ME exam covers the Java language and MIDP application programming concepts. It focuses on features supported by Symbian OS implementations, with the exam testing the ability of a developer to address the differences between Symbian and other Java platforms. Some of the main topics covered are:

- Java language basics
- object-orientation concepts
- package `java.lang`
- MIDlet deployment and MIDP 2 security model
- MIDlet class and lifecycle
- generic connection framework and networking
- LCDUI GUI applications
- RMS and utilities
- Java ME optional packages (Wireless Messaging and Mobile Media APIs)

This book and the standard sources (the specifications and their clarifications) are the key references for this exam. These sources have guided the creation of the curriculum and database of test questions.

For more details on the Symbian OS J2ME exam, see www.majinate.com/j2me.php.

2.12 Summary

Most, if not all, Symbian OS devices currently selling in Western markets support MIDP plus a wide range of optional APIs from the Java ME JSRs, such as:

- Wireless Messaging API (JSR-120)
- Bluetooth API (JSR-82)
- PIM and FileConnection API (JSR-75)
- Mobile Media API (JSR-135)
- Mobile 3D Graphics (JSR-184).

JSR-120 and JSR-135 have been discussed here. The others are discussed in the following chapters. The latest generation of Symbian OS phones, such as Nokia N96 and Sony Ericsson G900, supports MIDP and the Mobile Service Architecture (JSR-248) APIs.

This is a very different (and much more exciting) picture from the one we found in 2004, when only two models, Nokia 6600 and Sony Ericsson P900, supported the MIDP specification. The large installed base of devices that are enabled by MIDP 2.0+ running on Symbian OS these days allows developers to create and distribute sophisticated applications using Java ME APIs to a wide target market.

We have covered the basic operations such as building and packaging MIDlets using the WTK. We have also had a good look at the APIs and components (such as LCDUI, RMS and the Game API) of MIDP, giving you the baseline information that is a prerequisite for reading the rest of the book, which assumes a certain level of knowledge of Java programming and specifically Java ME.

Lastly, we have covered the relation between MIDP and the JTWI specifications and seen how these two work hand in hand to reduce API fragmentation and provide a solid and consistent environment for developing applications with multimedia and messaging features, thus reducing the need to maintain multiple versions of your application.

Part Two

Java ME on Symbian OS

3

Enter Java ME on Symbian OS

Once you've read Part Two, you can proudly claim that you understand Java ME on Symbian OS. Thinking about it, why can't you do that already? As you noticed, no new tricks were mentioned in Chapter 2. Java is Java, ME is ME and it is all too familiar. Even suspiciously familiar . . . How can Java ME on Symbian OS be the same as everywhere yet more than anywhere? And at the same time? Well, it is true! Just to confirm again, Java ME on Symbian OS is still Java ME. Only, there's much more!

Chapter 2 defined the prerequisite knowledge required for this book. Having reached this point in the book, we can safely assume that you are familiar with MIDlets, MIDlet suite, JAR, JAD, JSR, LCDUI, GCF, RMS, WMA, OTA, permissions, and so on. (If any of these terms is unfamiliar to you, please go back and read Chapter 2 or another Java ME tutorial.¹) We assume that you are not yet familiar with Java ME on Symbian OS, from a platform point of view. Java ME, the leading mobile development technology, exposes the strength and richness of Symbian OS, the leading smartphone platform, through standardized Java APIs.

In this chapter, we first run an application on a real mobile phone. Then, we consider additional APIs and which JSRs are supported. The next two sections discuss how the lack of fixed limits on computing resources and the powerful native platform give Java ME on Symbian OS more power. We then demonstrate how Java and native code can interoperate. We finish the discussion with a few more practical topics, such as Java ME management on Symbian smartphones and a preview of Chapter 4, and describe a structured approach on how and where to find more information about Java ME on Symbian OS.

This chapter is an introductory crash course. We assume that if you know Java ME, you are ready to learn a few things that will open your eyes to the possibilities that are waiting for you in the Symbian world. (Another title to the chapter could have been 'Very important stuff that you should know about Java ME on Symbian OS'.)

¹ developers.sun.com/mobility/learning/tutorial

There are many Symbian smartphones. In many cases, giving a single answer that applies to all models, versions, manufacturers, and so on, is not possible. Attempting to give such an answer is simply is not the right approach. Instead, our intention is to direct you towards asking the right questions and to equip you with the right mindset, approach, tools and the relevant information sources. Knowing what you know, you can safely steer yourself in a direction appropriate for you.

So let's start the journey. There is fun to be had!

3.1 Running a MIDlet on a Symbian Smartphone (not another 'Hello World')

Being able to deploy and run an application on a target Symbian smartphone is a prerequisite for any discussion or development.

A Symbian smartphone can be built on different versions of Symbian OS, can come from different phone manufacturers, can have different form factors and different UI designs. For that reason, we choose three devices that represent the different flavors of Symbian smartphones that are currently in the market (see Figure 3.1). The Nokia N95 is an S60 3rd Edition device that can be operated with one hand; the Nokia 5800 XpressMusic is an S60 5th Edition device with a resistive touch screen and tactile feedback; the Sony Ericsson W960i is a UIQ 3 device with a touch screen.



Figure 3.1 Smartphones: a) Nokia N95, b) Nokia 5800 XpressMusic and c) Sony Ericsson W960i

Tradition says that the first application should be the canonical ‘Hello World’ program. Since we assume that you are already familiar with MIDP development, we’re going to use a new introductory application that allows us both to run our first application and to answer a question that developers ask every time they get a new device: what JSRs are supported on the device?

The first application that we run on a Symbian smartphone is a utility to detect which APIs are supported. At the end of this section, not only will you be able to run your first application, but you will also have an extensible Java ME detection tool which is used in other chapters. We call it the Java ME Detectors suite.

3.1.1 Implementing the Java ME Detectors Suite

To develop the application we need an SDK and an IDE. NetBeans and Java ME SDK 3.0 are sufficient for creating your first MIDlet. That tells you that Java ME development for Symbian OS can be done using the standard Java ME tools and SDKs that you have always used. However, there are many other tools for you to benefit from, and they are discussed in Chapter 5.

Create a project named ‘Java ME Detectors’ in NetBeans. It will contain three classes: JSRsDetectorMIDlet, JSRsView and ApiInfo (source code and binaries can be downloaded from the website for this book, developer.symbian.com/javameonsymbianos). The main functionality of the application is to receive as input the name of a JSR and a main class from that JSR, and to detect if it is supported. The input can come from the JAD, JAR manifest file, properties file, or the UI.

The Mobile Services Architecture (MSA) defines system properties for each JSR that indicate whether the JSR is supported or not. For example, `wireless.messaging.version` returns the WMA version supported by the device. However, to make this utility portable for devices which are not compliant with MSA, we detect the JSR by trying to load a sample class.

ApiInfo encapsulates the information that is read as input: the name of the API and its main class, which indicates that the JSR is supported if it can be dynamically loaded:

```
// Encapsulates an API's name and main class
public class ApiInfo {
    String name;
    String mainClass;
}
```

JSRsView contains the core application task, which is to detect the list of APIs given as input. When the user presses Detect, an array of ApiInfo is requested from the MIDlet and, for each ApiInfo in the

array, the `JSRsView.detectAPIs()` method tries to load a Java class whose name is stored in the `ApiInfo.mainClass` member:

```
// The user view of the JSRs Detector MIDlet
public class JSRsView extends Form implements CommandListener {
    private Command exit = new Command("Exit", Command.EXIT, 0);
    private Command detect = new Command("Detect", Command.OK, 0);
    private JSRsDetectorMIDlet midlet;

    // CommandListener implementation
    public void commandAction(Command cmd, Displayable disp) {
        if (cmd == exit) {
            midlet.notifyDestroyed();
        } else if (cmd == detect) {
            detectAPIs();
        }
    }
    // Detect supported APIs
    private void detectAPIs() {
        resetUl();
        ApiInfo[] detectedApis = midlet.getApiInfoArray();
        for (int i = 0; i < detectedApis.length; i++) {
            try {
                Class.forName(detectedApis[i].mainClass);
                // class loaded successfully - API is supported
                append(detectedApis[i].name + ": Yes\n");
            } catch (ClassNotFoundException cnfe) {
                // class not loaded- API is NOT supported
                append(detectedApis[i].name + ": No\n");
            }
        }
    }
}
```

The default input to `JSRsDetectorMIDlet` can be read from the JAD file by calling the MIDlet's `getAppProperty()` method sequentially until `NULL` is returned. The following JAD properties are loaded and parsed by the MIDlet to create the array of `ApiInfo` objects:

- JSR1: JSR-139 CLDC 1.1, `java.lang.Float`
- JSR2: JSR-118 MIDP 2.0, `javax.microedition.lcdui.CustomItem`
- JSR3: JSR-75 PIM, `javax.microedition.pim.PIM`
- JSR4: JSR-75 FileConnection, `javax.microedition.io.file.FileConnection`
- JSR5: JSR-82 Bluetooth, `javax.bluetooth.LocalDevice`
- JSR6: JSR-135 MMAPI, `javax.microedition.media.Manager`
- JSR7: JSR-172 WS, `javax.xml.rpc.JAXRPCException`

- JSR8: JSR-120 WMA 1.0, `javax.wireless.messaging.MessageConnection`
- JSR9: JSR-177 SATSA-APDU, `javax.microedition.apdu.APDUConnection`
- JSR10: JSR-177 SATSA-JCRMI, `javax.microedition.jcrmi.JavaCardRMICConnection`
- JSR11: JSR-177 SATSA-PKI, `javax.microedition.security-service.CMSMessageSignatureService`
- JSR12: JSR-177 SATSA-CRYPTO, `javax.crypto.Cipher`
- JSR13: JSR-179 Location, `javax.microedition.location.Location`
- JSR14: JSR-180 SIP, `javax.microedition.sip.SipConnection`
- JSR15: JSR-184 3D, `javax.microedition.m3g.Graphics3D`
- JSR16: JSR-205 WMA 2.0, `javax.wireless.messaging.SizeExceededException`
- JSR17: JSR-211 CHAPI, `javax.microedition.content.ContentHandler`
- JSR18: JSR-226 SVG, `javax.microedition.m2g.SVGImage`
- JSR19: JSR-229 Payment, `javax.microedition.payment.TransactionModule`
- JSR20: JSR-234 AMMS, `javax.microedition.amms.GlobalManager`
- JSR21: JSR-238 i18n, `javax.microedition.global.ResourceManager`

After downloading Java ME Detectors from the Sun Developer Network, you can build and deploy it to a device.

3.1.2 Deployment over a Short Link

Symbian OS, as an open platform, serves developer needs well and serves as an excellent open Java ME development platform even if you are developing a mobile applications for platforms other than Symbian OS.

You may deploy your suite over the air by putting it on a web server. In reality, most developers prefer a quicker and simpler mechanism so beaming the suite JAR over a short-link connection is simpler and faster. For our first deployment, we do not use a device-manufacturer tool or SDK but only the reference devices and a laptop with Bluetooth.

After pairing the devices (Nokia N95 and Sony Ericsson W960i) with the laptop, you can send the JAR file for installation. From Windows File Explorer, go to the location of the suite binaries and drag the JAR to the appropriate file transfer service in My Bluetooth Places (see Figure 3.2). Alternatively, from Windows File Explorer, go to the location of the suite binaries, right click on the JAR file, and select Send to, Bluetooth. A submenu shows the paired devices and you can choose the correct device.



Figure 3.2 Installing a JAR through My Bluetooth Places on a Windows PC

When you send the JAR to the Nokia N95, it is stored in the Messaging Inbox (Figure 3.3a). When you send the JAR to the Sony Ericsson W960i, you are immediately presented with it (Figure 3.3b).

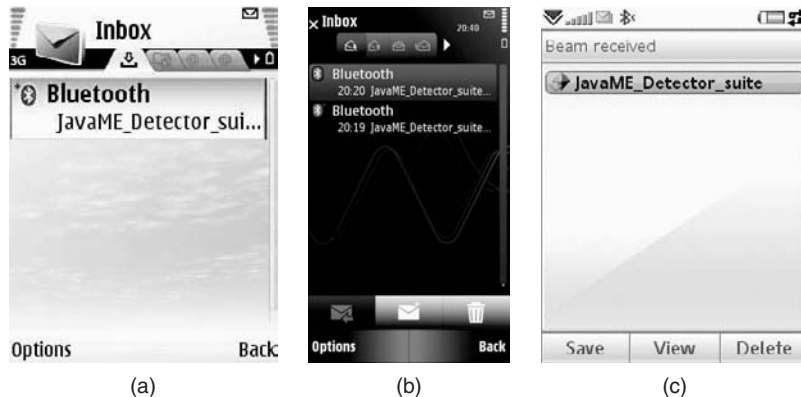


Figure 3.3 Java ME Detectors beamed to a) Nokia N95, b) Nokia 5800 XpressMusic and c) Sony Ericsson W960i

As you can see, how the device handles the suite is specific to that device. For example, the MIDlet may go into the Messaging Inbox or

the Application Management System (AMS) may be automatically or manually activated. In any case, the process itself is very simple and straightforward.

To complete the installation:

- on the Nokia N95, click all the way through until you are notified that the installation has completed
- on the Sony Ericsson W960i, press View and click all the way through until AMS launches the application.

3.1.3 Launching Java ME Detectors

On the majority of Java-enabled feature phones, you launch a Java application from a designated menu on which all installed Java applications reside. An important feature of Symbian OS is that Java applications are executed from the main applications menu, just like native applications. In this respect, Java applications are treated as ‘first-class citizens’ and are no different to any native application installed or running on the device.

Now that the Java installer has completed its task, let’s run JSRsDetectorMIDlet on our reference devices:

- On the Nokia N95, click on Main menu, Applications.
- On the Sony Ericsson W960i, from the home screen, press Menu, Tools.

Both devices present you with the main application menu from which all installed applications are launched (see Figure 3.4). Select the MIDlet and run it. Congratulations, you have just completed the task of running a MIDlet on a Symbian smartphone.



Figure 3.4 Main applications menu on a) Nokia N95, b) Nokia 5800 XpressMusic and c) Sony Ericsson W960i

Table 3.1 shows the results of running JSRsDetectorMIDlet. As you can see, all the reference devices support more JSRs than there are in the MSA subset. In fact, they are very close to supporting all MSA Component JSRs. These are not the only supported APIs; we discuss other APIs later in the chapter.

Table 3.1 Supported JSRs in Reference Phones

JSR	Nokia N95	Nokia 5800 XpressMusic	Sony Ericsson W960i
JSR-139 CLDC 1.1	✓	✓	✓
JSR-118 MIDP 2.0/2.1	2.0	2.1	2.0
JSR-75 PDA Packages	✓	✓	✓
JSR-82 Bluetooth and OBEX	✓	✓	✓
JSR-120 WMA 1.1	✓	✓	✓
JSR-135 Mobile Media API 1.1	✓	✓	✓
JSR-172 Web Services 1.0	✓	✓	✓
JSR-177 SATSA JCRMI, APDU, PKI, CRYPTO	-- ✓✓	-- ✓✓	–
JSR-179 Location	✓	✓	–
JSR-180 SIP	✓	–	–
JSR-184 Mobile 3D Graphics	✓	✓	✓
JSR-205 WMA 2.0	✓	✓	✓
JSR-211 CHAPI	–	–	–
JSR-226 SVG	✓	✓	✓
JSR-229 Payment	–	–	✓
JSR-234 AMMS	✓	✓	–
JSR-238 i18n	–	–	–

3.1.4 Summary

This is probably not the first time you have deployed a MIDlet suite onto a real device but you can clearly see how fast and easy it is to do on

a Symbian smartphone. Java ME development and deployment is easy because Symbian OS is an open platform which serves developers needs well. Later, we learn about additional ways to deploy, launch and manage Java ME applications, which will help you throughout your development.

We intentionally replaced the standard ‘Hello World’ program with the Java ME Detectors suite, which gives you a diagnosis tool to put into your personal development toolbox. You can always check on the web² to see the Java specification of a device but the Java ME Detectors suite gives you an extensible tool that refers specifically to the device you are holding in your hands.

3.2 Which APIs Are Supported?

There is no single answer to which JSRs are supported on Symbian smartphones. The list of detected JSRs in Table 3.1 represents the family of devices to which the three reference devices belong (respectively, S60 5th Edition, S60 3rd Edition FP1, and UIQ 3.0). As you have seen, the support provided by the Nokia N95, the Nokia 5800 XpressMusic, and the Sony Ericsson W960i exceeds the list of MSA Subset Component JSRs and is nearly at the level of full MSA Component JSRs. New devices are coming out all the time and so more JSRs may be supported.

For example, S60 3rd Edition FP2 and S60 5th Edition include additional APIs and UIQ 3.3 supports JSR-248 MSA Fullset.³ A Forum Nokia Wiki page lists the supported APIs for both Nokia S40 and S60 devices.⁴ However, you can assume that generally the direction of Java ME on Symbian OS is towards MSA Fullset and beyond.

In this section, we cover additional (non-JSR) APIs that are supported. But first we look at the advantage of having a common and consistent JSR implementation for a variety of devices and manufacturers.

3.2.1 Consistent Implementation

The major shortcoming of Java ME on feature phones is having inconsistent implementations. For example, to take a snapshot on different feature phones, you might need to call the same methods in a different order. You may also find different defects on different devices, differences in the interpretation of a JSR’s specification, and so on. Symbian OS provides a

² For Nokia, go to www.forum.nokia.com/devices.

³ developer.uiq.com/news_1252.html

⁴ wiki.forum.nokia.com/index.php/Java_ME_API_support_on_Nokia_devices

single common Java ME implementation which is the base for a variety of models from many manufacturers.

Although not a technical issue, it is important to understand the delivery chain in which Symbian OS plays a central role but is not the only actor. The Symbian ecosystem is not a single monolithic entity. At the time of writing, Symbian OS, including the Java ME subsystem, is shipped to phone manufacturers who can extend and customize the operating system. The Java ME platform is equipped with a large standard selection of Java ME APIs (see Chapter 10) when it reaches the phone manufacturer, who can then add additional JSRs of their own. This customization approach is what ensures one common platform for a large number of devices from many handset manufacturers, all of whom can still configure the platforms for their needs and brand for network operators.

The Java ME developer, at the end of the delivery chain, sees many devices which have much in common. There is still much diversity, so we dedicate Chapter 4 to considering how to deal with it. Luckily, having a common implementation on Symbian OS considerably reduces the fragmentation problems that plague other platforms.

Apart from JSR APIs, there are other APIs that are not standardized by the Java Community Process (JCP) but which can help you to implement a solid mobile application.

3.2.2 Nokia UI API

The Nokia UI API has been available since the days of MIDP 1.0 and devices such as the Nokia 6600. Back then, MIDP 1.0 was the standard for ensuring maximum portability and concentrated only on features that could be implemented by all mass-market devices. Some of those devices had very limited graphics capabilities and sometimes no sound capabilities at all. The Nokia UI API was introduced to make such features available to Java applications on Nokia devices, which already had sound capabilities and better graphics capabilities.

Although the API is named after Nokia, it is also available on phones that belong to other manufacturers. Because of its success among MIDP 1.0 developers, other phone manufacturers (including Sony Ericsson) made the API available on their platforms.

The Nokia UI API consists of a small number of classes defined in the proprietary `com.nokia.mid.ui` and `com.nokia.mid.sound` packages. They add the following features:

- control of vibration and the LCD backlight
- a basic sound API for playing tones and digitized audio
- the ability to draw and fill polygons and triangles

- rotation and flipping of images
- alpha-channel color support
- low-level access to image pixel data.

At the time of writing, the Nokia UI API is still available on many SDKs and devices from different manufacturers. For example, it is available on the Nokia S60 5th Edition SDK; on S60 3rd Edition FP2 SDK and devices such as the Nokia N95; and on the Sony Ericsson SJP-3 SDK and devices such as the UIQ 3 Sony Ericsson W960i.

However, MIDP 2.0 includes almost all the Nokia UI API's features and since then the Nokia UI API has been a deprecated API which remains available mostly to maintain backward compatibility. It is strongly recommended that developers do not develop using it, but use standard APIs, such as MIDP 2.0 LCDUI, to guarantee that their MIDlets work in future devices.

3.2.3 Embedded Standard Widget Toolkit

From S60 3rd Edition FP2, Nokia introduced support for the Embedded Standard Widget Toolkit (eSWT) which is an alternative UI toolkit to LCDUI. eSWT is a subset of the Standard Widget Toolkit (SWT) API, which was jointly developed by IBM and Nokia as part of the Eclipse Foundation eRCP project.⁵ The SWT was developed as an alternative to the AWT and Swing desktop UI toolkits included in Java SE. For example, the Eclipse IDE uses SWT for its user interface.

It is very easy for SWT developers who are already experienced with Java ME development to use eSWT, since eSWT shares the core components, design, idioms and development principles of SWT. The core features of eSWT are:

- native platform look and feel
- high performance and minimal overhead
- customizable UI components
- low-level graphics support.

eSWT is divided into three components (see Figure 3.5): Core eSWT, Expanded eSWT and SWT Mobile Extensions. Core eSWT includes the basic SWT classes and a set of the most commonly used widgets, such as buttons and text fields, low-level graphics APIs and layout managers, and defines the event model. More advanced widgets, such as tables, trees, dialogs and a powerful browser control are part of the Expanded eSWT component. SWT Mobile Extensions includes widgets that are specific

⁵ www.eclipse.org/ercp

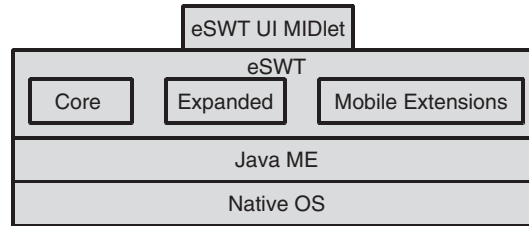


Figure 3.5 Main components of eSWT

to mobile devices, such as commands or full-screen shells. The eSWT widgets gallery⁶ presents the range of eSWT widgets.

The eSWT APIs provide:

- visually compelling and rich user interface components
- flexible components layout using layout managers
- fine-grained support for user interface events.

We show two examples for eSWT: the first is a very basic example and the second demonstrates one of the more powerful features of eSWT – embedding a browser in your application.

Basic Use of eSWT

Figure 3.6 shows a simple ‘Hello World’ application written using eSWT. The basic building blocks of every eSWT MIDlet are `org.eclipse.swt.widgets.Display`, `Shell`, and other widgets. Every eSWT application requires a single display object and one or more shells. The



Figure 3.6 ‘Hello World’ example using eSWT

⁶ www.eclipse.org/ercp/eswt/gallery/gallery.php

`org.eclipse.swt.widgets.Display` singleton can contain one or more shells which can contain other shells and other types of widget.

```
import org.eclipse.ercp.swt.mobile.*;
import org.eclipse.swt.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
public class eSWTExamplet extends MIDlet implements Runnable,
                                   SelectionListener, PaintListener {

    private Display display;
    private Shell shell;
    private Thread eSWTUIThread;
    private boolean exit = false;
    public void startApp() {
        // start a Thread in which eSWT will execute its event loop
        if (eSWTUIThread == null) {
            eSWTUIThread = new Thread(this);
            eSWTUIThread.start();
        }
    }
    ...
    public void run() {
        // init eSWT core objects
        display = new Display();
        shell = new Shell(display);
        shell.open();
        // init rendering of the "Hello World"
        shell.addPaintListener(this);
        shell.redraw();
        // allow eSWT to execute the event loop
        while (!exit) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }
        // perform cleanup required for eSWT objects
        display.dispose();
        notifyDestroyed();
    }
    ...
    public void paintControl(PaintEvent e) {
        e.gc.setForeground(new Color(display, 255, 0, 0));
        e.gc.drawText("Hello eSWT!", 0, 0, SWT.DRAW_TRANSPARENT);
    }
}
```

As you can see in the example code, eSWT MIDlets implement the MIDlet lifecycle methods just like any other MIDP MIDlets. When the MIDlet enters the active state for the first time, it creates the eSWT application UI thread, which constructs a single `org.eclipse.swt.widgets.Display` instance that is not disposed of until the MIDlet enters the destroyed state. Before exiting, the eSWT widgets and objects must generally be manually deallocated by calling the `dispose()` method.

Advanced Use of eSWT

To demonstrate a highly powerful feature of eSWT, we use `org.eclipse.swt.browser.Browser` embedded into an application (see Figure 3.7).



Figure 3.7 eSWT browser

In the following example code, the user can set a URL and navigate back and forth between visited web pages. (Using `MIDlet.platformRequest()` does not allow you to programmatically navigate between different pages; the eSWT Browser API does let you do it.)

```
import org.eclipse.swt.browser.Browser;
public class BrowserMIDlet extends MIDlet implements SelectionListener,
                                                    Runnable {
    ...
    // UI Thread run() method
    public void run() {
        ...
        browser = new Browser(shell, SWT.NONE);
        browser.setUrl("http://forumnokia.mobi");
        shell.open();

        while (!exiting) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }

        browser.dispose();
        shell.dispose();
        display.dispose();
        notifyDestroyed();
    }
}
```

```

public void widgetSelected(SelectionEvent evt) {
    if (evt.widget.equals(urlCommand)) {
        QueryDialog urlDialog =
            new QueryDialog(shell, SWT.NONE, QueryDialog.STANDARD);
        urlDialog.setPromptText("URL:", "http://");
        browser.setUrl(urlDialog.open());
        shell.redraw();
    }
    // handle "Back"
    else if (evt.widget.equals(backCommand)) {
        if (browser.isBackEnabled()) {
            browser.back();
        }
    }
    // handle "Forward"
    else if (evt.widget.equals(forwardCommand)) {
        if (browser.isForwardEnabled()) {
            browser.forward();
        }
    }
}
...

```

When to Use eSWT

There are some considerations to be taken into account when choosing to use eSWT instead of LCDUI. MIDP LCDUI has not been updated with any significant improvements since it was designed, because it must be available on low-end devices. For that reason, LCDUI does not provide much support for the user experience of high-end devices, such as Symbian smartphones.

On the other hand, although SWT was designed as a cross-platform GUI toolkit and the embedded version aims to reach more mobile platforms, eSWT is not defined by the JCP and remains a non-standard API that is not supported by every mobile platform.

In summary, eSWT is a powerful option on Symbian smartphones, which provides a rich and compelling set of UI components that can be used as an alternative to MIDP LCDUI, provided that the Java ME implementation supports it.

3.2.4 IAPInfo API

The IAPInfo API, which was introduced in S60 3rd Edition FP2, enables Java applications to access information, such as:

- Internet Access Points (IAP) information, such as ID, name, bearer and service type
- Destination Network information, such as ID, name and associated access points

- Preferred access points
- Last-used access point.

The following code snippet demonstrates simple usage of the IAP information:

```
// get the singleton IAPInfo object
IAPInfo iapInfo = IAPInfo.getIAPInfo();
// get the available Destination Networks and the list of IAPs
DestinationNetwork destNetworks[] = iapInfo.getDestinationNetworks();
if(destNetworks.length > 0) {
    AccessPoint accessPoints[] = destNetworks [0].getAccessPoints();
}
// get the last used IAP
AccessPoint lastUsedIAP = iapInfo.getLastUsedAccessPoint();
```

The Generic Connection Framework (GCF) of Java ME does not have a notion of choosing between Internet Access Points and the default behavior is to use the system default IAP. The IAP-Control feature of the IAPInfo API allows code to explicitly specify the IAP that should be used when creating a new GCF network. This is done by extending the GCF URI scheme definition with two optional parameters: `nokia_apnid` and `nokia_netid`. For example, this short code snippet opens a new connection, using the IAP whose system ID is 2:

```
Connector.open("http://www.nokia.com;nokia_apnid=2")
```

The parameters are removed from the URL before opening the connection. They are used internally without being sent to a web server.

`AccessPoint.getURL(String aURL)` is a convenience utility method which returns a new URL string with the optional IAP ID parameter added. For example, the following snippet of code opens the new connection with the IAP encapsulated by `accessPoint`:

```
AccessPoint accessPoint;
...
Connection.open(accessPoint.getURL("http://www.nokia.com"));
```

IAPs for Java applications on Symbian OS can also be set non-programmatically. For more information, please refer to Section 3.7.2.

3.2.5 SNAP Mobile

The Scalable Network Application Package (SNAP) is Nokia's end-to-end solution for developing connected Java games and creating mobile

games communities. SNAP Mobile⁷ includes a lightweight, client-side Java library (the SNAP Mobile Client API) that communicates over HTTP or TCP with a remote backend system called the 'SNAP gateway'.

The SNAP Mobile Client API enables creation of connected games with features such as account creation and login, presence, instant messaging, score tables and rankings, and multiplayer game playing.

The SNAP gateway includes a Session Manager and a Virtual Client system that handle requests from SNAP Mobile clients and responses from SNAP community services, such as web services, the Instant Messaging and Presence Service (IMPS), and SNAP Game services.

Once your game successfully completes the SNAP Mobile Compliance Test, you can take it to market via a number of Nokia and third-party channels. For more information, please refer to Appendix B.

3.2.6 Retrieving Telephony Information

From S60 3rd edition FP2 onwards, there are additional system properties that provide access to telephony information. The information provided is minimal but it is meant to serve telephony services to applications that are not themselves telephony-oriented.

For example, consider an application that requires locking onto a specific device. Developers have to make their applications aware of the unique IMEI of the phone and provide a mechanism to lock their applications to it. The scenario in such a case could be:

1. The user sends the phone IMEI when they purchase the application.
2. The IMEI is used to bind the application to the device.
3. The application verifies the phone IMEI at first launch.

There is no Java API that provides access to telephony information. The information is available only via the following system properties:

- `com.nokia.mid.imei`: International Mobile Equipment Identity (IMEI) number
- `com.nokia.mid.imsi`: International Mobile Subscriber Identity (IMSI) number
- `com.nokia.mid.networkid`: network identification parameters
- `com.nokia.mid.networksignal`: current (GSM/CDMA) network signal strength
- `com.nokia.mid.networkavailability`: network availability

⁷ www.forum.nokia.com/snapmobile

- `com.nokia.mid.batterylevel`: percentage of battery level
- `com.nokia.mid.countrycode`: current network country code.

3.2.7 Summary

Symbian OS provides a wide, common Java ME platform for a plethora of devices by various manufacturers, which minimizes JSR support differences between different families of devices. The rich selection of JSRs allows you to create rich applications using standard Java APIs only.

Nevertheless, additional APIs are added to the Java stack by vendors in order to enrich their platform. You may see these APIs as additional opportunities or you may choose to stay on the common and generic ground of using only JCP-standardized JSRs. In any case, Java ME on Symbian OS gives you the power of choice.

3.3 Proprietary JAD Attributes

There are various edge cases that are not handled by Java ME specifications but are handled by S60-specific JAD attributes (see Table 3.2 for details of some attributes).

Support for these attributes may vary between S60 devices. For example, S60 5th Edition introduced a few JAD attributes that support screen orientation and scaling (for more information, please see Chapter 4). These JAD attributes are also supported on S60 3rd Edition

Table 3.2 Proprietary S60 JAD Attributes

Attribute name	Since S60 Edition	Description
Nokia-MIDlet-On-Screen-Keypad	5th Edition	Offers backwards compatibility for MIDlets, that use Canvas and are not originally made for touch devices
Nokia-Scalable-Icon	3rd Edition FP2	Specifies scalable MIDlet icon support
Nokia-MIDlet-Background-Event	3rd Edition FP2	Specifies MIDlet lifecycle methods invocation when switching between background and foreground
Nokia-MIDlet-Category	All editions	Specifies if the MIDlet is an application or game
Nokia-MIDlet-No-Exit	3rd Edition FP2	When pressing the End key, the MIDlet will go to the background instead of terminating

FP2 devices released after S60 5th Edition. Earlier S60 3rd Edition FP2 devices do not support those attributes.

Developers can take benefit of other proprietary JAD attributes after verifying that they are supported on the target device. For more information and the full list of proprietary JAD attributes, please refer to Forum Nokia and the online Java Developer Library.⁸

3.4 Computing Capabilities of Java ME on Symbian OS

Unlike many feature phones, Java ME on Symbian OS provides Java applications with an execution environment that has no fixed limit on Java computing resources. Developers can focus on the main task of their application rather than on managing resource utilization and working around various computing constraints.

If you have ever ported an MIDP application from one feature phone to another, you probably understand what we mean and still remember the pain. If a certain JSR is not supported, you cannot target a certain device using Java ME but constraints on computing resources are a different story. They make you do things that are not necessary in order to get you closer to your goal of the application's main use cases.

From a software engineering point of view, to build an application, you need to create an abstraction of the problem domain as close as possible to real life. Then you apply relations between the various entities and lay down the execution path that enables each of the application's use cases. But when you have computing constraints, you find yourself resorting to workarounds and tricks to reduce resource utilization, such as manipulating the JAR size, the RMS size, the number of threads, the number of connections, and so on. On low-end devices, computing constraints can be a real nuisance. Not so with Java ME on Symbian OS. In general, Java ME on Symbian OS poses no limits on computing resources.

3.4.1 Constraints on JAR Size

There is no predefined limit on the size of the MIDlet suite's JAR file. Even an extremely large Java application (e.g., 700 KB) will be installed successfully on a Symbian smartphone.

There are low-end phones and feature phones (not Symbian smartphones) which limit the JAR size to 120 KB. As a developer, you find yourself checking every few builds to ensure you have not exceeded the 120 KB limit. So you start resorting to various techniques that restrict your application from growing in size. For example, each class has an overhead, so developers may avoid defining Java interfaces or abstract

⁸ www.forum.nokia.com/document/Java_ME_Developers_Library

classes, possibly squeezing more code into fewer classes, and so on. These techniques have a negative impact on the abstraction of the problem domain and can make the code less readable, less maintainable and less extendable.

Additionally, images and other resources in the JAR demand a high price when there are fixed limits, so you resort to converting resources to smaller formats but these probably also have an impact on quality. Reducing the size of images by using PNG-8 instead of PNG-32 makes the images smaller but may also reduce their quality. For example, the background image for a Java game would not be as visually compelling as it could be.

These constraints don't apply when you use Java ME on Symbian OS. There's no need to think twice before adding another class to improve your abstraction. An additional class, however big it is, does not pose any problem to the installation or run-time environment. You can code freely and safely without counting the number of classes and resorting to code-size-reduction tricks. The same goes for static resources, such as images – naturally, you need to consider the footprint (and if the format you choose is supported), but not having a JAR size limit means that you can be more relaxed and put much more emphasis on the quality of the images rather than how big they are.

A traditional solution for reducing the JAR size by 10–20% is to use an obfuscator. While obfuscators achieve a substantial reduction of code size, you need to remember that the primary target of obfuscation is to protect your intellectual property by preventing decompilation and reverse engineering of your code.

The lack of constraint does not mean that JAR size is not relevant – a sizeable JAR takes longer for the user to download and the installed application inevitably occupies significant amounts of memory. But you can have a much more relaxed approach on Symbian smartphones and give higher priority to abstraction, maintainability, quality, and so on.

3.4.2 Constraints on Heap Size

There is no predefined limit on the application heap size or on the number and stack size of Java threads. Of course, eventually you will run out of memory but you need to do something quite intense and memory consuming to get to that point. The Java Objects heap is allocated a certain size on application start up. If there is not enough memory to allocate a new object, after a few memory allocations and garbage collections, the CLDC-HI VM expands the application's Java heap to allow more memory allocations. The process is reversible: the heap can shrink, in order to free resources back to the system.

This does not mean that there is no need to plan proper memory management for your Java application – dereferencing unused Java objects

and closing resources remains valid and important, just as on any other Java ME platform.

3.4.3 Constraints on Threads

There is no fixed limit to the number of threads you can create or run. As always, you need to be considerate of resource utilization, but another Java thread is unlikely to make a significant difference. On some low-end platforms, you might have a small pool of Java threads that you would use repeatedly for different tasks. On Symbian OS, your code can have Java threads according to its needs, without the burden of thread pool management.

Of course, this does not mean that you should not consider cooperative multithreading in your application. But if a thread is what you need for the proper design or task execution in your application, then you should use it.

3.4.4 Constraints on Number of Connections

Sockets and other connections are also a scarce resource on low-end platforms. For that reason, some devices impose a predefined limit on the number of connections that an application can open. For example, a Java application on such a platform may only be able to open two connections at a time, which might not be enough for some applications. Again, Java ME on Symbian OS lets you open as many connections as you need for your application.

This does not mean you should not dispose of connections that are no longer needed. For example, once an application completes a web services transaction and the connection is no longer needed, you need to dispose of the connection in order to release all acquired resources.

3.5 Java ME: Exposing the Power of Symbian OS

Because Symbian OS is such a powerful host, it has some capabilities that are not common on the majority of feature-phone Java ME platforms. Symbian OS is a powerful native platform and Java ME exposes that power through standard Java ME APIs. The platform has to support both the more advanced JSRs and their optional features. Symbian OS v9 and later versions support many JSRs. When looking for an advanced and powerful JSR (for example, JSR-184 3D graphics or JSR-226 SVG), you can assume that it is supported.

For example, `javax.microedition.io.SocketConnection` is optional in MIDP 2.0 but has been supported in Symbian OS for a long time, even in older smartphones. `javax.microedition.io.`

`ServerSocketConnection` is still optional in MSA 1.1 but is supported on all Symbian smartphones.

The MIDP 2.0 specification does not define which protocols and services must be supported by the Push Registry, yet the support of Java ME on Symbian OS protocols exceeds the MSA 1.1 mandated list; for example, the Push Registry supports TCP server sockets.

For a developer, a powerful common Java ME implementation for many devices improves the predictability of advanced features.

We do not list all of the optional features for all Symbian smartphones because such a list would be long, quite tedious and of questionable value. We are not attempting to provide you with answers for every possible question; we're trying to provide you with the right questions and the right techniques or advice to answer the questions.

3.5.1 Multitasking Between Applications

One of the powerful features of Symbian OS is that there can be more than one application running at the same time and the user (or Symbian OS itself) may switch between applications and move them between the foreground and the background. For example, the user can start a Java application, switch to the applications menu and start another native application and possibly a third Java or native application as well. Since Symbian OS is a multitasking operating system, all three applications can continue running concurrently without being terminated. The AMS can also move an application to the background, for example, when the device receives a phone call. This is a major difference from low-end platforms, in which switching from the current application to another terminates the former application.

Your Java application needs to be designed with moving between the background and foreground in mind. It must be 'a good citizen'; while it is in the background, it must allow the foreground application to acquire the resources it needs. To ensure this, adhere to the MIDlet lifecycle guidelines:

- When your application is sent to the background, the `MIDlet.pauseApp()` method is called and the MIDlet should release temporary resources and become passive (e.g., it should close open connections and pause rendering application threads).
- When your application is brought back to the foreground, the `MIDlet.startApp()` method is called and the MIDlet can acquire any resources it needs and resume (e.g., it should resume application threads that handle the UI rendering).

The following snippet of code illustrates an implementation of `MIDlet.startApp()` and `MIDlet.pauseApp()`:

```
public void startApp() {
    if (!started) {
        // MIDlet started first time
        workerThread = new WorkerThread();
        started = true;
    }
    // MIDlet brought to foreground
    paused = false;
    workerThread.resumeWork();
}
public void pauseApp() {
    // MIDlet sent to background
    paused = true;
    workerThread.pauseWork();
}
```

In reality, things might work slightly differently so Section 4.5 discusses how to handle the state switches across different Symbian smartphones. For now, note the importance of releasing unused resources and suspending certain operations while running in the background.

Your application should not consume resources, such as CPU, for operations that are not necessary while in the background (e.g., rendering threads) but it is still alive and running. Depending on the core task of your application, it may be possible to fulfill some of those tasks even when running in the background. For example, if your application is registered to receive events, such as incoming SMS messages or location notifications, the registration for receiving events does not need to be removed every time the MIDlet is sent to the background.

To illustrate such a case, the following example implements WMA `MessageListener`, which queues or displays an incoming message according to the state of the application. If the application is in the Paused state, then only minimal processing is done. If the application is Active, then it should operate with full functionality enabled.

```
public void notifyIncomingMessage(MessageConnection conn) {
    if (paused) {
        // TODO: silently add incoming message to queue, to be handled later
    } else {
        // TODO: display incoming message to user or perform another action
    }
}
```

3.5.2 Multitasking of MIDlets in the Same Suite

Let's see how two MIDlets from the same suite, both running at the same time, can not only access the same classes in the JAR file but also invoke methods of objects owned by the other running MIDlet. All MIDlets from

the same suite run in the same space. A new Java ME run-time process is spawned for every suite and all MIDlets from that suite run within that process and in the same Java memory space; this implies that MIDlets from the same suite can communicate, programmatically.

For example, MIDlet1 and MIDlet2 are both in the same suite:

```
public class MIDlet2 extends MIDlet implements CommandListener {
    public static int value = 0;
    public static MIDlet2 instance;
    public Object lock = new Object();
    private Form form = new Form("MIDlet2");
    private Command okCmd = new Command("Show Value", Command.OK, 0);
    public MIDlet2() {
        form.addCommand(new Command("Exit", Command.EXIT, 0));
        form.addCommand(okCmd);
        form.setCommandListener(this);
        instance = this;
    }
    public void commandAction(Command cmd, Displayable arg1) {
        if (cmd.getCommandType() == Command.EXIT) {
            notifyDestroyed();
        }
        if (cmd == okCmd) {
            form.append("\nvalue=" + value);
        }
    }
}
```

As you can see, when you press the Show value button, MIDlet2 prints the integer member value. So far, not very exciting.

Because MIDlet1 and MIDlet2 run in the same memory space, MIDlet1 can change MIDlet2.value from its CommandListener.commandAction() implementation:

```
public void commandAction(Command cmd, Displayable disp) {
    if (cmd.getCommandType() == Command.EXIT) {
        notifyDestroyed();
    }
    if (cmd == incCmd) {
        // change MIDlet2.value member, from MIDlet1
        MIDlet2.value++;
    }
}
```

This is quite exciting but we can do something even more entertaining and invoke methods in MIDlet2 from MIDlet1. For example, let's add to MIDlet2 that, when a command is pressed, it waits on a thread and a method that resumes the waiting thread:

```

public void wakeUp() {
    try {
        synchronized (lock) {
            lock.notify();
        }
    }
    catch (Exception e) {
        form.append("err: " + e);
    }
}

private void waitForNotify() {
    new Thread() {
        public void run() {
            try {
                form.append("\nzzz...");
                synchronized (lock) {
                    lock.wait();
                }
                form.append("\noy!");
            }
            catch (Exception e) {
                form.append("err: " + e);
            }
        }
    }.start();
}

```

MIDlet2 invokes `waitForNotify()` when we press a new button:

```

if (cmd == waitCmd) {
    this.waitForNotify();
}

```

And now comes the fun part – we let MIDlet1 invoke MIDlet2.
`wakeUp()`:

```

if (cmd == notifyCmd) {
    MIDlet2.instance.wakeUp();
}

```

Let's run the example. We start both MIDlet1 and MIDlet2. In MIDlet2, we press Wait and now there is a thread in MIDlet2 which is waiting. We switch back to MIDlet1 and press Wake up. When we move back to MIDlet2, we see the screen in Figure 3.8. During all that time, there can be other applications, Java or native, running in the background (see Figure 3.9).

Multiple applications can run concurrently and multiple applications from the same suite even run in the same space. There is power here that you can use but which should be handled with care.



Figure 3.8 MIDlet2 output

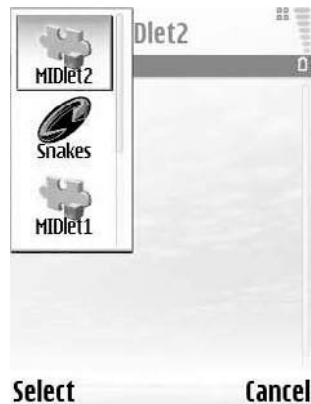


Figure 3.9 Multiple applications running concurrently

3.5.3 MIDP Applications and Symbian Signed

One of your friends or colleagues may ask you, "Have you passed your application through Symbian Signed?" The question might confuse you (or, possibly, cause a massive panic if you are about to release an application to a very demanding customer).

Symbian Signed is a program run by Symbian for and on behalf of the wireless community to digitally sign applications which meet certain industry agreed criteria. If a native application is Symbian Signed, it has successfully met the criteria defined in the Symbian Signed Test Criteria.

However, the Symbian Signed program is not relevant for MIDP applications. MIDP defines its own security policy that applies and usage of restricted APIs and functions requires permissions granted either by signing the suite or explicitly by the user. MIDP applications are considered safe and secure from the platform perspective and there is no reason to impose an additional security mechanism on MIDP applications. MIDP applications are, therefore, exempt from passing Symbian Signed.⁹

3.6 Tools for Java ME on Symbian OS

There are excellent Java ME SDKs for Symbian smartphones available for download on the web (see Chapter 5). The approach which we recommend is to become familiar with several SDKs and tools, and use the one which is most appropriate for the family of target Symbian smartphones, your development stage and the problem you are trying to solve.

At the early stages of development, you would probably use Symbian OS Java ME SDK in conjunction with a general-purpose SDK such as the Java ME SDK 3.0. As you progress in product development, you would gradually move more towards using a Java ME SDK for specific Symbian OS UI platforms.

At some point, as tends to happen, you'll probably encounter a certain challenge that requires an additional tool. You may find an appropriate tool in one of those SDKs specific to Symbian OS. For example, on-device debugging will assist you in identifying a problem that occurs on a device but is not reproducible on the emulator.

The option of such a powerful (and liberating) approach to tools is unique to Java ME on Symbian OS, which is rich in Java ME development tools. That is enough about SDKs for this chapter. Please refer to Chapter 5 and ensure you are aware of the variety of Java ME SDKs for Symbian OS.

3.7 Java ME Management on Devices

Java ME on Symbian OS is tightly integrated with the native platform services and is not managed as a separate entity; MIDlets are launched from the main menu like any other application (and not from a separate Java application submenu). At some point you may want to manage certain aspects related to Java ME in general or to a specific application; for example, you may want to change security settings, or install or remove a MIDlet suite. On Symbian smartphones, Java management operations are found in the native system management services.

⁹ For more information about Symbian Signed, please refer to www.symbiansigned.com. A few minutes can be well spent scrutinizing the Symbian Signed Test Criteria. If they are applicable to MIDP and your application, you can apply them voluntarily.

The locations or management capabilities of the following management services may differ depending on the Symbian OS version, UI platform or device manufacturer. However, it should be easy for you to spot those differences and find your way through.

3.7.1 MIDP Security Settings

During development, you may want be able to view and change the security settings of your tested MIDlets, for example, to see which certificates are deployed on the device. If your tested MIDlet uses the Push Registry and is in the untrusted domain, you might want to change the security settings so that it asks the user for permission only at the first launch.

We now explore the security settings on Symbian smartphones through the reference devices introduced in Section 3.1. For example, on the Nokia N95, select Menu, Tools, Settings, Security, Certificate management (see Figure 3.10). To view details of a certificate, such as its validity period and fingerprint, scroll to the certificate and press the middle key.



Figure 3.10 Trust roots on a) Nokia N95 and b) Nokia 5800 XpressMusic

To view and change the security settings for a MIDlet suite on the Nokia N95, open the Application Manager (Main Menu, Tools, App. mgr.) as shown in Figure 3.11.

After opening the Application Manager, scroll down to the suite name, press the middle key and select Open to see the security settings for each of the function groups (see Figure 3.12).

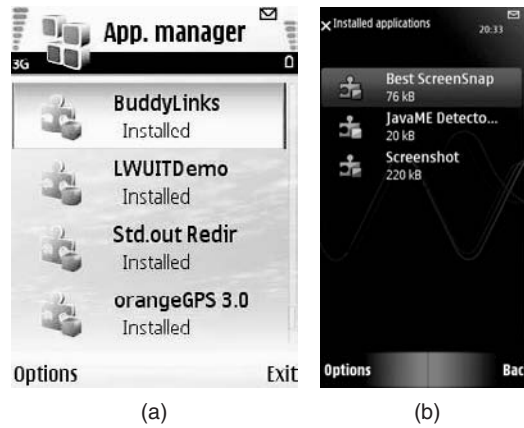


Figure 3.11 List of installed applications on a) Nokia N95 and b) Nokia 5800 XpressMusic

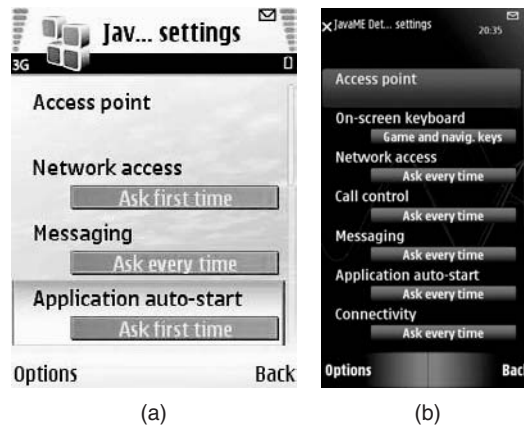


Figure 3.12 Suite security settings on a) Nokia N95 and b) Nokia 5800 XpressMusic

On S60 3rd Edition FP2 devices (and later), a new kind of run-time security prompt was introduced. It allows users to directly change the security setting for the application while it is running.

On the Sony Ericsson W960i, MIDP security settings are found at the same location on the device as all system-wide security settings. Go to Main menu, Tools, Control panel, Security and you will see the device security settings options (see Figure 3.13).



Figure 3.13 Sony Ericsson W960i security settings panel

Tap Certificate manager to view the system-wide digital certificates. Back at the top-level Security panel, tap Java certificates to view the digital certificates according to the security domains manufacturer, operator and third party (see Figure 3.14). Java certificates cannot be added or deleted, but you can disable or enable the existing certificates.

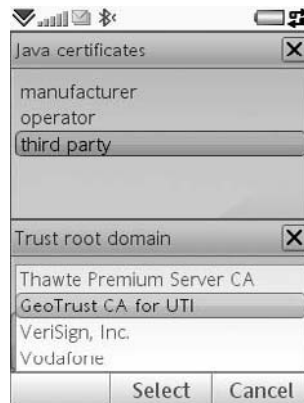


Figure 3.14 Third-party trust roots on Sony Ericsson W960i

At the top-level Security panel, tap MIDlet accounts to view all installed MIDlet suites (see Figure 3.15a) and MIDlet permissions to view and set the security permissions of the installed MIDlet suites. This is where you can set, for example, the messaging user permission to one of the available security options according to the Java trust domain (e.g., Always ask or Do not allow for an untrusted MIDlet suite).

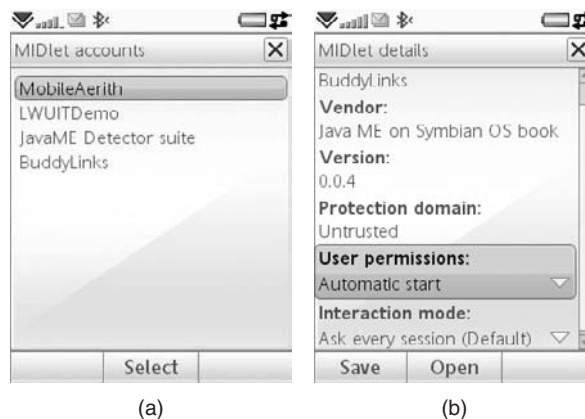


Figure 3.15 a) MIDlet accounts and b) user permissions on Sony Ericsson W960i

3.7.2 Controlling the Java Application's IAP

In Symbian OS, Internet Access Points (IAP) define the variable configuration factors that determine how an Internet connection is established. These factors can include the bearer (e.g., CDMA, GSM or GPRS), the dial-in number, and network login names and passwords. All IAPs are stored in the native Symbian OS communication database (CommsDat), which also stores the preferred order in which the IAPs should be used.

On many other Java ME platforms, the underlying implementation always uses the system default IAP configured for the browser. On Symbian smartphones, there are more flexible mechanisms which enable a specific IAP to be used for each Java application. For example, while the native browser may be configured to use one IAP, a Java application can use a different IAP.

On Nokia S60 devices, the phone user can set the connection settings specific to the Java application from an Application Manager menu item. For example, on the Nokia N95, if you configure a default IAP for a particular Java application, the application does not prompt the user for the IAP but automatically connects to the network. In the Application Manager, scroll down to the suite name, press the middle key and press Open. Select the Access point list item and press the middle key to select from the list of IAPs (see Figure 3.16).

On the Sony Ericsson W960i, you set the IAP to be used in the following way (see Figure 3.17):

1. Go to Main menu, Tools, Control Panel, Security, MIDlet accounts.
2. Select a MIDlet.
3. Pick an IAP from the drop-down menu in the Internet account field (the default is marked as System preferred).

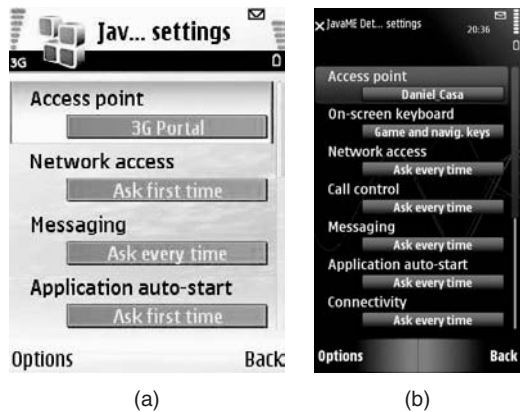


Figure 3.16 Changing the suite default IAP on a) Nokia N95 and b) Nokia 5800 Xpress-Music



Figure 3.17 IAP settings for a MIDlet suite on Sony Ericsson W960i

3.7.3 Installation and Removal of MIDlet Suites

In the Nokia N95 Application Manager, you can install a Java application that has been transferred to your device or remove previously installed suites. From the Application Manager menu, select Install or Remove (see Figure 3.18).

On the Sony Ericsson W960i, go to Main menu, Tools, Control panel, Other, where you will find Install and Uninstall menu items (see Figure 3.19). Tap on the Uninstall item to get the list of all installed applications, including MIDlet suites, on your device.

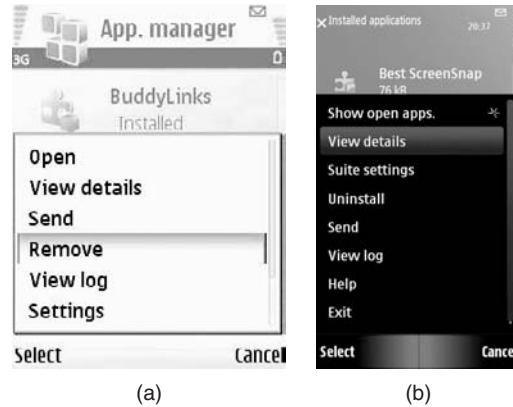


Figure 3.18 Removing an installed suite from a) Nokia N95 and b) Nokia 5800 Xpress-Music

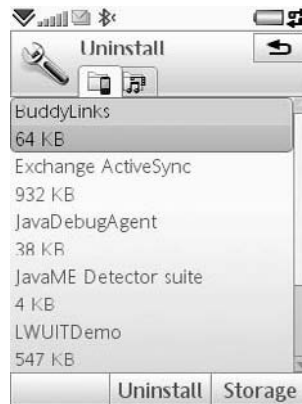


Figure 3.19 Removing an installed suite from Sony Ericsson W960i

3.7.4 Task Manager

The Task Manager helps you switch between applications and could save you going back and forth between different locations on the device. Leaving an application via Task Manager, rather than closing it, lets you return to the same view when you switch back to the application. This is useful when switching between concurrently running applications.

The Nokia N95 has a short cut to the Task Manager that allows you to quickly and easily view which applications are running on the phone. To open Task Manager hold down the main menu button (to the left of the directional keys) until the list of running applications appears at the top left of the screen (see Figure 3.20). You can either switch to a selected application by pressing on it or terminate it by pressing the C key.

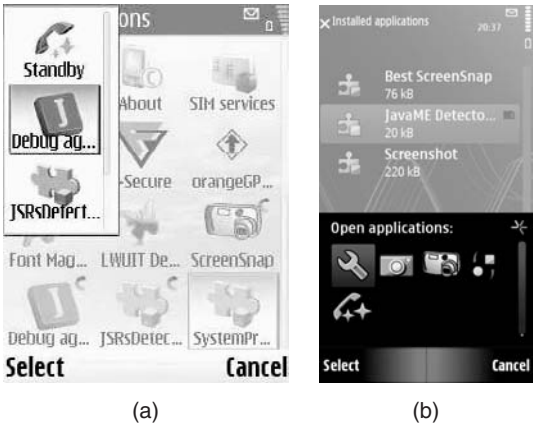


Figure 3.20 Task Manager on a) Nokia N95 and b) Nokia 5800 XpressMusic

To open the Sony Ericsson W960i Task Manager, tap in the device status bar (top right of the screen) or select from the main menu More, Task manager. You will be presented with Recent and Open tabs.

From the Recent tab, you can launch recently used applications. To switch to an application in the list, tap it or highlight it and select Switch. The Open tab (Figure 3.21) contains a list of all currently running applications and how much run-time memory they consume (it also lists applications that are closed but still reserve memory). Highlight an application and select End to terminate it and free up all memory used. From the More menu, you can sort the applications in the list by time, name or reserved memory (select the size menu item).

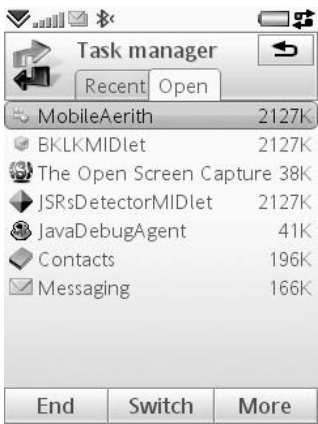


Figure 3.21 Task Manager on Sony Ericsson W960i

3.8 Crossing to the Native Land of Symbian OS

We now deal with areas that are part of the native capabilities of Symbian OS. We first give advice on how to better understand some of the messages that appear in Java exceptions. Then we discuss how some mobile applications can use both Java and native capabilities.

3.8.1 Native Error Codes in Java Exception Messages

Error handling in Java ME on Symbian OS remains the same as error handling anywhere else in Java ME; it is identical to standard Java error handling. The platform implementation informs you of errors through the usual mechanism of throwing Java exceptions as specified in the Java Language Specification. That does not change; what changes is what appears in the error message.

The thrown exception carries a description, which should provide the developer with sufficient information about the problem, e.g., ‘Connection not found’. As we discuss in Chapter 11, Java ME on Symbian OS is tightly integrated on top of the native Symbian OS APIs, therefore an error which occurs deep down in the native domain eventually triggers a Java exception to be thrown back in the Java ME run-time environment. In some cases, the Java exception error message includes the native error code which has triggered the Java exception. For example, your Java code might catch an `IOException` with a description that includes a native error code such as `-36`, which means that a local IPC session was disconnected (and you might want to restart the device). Naturally, at run time your application should treat the thrown Java exception in the same way as on any other Java ME platform. However, during development, if a native error code is specified, it could provide you with some additional information that might help you to diagnose the error and track down the reason it occurred.

In most cases, native error codes should not appear in the Java exception message. For example, if your application fails to send an SMS message or a TCP peer connection is dropped, there is no reason to expect a native Symbian OS error code. When there is a native error code, it might be related to missing settings on the device. Then it would be worth looking up the native error code in the Symbian OS documentation¹⁰ and using this additional information to identify or better understand the problem.

3.8.2 Combining MIDlets and Other Run-time Environments

CLDC was not designed to be interoperable with native code or other run-time environments but when something is possible, a developer

¹⁰ As well as the Symbian OS reference, other free online resources, such as [www.newlcom/Symbian-OS-Error-Codes.html](http://www.newlc.com/Symbian-OS-Error-Codes.html), may help.

will find a use for it. In Symbian OS, there are many other run-time environments and very quickly some developers understood that this opens up opportunities; for example, native applications can launch a MIDlet or provide a MIDlet with services that it cannot get from the Java ME APIs. Such interoperability can be achieved without breaking any Java ME rule, guideline or specification. All it takes is a bit of creativity and a small amount of work. Note that you are doing something which is proprietary to Symbian OS and requires knowledge of native Symbian C++ development.

Three questions frequently occur in discussions on this topic – can a MIDlet suite be installed together with a native application, can a native application launch a MIDlet and can MIDlets use the native Symbian OS APIs?

A MIDlet suite can be installed with a native application. The suite can be bundled with the native application in the SIS file.¹¹ In exceptional circumstances, when there is dependency between a MIDlet and an associated native application, it may be convenient to deliver the entire application in a single SIS file. By default, the user must manually install the suite using a file browser. On some platforms, it is possible to automate this process but the System Installer is a manufacturer-customizable component and not all platforms support this behavior.

A MIDlet can be launched from a native application using the MIDP Push mechanism. For example, the JAD attribute below shows static registration of an inbound socket connection on port 5000, with no address filtering, which activates PushMIDlet:

```
MIDlet-Push-1: socket://:5000, PushMIDlet, *
```

Using JSR-211 CHAPI, you can register your MIDlet to be a handler for a specific type of content and use that mechanism to launch it from native code.

There is no standard way to invoke native Symbian OS APIs from CLDC, in general, or on Symbian OS, specifically. Java Native Interface (JNI) is a powerful Java SE mechanism that enables desktop applications to jump out of the Java run-time environment and execute native code but CLDC does not include JNI.¹²

3.8.3 Simulating Inter-process Communication

A recurrent question is whether it is possible to invoke native Symbian OS APIs, either through Java Native Interface (JNI) or by using another mechanism. The answer is that there is no standardized way to do so in

¹¹ If you don't know what a SIS file is, refer to information about native development.

¹² CDC-based platforms include JNI support but CDC is out of the scope of this book.

CLDC in general or on Symbian OS specifically. JNI is indeed a powerful Java SE mechanism that enables desktop applications to jump out of the Java run-time environment and execute native code but, due to the nature of mobile platforms, CLDC does not include JNI. (CDC-based platforms do have JNI support but CDC is out of the scope of this book.)

You may need to invoke native Symbian OS APIs where the required functionality is not available on the Java ME platform (e.g., when an MIDP application requires an SQL database) or the Java API does not have sufficiently fine-grained control (e.g., extremely fine-grained multimedia capabilities are not found in JSR-234 AMMS but are available in the native Multimedia Framework of Symbian OS). It may also be desirable for a MIDlet to communicate with an application (another MIDlet, a native Symbian C++ application or an application hosted by another run-time environment, such as Flash, Python, or Ruby) running in another process.

Symbian OS is a multitasking open platform in which native applications can be installed and multiple applications can run concurrently. Therefore, an inter-process communication (IPC) mechanism can be a solution, if required; for example, you may want to develop a hybrid mobile application that uses Java ME APIs for the back end and Flash Lite for the front end.

A possible solution could utilize a very simple concept (see Figure 3.22). Just as MIDlets can communicate over a network protocol with a remote server or a remote peer, they can communicate with another application running on the same host using the `localhost` address. Both sides then need to agree a common protocol to be used for requests and responses. Here we present a simple and generic reference design in order to demonstrate how the concept could be implemented.

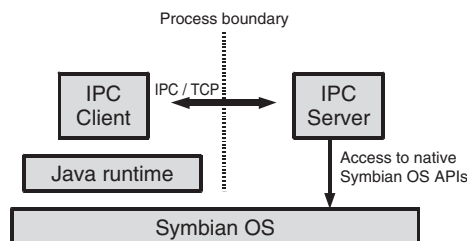


Figure 3.22 Inter-process communication over TCP localhost

First we start with possible code for an IPC client:

```
private static final int MY_NATIVE_SERVER_PORT = 9876;
private IpCommand cmd = new IpCommand(...);
private IpResponse res = null;
```



```

...
private void sendAndReceive() {
    IpcClientConnection ipcConn;
    try {
        ipcConn = IpcClientConnection.openConnection(MY_NATIVE_SERVER_PORT);
        // send command
        ipcConn.sendCommand(cmd);
        // receive response
        res = ipcConn.receiveResponse();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

```

So the steps to perform are:

1. Open an IPC client connection.
2. Send an IPC command.
3. Read the IPC response.
4. Process the data received.

For the sake of clean code and encapsulation, we create command and response classes:

```

public class IpMessage {
    private byte[] data;
    public IpMessage(byte[] data) {
        this.data = data;
    }

    byte[] getBuffer() {
        // TODO: handle buffer according to protocol
        return data;
    }
}
// TODO: implement derived classes according to protocol
public class IpResponse extends IpMessage {
    ...
}
public class IpCommand extends IpMessage {
    ...
}

```

We define the base class for all IPC connections, which provides low-level synchronous bi-directional communication:

```

public abstract class IpConnection {
    private final InputStream in;

```

```

private final OutputStream out;

IpcConnection(InputStream in, OutputStream out) {
    this.in = in;
    this.out = out;
}

// Generic IPC operations

protected void send(byte[] serializedCommand) throws IOException {
    out.write(serializedCommand.length);
    out.write(serializedCommand);
    out.flush();
}

protected byte[] receive() throws IOException {
    int responseLength = in.read();
    byte[] responseBuffer = new byte[responseLength];
    int read = 0;
    while(read != responseLength) {
        read += in.read(responseBuffer, read, responseLength - read);
    }
    return responseBuffer;
}
}

```

We define the IPC client-side connection class which can be returned from the static factory method `IpcClientConnection.openConnection()` when it is called with the remote IPC server port parameter:

```

public class IpcClientConnection extends IpcConnection {
    private final StreamConnection sc;
    private IpcClientConnection(StreamConnection sc) throws IOException {
        super(sc.openInputStream(), sc.openOutputStream());
        this.sc = sc;
    }

    public static IpcClientConnection openConnection(int ipcServerPort)
        throws IOException {
        StreamConnection sc = (StreamConnection)Connector.open(
            "socket://127.0.0.1:" + ipcServerPort,
            Connector.READ_WRITE, false);
        return new IpcClientConnection(sc);
    }

    // IPC operations
    public void sendCommand(IpcCommand cmd) throws IOException {
        super.send(cmd.getBuffer());
    }

    public IpcResponse receiveResponse() throws IOException {
        return new IpcResponse(super.receive());
    }
}

```

On the other side, there could be a native Symbian C++ application, a Flash Lite IPC client that uses ActionScript or a Python script. The IPC server code can be implemented in any run-time environment that supports accepting incoming TCP connections.¹³

For the sake of providing a reference implementation, here is an IPC server-side connection in Java ME:

```
public class IpcServerConnection extends IpcConnection {
    private final SocketConnection sc;
    private IpcServerConnection(SocketConnection sc) throws IOException {
        super(sc.openInputStream(), sc.openOutputStream());
        this.sc = sc;
    }
    public static IpcServerConnection openConnection(int ipcServerPort)
        throws IOException {
        ServerSocketConnection ssc =
            (ServerSocketConnection)Connector.open("socket://:9876");
        SocketConnection sc = (SocketConnection) ssc.acceptAndOpen();
        // TODO: use sc.getAddress() to ensure only local connections are
        // handled
        return new IpcServerConnection(sc);
    }

    // IPC operations

    public IpcCommand receiveCommand() throws IOException {
        return new IpcCommand(super.receive());
    }
    public void sendResponse(IpcResponse response) throws IOException {
        super.send(response.getBuffer());
    }
}
```

Opening a TCP port on your device is potentially a security hole. You should, therefore, apply any methods required to ensure authorization of incoming connections (e.g., accept incoming connections only from the device itself).

An additional workaround that could be considered is a somewhat more primitive form of communication. The passed information is put into a file (using JSR-75, `FileConnection`) at a known location and is polled on the other side. Obviously, this is not a suitable solution for the majority of cases, which require either JNI or IPC. However, it is an additional option for passing a message in an unreliable protocol.

There are obvious downsides to using this solution, such as breaking platform portability and the need to manage and maintain both Java and native support. However, TCP can be used to invoke native functionality that is not available through Java APIs or to provide IPC between a MIDlet and an application running in another process. In some cases,

¹³Of course, the Java application may be the client and the other process may be the server.

this option can mean that implementing a mobile application in Java becomes feasible.

3.9 Finding More Information

Before we finish this chapter, we give you some guidance on how to find answers to other questions you may have and how to find out information about other devices. You may need to develop a Java application or port it to another Symbian smartphone. What should be your approach when searching for information and where can you find information on a particular device?

As you have seen, technical competence is not enough when developing or porting a Java application to a Symbian smartphone. You need to know your way around the Symbian OS ecosystem. In the Symbian smartphone delivery chain, different stakeholders carry different functions and that brings different sources of information. It might appear confusing at first, so let's suggest a structured approach to finding more information.

Your information quest should follow the same route that a Symbian smartphone, and its Java ME subsystem in particular, follow until they reach the market. They start in Symbian, move to the UI platform and then to the phone manufacturer. Instead of starting from the point of view of the phone model, we suggest you start at the common denominator for all Symbian smartphones: Symbian OS itself. In en.Wikipedia.org/wiki/Symbian#Devices_that_have_used_the_Symbian_OS you can find general information about nearly every Symbian smartphone that is launched. For example, if you look up the Nokia N95, you are directed to en.wikipedia.org/wiki/Nokia_N95 which provides you with a wealth of information on the smartphone, such as the history, features, specification sheet, and variations. If you look up the Sony Ericsson W960i, you are directed to en.wikipedia.org/wiki/Sony_Ericsson_W960 which provides you with similar information. From these resources, you can also learn more about the main function of the device, the target audience which uses this device and how to best serve the needs of a user in that group.

Your next step is to visit the online resources and developer programs of the UI platform (e.g., S60 for the Nokia N95 and UIQ for the Sony Ericsson W960i), where you can find generic UI platform information as well as information for specific device models. You should look for the UI platform version and for more detailed technical information about the specific device, the Java ME specification, and the family of devices to which it belongs. For example, to find more information about an S60 device, we can go to www.forum.nokia.com/devices and select a device model number to see detailed information including: platform family, lead software APIs and other detailed information. To find information

about the Nokia 5800 XpressMusic or the Nokia N95, you can select the model from the list or filter the matrix according to device family e.g., S60 5th Edition or S60 3rd Edition FP.¹⁴ Similarly, to find more information about the Sony Ericsson W960i, you should visit **developer.uiq.com** and identify the UIQ version to which it belongs.

The next step would be to visit the phone manufacturer's online resources and developer programs. Here you can learn about device specific functionality. We chose the reference devices to show you another difference: in the case of the Nokia 5800 XpressMusic and Nokia N95, the UI platform owner and the device manufacturer are both Nokia. In the case of the Sony Ericsson W960i, the UI platform is UIQ but the device manufacturer is Sony Ericsson. So in this case, you have an additional developer program to go to, Sony Ericsson Developer World at **developer.sonyericsson.com**.

We recommend that you register for both the relevant UI platform and phone manufacturer developer programs as these are a major source of information. You can find articles, examples, tutorials, community support, blogs,¹⁵ discussion boards, SDKs and tools, and much more.

An excellent information source for Java developers is the Java Developer's Library (JDL), which covers both Series 40 and S60 platforms. It is available online¹⁶ and can be downloaded as an Eclipse-based viewer or a documentation plug-in for Eclipse-based IDEs.

To summarize, we suggest that you complement the usual Java ME resources (e.g., **java.sun.com**) with other resources that specialize in Java ME on Symbian OS. You probably need more than one developer program but, once you know your way around the Symbian OS ecosystem, you can get all the information you need quickly and effectively.

3.10 Summary

In this chapter, our intention was not to provide you with final or absolute answers. but to direct you towards the right questions and the right mindset to think about Java ME in a Symbian way.

In Section 3.1, we started by running the Java ME Detectors suite on reference devices that represent different flavors of Symbian OS. Section 3.2 then discussed the supported JSRs and additional APIs that you can use.

The lack of hard limits in Symbian OS for computing resources is liberating for Java ME. It allows you to code freely and focus on what's really important and not have to deal with workarounds. Of course, running on a powerful multitasking native platform gives you so much

¹⁴ FP stands for Feature Pack.

¹⁵ For example, **blogs.s60.com/tag/java**.

¹⁶ **www.forum.nokia.com/document/Java-ME-Developers-Library**

more power. Chapter 5 is dedicated to SDKs but we thought it important to explain the approach to tools, so we covered this in Section 3.6.

Setting and managing Java ME on a device is unique on Symbian OS because it is so tightly integrated with the native operating system. Section 3.7 took us through a tour of the settings on the reference devices. Having powerful native C++ APIs and a powerful platform gives choice. With choice comes potential and developers tend to realize potential, so Section 3.8 covered how you can reach out of the Java ME sandbox.

Finally, because there is so much knowledge and information floating around, we showed a structured approach to finding information about Java ME on Symbian OS in Section 3.9.

4

Handling Diversity

Walking into a mobile phone retail shop and looking around is a good way to appreciate the enormous diversity of devices and what different people want to do with them. There are devices that are used with one hand and devices that have touch screens; screens can be large or small; some devices have more than one screen; some have built-in GPS; and the list could go on. Symbian OS provides a common platform for many devices and families of devices, which ensures consistency of behavior and functionality for users and gives developers a common and consistent development platform across many models. For example, devices using S60 5th Edition, S60 3rd Edition, UIQ 3.1, and UIQ 3.3 all have a large common base.

The prevalence of Java ME on mobile phones has resulted in many Java ME implementations, which in turn have resulted in fragmentation. A common problem in Java ME on feature phones is that there are many implementations, each with its own set of supported JSRs, implementation defects, specification interpretations, optional features, implementation gaps, and so on. The story of Java ME on Symbian OS is fundamentally different. Having a single common Java ME platform for S60 5th Edition, S60 3rd Edition, UIQ, and other Symbian OS UI platforms considerably reduces the problems that plague other platforms, resulting in *diversity differences* rather than fragmentation.

Realistically, it is hard to find two families of Symbian smartphones that are completely identical. The diversity is due to native UI appearance, branding, behavior, or functionality. This chapter discusses some general guidelines that help developers to handle this diversity. We evaluate some of the main cases of diversity and suggest a way of handling them. Even though you need to take some measures to handle diversity, your Java application, which will be deployed on many Symbian smartphones, runs in a highly consistent Java ME environment with greater commonality than almost any other Java ME platform.

The purpose of this discussion is not to give a list of all the differences between Symbian smartphones. A more realistic approach is to handle a few cases and learn how to apply a similar pattern to a similar problem. We first outline some general approaches to handling diversity and then examine diversity in specific areas to act as a reference.

4.1 General Approaches to Handling Diversity

We have a range of possible options for handling diversity. At opposite ends of the scale, we have the multiple-implementations approach and the feature-drop approach. Both are quite extreme when it comes to Java ME on Symbian OS, which has much in common across the various devices.

The multiple-implementations approach implies fundamentally different implementations of the same functionality and the need to maintain multiple JARs. Taking this approach further means that, for each device model, you need to maintain a JAR file. The multiple-implementations approach usually starts with being uneconomical and ends up as being impractical. It should therefore be avoided.

The feature-drop approach implies that you totally remove a feature from the application, which is very harsh, unless it is a case of a JSR that is not supported. That might happen on Symbian OS but, generally, JSR support on Symbian smartphones is very predictable.

The multiple-implementation and feature-drop approaches are more suitable for handling fragmentation than diversity. Between these two extremes, there are a few more moderate approaches that can be combined to produce the fewest variants to cover most Symbian smartphones. These are techniques such as:

- **Detection:** The Java ME capabilities can be detected at various stages, each involving different types of required action. We utilized detection with the Java ME Detectors suite (see Section 3.1) and we use similar techniques.
- **Adaptive code:** Code that can handle variants in a simple way, without major refactoring or code instrumentation is termed 'adaptive code'. The same simple code can handle differences that may occur on devices coming from different Symbian OS UI platforms.
- **Flexible and modular design:** There are endless options when you take this route. For example, you can create abstraction layers, apply design patterns, or decouple functionality into modular subsystems.

Now we look at various areas in which diversity appears and examine how to handle it in simple ways that combine those three approaches.

4.2 Detecting Diversity using Properties

In Chapter 3, we used `JSRsDetectorMIDlet` to detect support for JSRs on Symbian smartphones. We use this technique again in Section 4.4. For now, there are other details about the Java ME subsystem that you probably want to find out. Another method of detecting device capabilities is by querying the system properties.

System properties can be retrieved using the `System.getProperty()` method, which receives as a key the name of a system property and returns its value (or `NULL` if there is no property with that key). The system properties are logically grouped into properties that give information about the environment, package discovery, JSR capabilities and the device-specific properties.

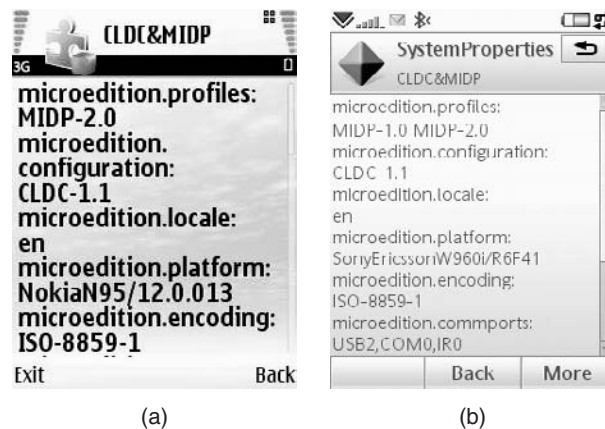


Figure 4.1 Environment system properties on a) Nokia N95 and b) Sony Ericsson W960i

You can get information about the Java ME environment (see Figure 4.1), for example:

- `microedition.profiles` is a list of the profiles that the device supports. For MIDP 2.0 devices, this property contains at least MIDP-2.0.
- `microedition.platform` returns the identifier of the platform (e.g., you can identify whether you are running on the Nokia N95 or Sony Ericsson W960i).

- `fileconn.dir.photos` points to the directory where photos captured with an integrated camera or other images are stored (it returns `file:///C:/Data/Images/`).

You can also check versions of supported JSRs. For example, `microedition.media.version` and `microedition.pim.version` indicate if the optional packages are supported. If they are, a version string is returned (e.g. "1.1"), as shown in Figure 4.2.

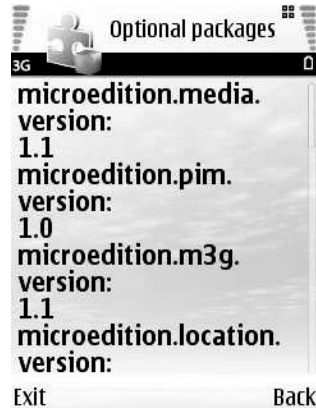


Figure 4.2 Support for optional packages

You can query about JSR capabilities (Figure 4.3). For example, `supports.mixing` returns true or false depending on whether MMAPI audio mixing is supported.



Figure 4.3 Support for JSR capabilities

There are ways to get JSR-specific information through system properties. JSR-184 Mobile 3D Graphics API properties can be queried with the `Graphics3D.getProperties()` method, which returns a hash-table of 3D graphics support properties and their values (see Figure 4.4a). In the Bluetooth API, `LocalDevice.getProperty()` should return properties that are specific to Bluetooth support such as the maximum number of connections (see Figure 4.4b).

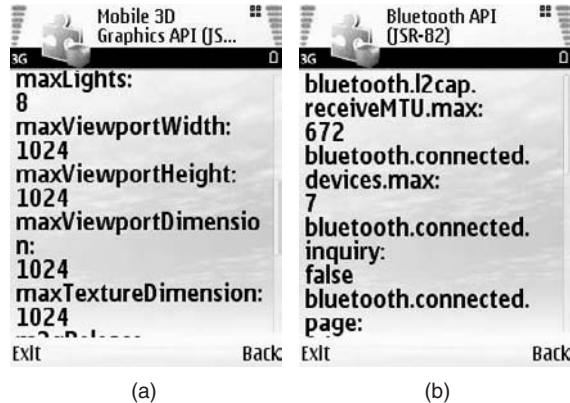


Figure 4.4 System properties for a) the Mobile 3D Graphics API and b) the Bluetooth API

The device-specific system properties can give you additional information which is proprietary to a device or manufacturer. For example, S60 3rd Edition FP2 and S60 5th Edition system properties include support for telephony-related information, such as the International Mobile Equipment Identity (IMEI) number, International Mobile Subscriber Identity (IMSI) number, current (GSM/CDMA) network signal strength and more.

There is a lot of information available to you through detection, whether you detect it through a thrown exception or through system properties. The general idea is to use detection to configure your application's behavior. Use the information you gain from detection to decide on the application behavior and actions that are affected.

4.3 Using Adaptive Code and Flexible Design to Handle Diversity

Not all diversity issues are as obvious as unsupported JSRs or media support fragmentation. The differences may be small yet their impact on the user can be significant.

There is more than one way to handle those subtle differences. You could choose to apply a technical solution and write more adaptable code that produces satisfactory results on all the different devices. Alternatively, you could change the usage and design of the application. In some cases, you need to apply both techniques.

As we said earlier, a central area of Symbian OS customization is the UI. Phone manufacturers are required to customize the Symbian OS UI and to implement their own look and feel. To ensure the user experience remains identical between Java applications and native applications, Symbian OS native customization of the UI applies to the Java ME subsystem as well. This is why similar LCDUI code, when run on Symbian smartphones from different manufacturers, might produce subtle differences that can nevertheless significantly change the usage of the application.

In this example, we consider how commands map to softkeys on two different UI platforms. Let us consider the following snippet of code:

```
private Command dismiss = new Command("Dismiss", Command.OK, 0);
private Command back = new Command("Back", Command.BACK, 0);
private Command exit = new Command("Exit", Command.EXIT, 0);
private Command reply = new Command("Reply", Command.SCREEN, 0);
private Form form;
public SoftButtonsExamp1et0(){
    form = new Form("Command Examp1et 0");
    form.addCommand(dismiss);
    form.addCommand(reply);
    form.addCommand(back);
    form.addCommand(exit);
    form.setCommandListener(this);
}
```

The example uses four command types: OK, BACK, EXIT and SCREEN to specify the intent of the commands. The four commands are mapped to softkeys according to the device's native layout. Figure 4.5 illustrates the differences in how the same application would appear on our reference devices. As you can see, the Nokia N95 has two softkeys and the Sony Ericsson W960i has three softkeys.

Now that we have seen the code and the differences, let us think about how such a small difference can have a significant impact on the user. For example, consider a usability case which defines that when an SMS message is received, the user should see the message and respond or dismiss it with only a single click. Bob uses the application to send an SMS message from his Sony Ericsson W960i to his colleague Alice, who uses Nokia N95. Alice should be able to respond to the message or dismiss it, with only a single click. At the moment, neither Bob nor Alice can do those actions with a single click.

Although we cannot change the restrictions imposed by LCDUI or by the softkey layout of the UI platform, there are a few options we can

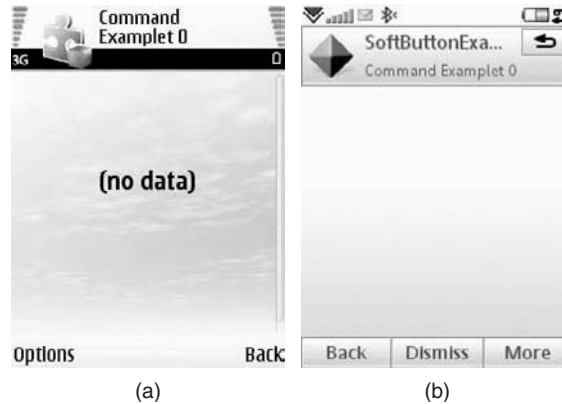


Figure 4.5 Layout of softkeys on a) Nokia N95 and b) Sony Ericsson W960i

think of. First, there is the option of two different code bases, one for each UI platform, and deploying different JARs according to the target device. The implications are increased maintenance, possible application fragmentation, increased costs, needing to sign the application for each code base, and so on. Multiple JAR files is not a scalable solution when you have an installed base of more than just a few target devices.

Another possible solution involves extra effort and a radical change to application usage: the application UI could use a screen menu instead of softkeys. The screen menu could be a flashy SVG UI or a simple LCDUI List. However, we can first try to find a simpler solution which does not require us to reimplement our code base.

Let's see if the screen actions and sequence of the current application can be improved. There are four options for the user: Reply, Dismiss, Back and Exit. It seems that users would very rarely use Back or Exit when an incoming message is displayed, so we can remove those actions from this screen. Of course, we must give the user an option to exit from other screens, which are more suitable to the Exit action.

```
private Command dismiss = new Command("Dismiss", Command.OK, 0);
private Command reply = new Command("Reply", Command.SCREEN, 0);
private Form form;

public SoftButtonsExamplet1(){
    form = new Form("Command Examplet 1");
    form.addCommand(dismiss);
    form.addCommand(reply);
    form.setCommandListener(this);
}
```

As you can see in Figure 4.6, the user still cannot respond with a single click. The Nokia N95 user has to select the Reply and Dismiss actions

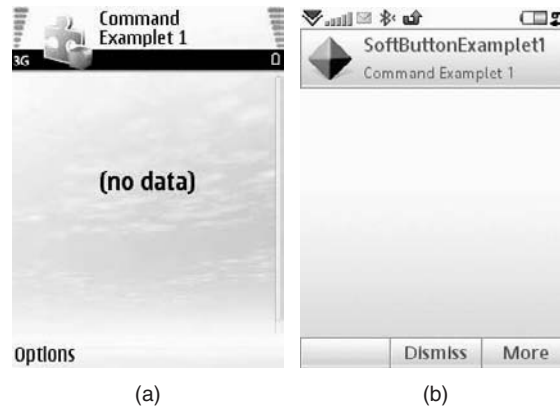


Figure 4.6 Reduced button layout on a) Nokia N95 and b) Sony Ericsson W960i

from the Options menu and the Sony Ericsson W960i user has to open the More menu to choose the Reply option.

We can change the command type of one of the commands to map to a different softkey. The following code changes the Dismiss action to be of command type CANCEL.

```
private Command dismiss = new Command("Dismiss", Command.CANCEL, 0);
private Command reply = new Command("Reply", Command.OK, 0);
```

The MIDP specification says this about command type CANCEL:

The application hints to the implementation that the user wants to dismiss the current screen without taking any action on anything that has been entered into it.

As you can see in Figure 4.7, that produces the correct result on the Nokia N95 (as Dismiss is mapped to the Back button, the only remaining option, Reply, is mapped to the other softkey). On the Sony Ericsson W960i, the Dismiss command has been mapped to the system button at the top right of the screen.

That is probably the right thing to do as it respects the native platform usage policy. In some cases, you may prefer both Sony Ericsson W960i and Nokia N95 users to see a softkey with a Dismiss label. Let's change the Dismiss command to be of type BACK and see what happens:

```
private Command dismiss = new Command("Dismiss", Command.BACK, 0);
private Command reply = new Command("Reply", Command.OK, 0);
```

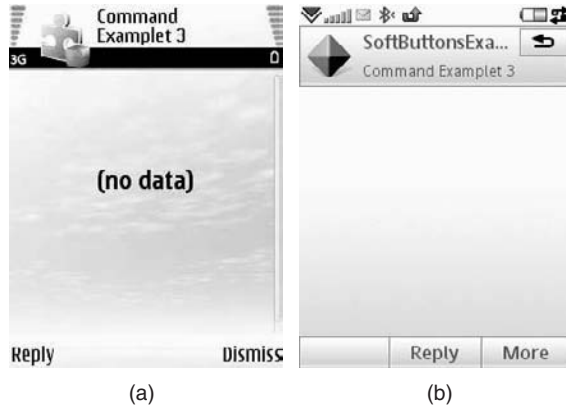


Figure 4.7 Dismiss command mapped to Cancel softkey on a) Nokia N95 and b) Sony Ericsson W960i

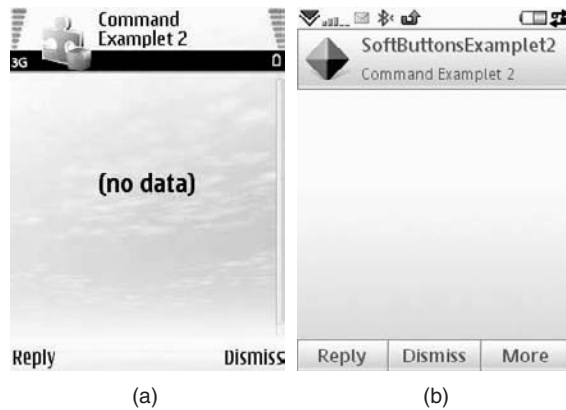


Figure 4.8 Dismiss command mapped to Back softkey on a) Nokia N95 and b) Sony Ericsson W960i

Figure 4.8 shows that both labels are visible and users of both UI platforms can perform the actions with a single click.

Is it a cheat to change the Dismiss type to BACK? Take a look at the MIDP specification definition of the BACK command type:

A navigation command that returns the user to the logically previous screen. ... Note that the application defines the actual action since the strictly previous screen may not be logically correct.

So this is not a cheat but a legitimate change that can be regarded as making our code adaptable to various Symbian OS UI platforms.

I hope that this example demonstrates how a simple rethink of the application usage and some minor changes in the code to make it more adaptable would give the same user experience on different Symbian OS UI platforms.

In general, Symbian OS is customized by phone manufacturers; therefore, there could be subtle differences which impact the usability of your application. Plan the UI to ensure that users on different UI platforms can use the application as intended. A rethink of the application usage and more adaptive code are proven techniques that you can apply before resorting to more radical changes.

4.4 Handling JSR Fragmentation

Fragmentation of JSRs is a problematic issue in Java ME, in general: your application requires a certain JSR for its functionality, yet that JSR is not supported on the target device. There are no magic solutions that you can apply, even when it comes to Symbian OS. If a JSR is not supported on a certain device, there is not much you can do about it. However, Java ME on Symbian OS gives phone manufacturers a large common set of JSRs, on the scale of the set of MSA 1.1 Component JSRs. That reduces the amount of JSR fragmentation considerably for Java ME applications targeting Symbian smartphones.

There is a difference between the Java ME platform being formally compliant with MSA and the Java ME platform supporting the set of MSA 1.1 JSRs. A Java ME platform that supports all MSA Component JSRs but does not comply with all MSA clarification is still not MSA-compliant. Without playing down the lack of compliance, the major importance for a developer is probably the availability of JSRs. Unlike many low-end platforms, which support only MIDP 2.0 or JTWI, the latest Symbian smartphones are already compliant with MSA Subset and support additional Component JSRs that bring them closer to MSA Fullset. In reality, JSR fragmentation will always exist to some extent.

We discuss two guidelines on how to handle JSR fragmentation according to the modularity of your application. In the graceful functional degradation approach, your application can execute without the unsupported JSR, although with degraded functionality. In the optionally replaceable module approach, your application can remain at the same functional level by using another JSR, which is supported.

4.4.1 Graceful Functional Degradation

All references and usage of a JSR that may not be supported need to be encapsulated within a bounded part of the application. If the JSR is

not supported by a specific device, that part of the application is never accessed in any way. It is not instantiated or used and there must be no direct usage of the JSR outside this module.

To encapsulate all usage of that JSR, you also need to define interfaces between that module and the rest of the application that are the only way of accessing the JSR (see Figure 4.9). The module is loaded and used only after detection to check for the JSR availability (in a similar way to JSRsDetectorMIDlet in Chapter 3).

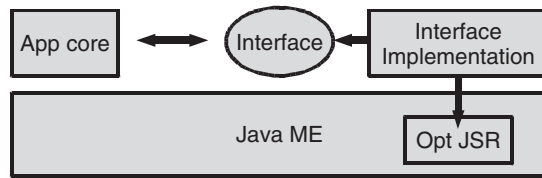


Figure 4.9 Implementation of graceful functional degradation

For example, your application should encapsulate all usage of JSR-82 in a module, behind a set of interfaces that are used to send and receive messages over Bluetooth. When your application starts, it detects if Bluetooth is supported and instantiates the Bluetooth module, or not. The usability of the application must also be changed and any functionality dependent on Bluetooth should be disabled (e.g., commands or menu items that trigger usage of Bluetooth should be disabled or removed).

```

private void initBluetoothFunctionality(){
    try {
        Class c = Class.forName("javax.bluetooth.LocalDevice");
        // if we reach here, JSR-82 is supported
        btModule = new MyBluetoothCommModule();
        // TODO: enable all Bluetooth-related functionality
    }
    catch (ClassNotFoundException e) {
        // JSR-82 is not supported on this handset
        btModule = null;
        // TODO: disable all Bluetooth-related functionality
    }
}

```

The advantage to this approach is that the module can always reside in the JAR. There is no need to maintain multiple versions with and without the module. If the JSR is not supported, the module is not used. The disadvantage is that the JAR is sometimes needlessly bigger than it could have been.

4.4.2 Optionally Replaceable Module

The optionally replaceable module approach implies that your JAR includes more than a single implementation of the required module (see Figure 4.10). The solution design is similar to the previous case but there is a fallback mechanism and the run-time module can be replaced by more than one implementation, so more than one module implementation would reside in the JAR.

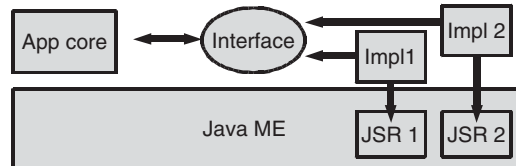


Figure 4.10 Implementation of optionally replaceable module

A possible usage of this solution is to implement the model–view–controller (MVC) design pattern with more than one view module. In the JAR, there would be two or more implementations of the application view, each using a different JSR. For example, one implementation may use JSR-226 SVG, another implementation may use JSR-184 3D Graphics, and a third implementation may use Canvas:

```

private void loadUiModule() {
    // first attempt - JSR-226 SVG
    try {
        Class c = Class.forName("javax.microedition.m2g.SVGAnimator");
        // TODO: load SVG implementation of UI
        ui = new UiModuleSVG();
    }
    catch (ClassNotFoundException e) {
        // JSR-226 SVG is not supported
    }
    // fallback option 1 - JSR-184 3D
    if (ui == null) {
        try {
            Class c = Class.forName("javax.microedition.m3g.Graphics3D");
            // TODO: load 3D implementation of UI
            ui = new UiModule3D();
        }
        catch (ClassNotFoundException e) {
            // JSR-184 3D is not supported (unlikely in Symbian OS!)
        }
    }
    // last fallback option - MIDP LCDUI
    if (ui == null) {
        ui = new UiModuleLCD();
    }
}
  
```

The advantages and disadvantages are the same as with graceful functional degradation – the module is always in the JAR, which means the JAR is bigger but there is only one JAR to maintain.

4.5 Handling Transitions Between Foreground and Background

Since Symbian OS is a multitasking operating system, the Application Management Software (AMS) can move your MIDlet to the background to allow another application to execute concurrently. After a while, your MIDlet may then be sent back to the foreground.

In S60 5th Edition and 3rd Edition FP2 devices, the MIDlet lifecycle methods are not called by default. To request the AMS to invoke `MIDlet.startApp()` and `MIDlet.pauseApp()`, you need to add the `Nokia-MIDlet-Background-Event`¹ proprietary JAD attribute with the value "pause". By default, the attribute value is "run" and `pauseApp()` and `startApp()` are not called by the AMS. An alternative way to deal with the MIDlet lifecycle methods not being invoked is to override the `Canvas.hideNotify()` method, which is called when the MIDlet is sent to the background, and the `Canvas.showNotify()` method, which is called when it is moved to the foreground.

What you should ensure is the handling of state switching in both cases so that your application behaves similarly on Symbian smartphones from different vendors. A possible solution would be to decouple the handling of state switching from the specific notification events to handle them in a separate location.

```
public class StateHandler{

    public synchronized void handleSwitchToBackground(){
        background = true;
        // TODO: handle switching to background
    }
    public synchronized void handleSwitchToForeground(){
        background = false;
        // TODO: handle switching to foreground
    }
}
```

Then, MIDlet lifecycle methods delegate the calls to the decoupled location:

¹ www.forum.nokia.com/document/Java_ME_Developers_Library

```
public void startApp() {
    handler.handleSwitchToForeground();
}
public void pauseApp() {
    handler.handleSwitchToBackground();
}
```

Canvas notification methods also delegate the calls to the decoupled location:

```
protected void hideNotify(){
    handler.handleSwitchToBackground();
}
protected void showNotify(){
    handler.handleSwitchToForeground();
}
```

In Canvas-based MIDlets, for high-level UI screens, the MIDlet could periodically check `Displayable.isShown()` on the current `Displayable` object. For example, a separate thread could check every once in a while (e.g., every second) if the current displayable object is visible. If it is not, the application is moved to the background state and, if it is, the application is moved to the foreground state.

Requesting a return to the foreground is possible but there are differences in the implementation. On some platforms, calling `MIDlet.resumeRequest()` is required, while on others calling `Display.setCurrent()` with a displayable object is required. Applications can switch themselves to the background on Nokia devices by calling `Display.setCurrent(NULL)`.

```
public void goToBackground(){
    // will work on Nokia devices
    Display.getDisplay(this).setCurrent(null);
}

public void returnToForeground(){
    // duplicate action to ensure it is performed on various devices
    this.resumeRequest();
    Display.getDisplay(this).setCurrent(current);
}
```

4.6 Supporting Diverse Input Mechanisms

At the beginning of the chapter, we mentioned that some devices can be used with one hand while others have a touch screen. For example, S60 5th Edition includes UI support for a touch screen as well as UIQ. If a user buys a Nokia 5800 XpressMusic device with touch-screen support,

a touch or stylus is used for interaction. Java ME developers should take touch devices into consideration when designing and planning the user interaction as more and more touch-enabled devices appear in the market.

According to the MIDP specification, high-level LCDUI widgets do not receive low-level touch events. LCDUI widgets continue to receive the same events as before. This section, therefore, applies mostly to canvas-based MIDlets.

4.6.1 Handling Stylus Events

Stylus events can be detected using the `Canvas` class, which provides developers with methods to handle pointer events:

- `Canvas.hasPointerEvents()` indicates whether the device supports pointer press and release events (e.g., when the user touches the screen with a stylus).
- `Canvas.hasPointerMotionEvents()` indicates whether the device supports pointer motion events (e.g., when the user drags the pointer).
- `Canvas.pointerDragged(int x, int y)` is called when the pointer is dragged.
- `Canvas.pointerPressed(int x, int y)` is called when the pointer is pressed.
- `Canvas.pointerReleased(int x, int y)` is called when the pointer is released.

The pointer events delivery methods are only called while the `Canvas` is visible on the screen. `CustomItem` also supports pointer events and the way to query for supported input methods is the `getInteractionModes()` method.

The following snippet of code demonstrates how to handle stylus events:

```
public class StylusSupportExamplet extends Canvas {

    // Members
    private final TouchPoint pressed = new TouchPoint();
    private final TouchPoint released = new TouchPoint();
    private final TouchPoint dragged = new TouchPoint();

    // Stylus support
    // Indicate if pointer and pointer motion events are supported
    public boolean supportsStylus() {
        return this.hasPointerEvents() && this.hasPointerMotionEvents();
    }
}
```

```

    }
    protected void pointerPressed(int x, int y) {
        pressed.x = x;
        pressed.y = y;
        // TODO: handle event accordingly
    }
    protected void pointerReleased(int x, int y) {
        released.x = x;
        released.y = y;
        // TODO: handle event accordingly
    }
    protected void pointerDragged(int x, int y) {
        dragged.x = x;
        dragged.y = y;
        // TODO: handle event accordingly
    }
}

// Encapsulation of [x,y] point on the screen
class TouchPoint {
    int x, y;

    public String toString(){
        return "[" + x + ", " + y + "]";
    }
}

```

To ensure that a user can use your application with a touch screen, you first need to add code to handle stylus events to the relevant Canvas-based screens.

4.6.2 Replacing Directional Keys

There are other issues that we need to consider when targeting a variety of Symbian smartphones and how your application must support the input mechanism used on that phone. These issues are not all enumerated, but we refer to a few to highlight how to plan to handle various input mechanisms.

A phone designed for touch-screen interaction might not have directional keys. You might design the greatest space invaders game but would a user enjoy playing it using the stylus? You need to think about such cases.

- It is possible that your application is simply not suitable for this class of devices.
- If there are no directional keys, perhaps there is another set of designated keys on the device? For example, the Sony Ericsson W960i has a jog dial as the standard mechanism for scrolling up and down and selecting items.

- You might want to add your own implementation of virtual keys that will be rendered at the sides of the screen to allow the user to interact via touch.
- You may want to consider Nokia's simple solution for MIDlets that use a Canvas and are not originally made for touch devices. The `Nokia-MIDlet-On-Screen-Keypad` proprietary JAD attribute enables a virtual keypad that generates key events for the canvas-based MIDlet (see Figure 4.11). The possible values for `Nokia-MIDlet-On-Screen-Keypad` are "no", "gameactions" and "navigationkeys". Regardless of the value, pointer events still work normally in the Canvas area.

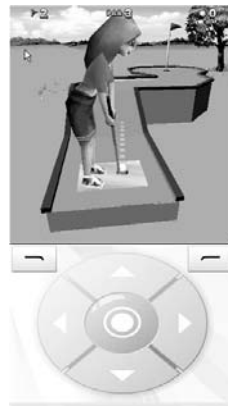


Figure 4.11 Canvas MIDlet with `gameactions` virtual keypad

4.6.3 Implementing Device Conventions

If your application supports the touch screen, it should also adhere to the usage conventions of the device. For example, touching the sides of the screen may be equivalent to normal usage of the directional keys; your application should follow this convention and handle touches at the sides of the screen as pressing on directional keys. Additionally, your application must not block the conventional mechanism from operating. If your game extends the `LCDUI GameCanvas` then it is your responsibility to ensure the key-press events are not blocked because of the handling of stylus events. Ensure that your stylus-event-handling methods invoke the same method of the base class to allow it to handle the stylus events and translate them to key events, if applicable. Pressing at the side of the Sony Ericsson W960i screen triggers a key event with a directional code (left or right). If your application extends the `GameCanvas` and does not delegate the stylus events to the same `GameCanvas`, the key event is not dispatched by the implementation.


```
protected void pointerPressed(int x, int y) {
    pressed.x = x;
    pressed.y = y;
    // TODO: handle event accordingly
    // Call super.pointerPressed(x, y) otherwise, depending on the UI
    // platform, keyPressed(int code) might not be invoked
    super.pointerPressed(x, y);
}

protected void keyPressed(int code){
    // TODO: handle key-event
}
```

4.6.4 Interacting with Graphical Elements

You should also consider the size and location of the graphical elements with which the user interacts. Users don't always use a stylus – they may use their fingers. If you render a button which the user will touch, the button has to be of sufficient width and height to allow the user to press it comfortably. Also, the gap between adjacent interaction elements should be sufficient to ensure that the user cannot accidentally press more than one element at a time.

4.6.5 Opportunities of Platform Input Mechanisms

Platform input mechanisms also provide a developer with good opportunities. If there is touch-screen support, you can add swipe interaction to your application. If you are using high-level LCDUI widgets, you have free access to all the native text-entry mechanisms, such as predictive text, handwriting recognition, and a virtual on-screen keyboard. Such input mechanisms remain transparent to your code, which receives the usual high-level events, without any additional cost.

The single common Java ME platform on a large number of Symbian smartphones mitigates the problem of Java ME key code fragmentation. The same Java ME key codes are used in the majority of Symbian smartphones, even from different phone manufacturers.

4.7 Handling Diverse Multimedia Formats and Protocols

The multimedia support in MMAPI is quite openly defined. The general-purpose design of MMAPI implies that the supported set of protocols and content formats is not mandated and is left for the Java ME implementation to define. Because of the lack of a mandatory set, MMAPI also cannot specify which content types should work on top of which protocol.

In Java ME on Symbian OS, the definition of MMAPI content types and protocol support is derived from the native Symbian OS Multimedia

Framework (MMF). This direct link between the two frameworks is because Java ME on Symbian OS tightly integrates with the native Symbian OS MMF. A clear benefit of this approach is that Java ME applications are consistent with the Symbian OS platform in regards to supported content, playing and recording behavior.

In order to give a Java application a way to query the system about its multimedia capabilities, the `MMAPIManager` class defines two methods that you can use:

- `Manager.getSupportedContentTypes(String protocol)` returns the list of supported content types for a given protocol. For example, if the given protocol is `http`, then the returned value would be the supported content types that can be played back with the HTTP protocol. To get all the supported content types, you can pass `NULL` as the protocol.
- `Manager.getSupportedProtocols(String contentType)` returns the list of supported protocols for a given content type, which identify the locators that can be used for creating MMAPI players. For example, if the given content type is `video/mpeg`, then the returned value would be the supported protocols that can be used to play back `video/mpeg` content. To get all the supported protocols, you can pass `NULL` as the content type.

The snippet of code below, which can also be added to the Java ME Detectors MIDlet, detects the supported content types and protocols and displays them:

```
private void detectSupport(){
    try {
        form.append("getSupportedContentTypes:");
        String[] supportedContentTypes =
            Manager.getSupportedContentTypes(null);
        for(int i = 0; i < supportedContentTypes.length; i++){
            form.append(supportedContentTypes[i]);
        }
        form.append("getSupportedProtocols:");
        String[] supportedProtocols = Manager.getSupportedProtocols(null);
        for(int i = 0; i < supportedProtocols.length; i++){
            form.append(supportedProtocols[i]);
        }
    }
    catch (Exception e) {
        form.append(e.getMessage());
    }
}
```

Another source of information is the MMAPI properties that can be queried by `System.getProperty(String key)`. JSR-135 MMAPI defines the following keys:

- `supports.mixing`
- `supports.audio.capture`
- `supports.video.capture`
- `supports.recording`
- `audio.encodings`
- `video.encodings`
- `video.snapshot.encodings`
- `streamable.contents`.

Now that we know how to detect, let's see how we can use the information. The first decision to take is when to use the detection – during development or dynamically at application run time.

If you detect the multimedia capabilities during development, you can package the multimedia content with the JAR according to the support on the target device. If you are targeting more than one device, you should package content whose type and format is supported by all the targeted devices. The advantage is that, once the suite is installed, the content is available on the device and does not need to be fetched. The disadvantages are that the content increases the JAR size and cannot be dynamically changed. So if your suite needs to be signed you cannot change the content later without going through the signing process again.

If you detect the multimedia capabilities at run time, you can package in your JAR different versions of the same content to be used according to the detected support or you can fetch the appropriate content from a remote server. When the MIDlet starts, it detects the available support and can play the content immediately by creating a `Player` with the appropriate locator or can fetch content of the appropriate type from the server. For caching, the MIDlet can use the RMS or JSR-75 `FileConnection` which is supported on all current Symbian smartphones. Handling can be improved further by delegating to the remote server the decision about which content to use or fetch. The MIDlet sends the query results to the remote server without processing them and the decision logic resides on the server. Your application can detect the multimedia capabilities once, the first time it is launched or can do it in subsequent runs.

The advantages of this approach are a smaller JAR size and the dynamic nature of content support. The disadvantage is the dependency on a remote server which must be accessible to download the content (the dependency can be minimized if the content is fetched just once at first application run).

4.8 Handling Screen and Display Diversity

All Java ME developers are familiar with issues of screen and display differences.

4.8.1 Proprietary Solutions

In S60 5th Edition and some of the S60 3rd Edition devices, proprietary JAD attributes were introduced to automatically handle scaling and orientation switches. For the full list of proprietary S60 JAD attributes please refer to the Java ME Developer Library.²

`Nokia-MIDlet-Target-Display-Size` specifies the target size for the MIDlet and `Nokia-MIDlet-Original-Display-Size` specifies the screen for which the MIDlet was designed. Automatic scaling does not distort image aspect ratio. The content is scaled to fit the display dimensions and then centered on the display with black color used to fill at the sides (see Figure 4.12).

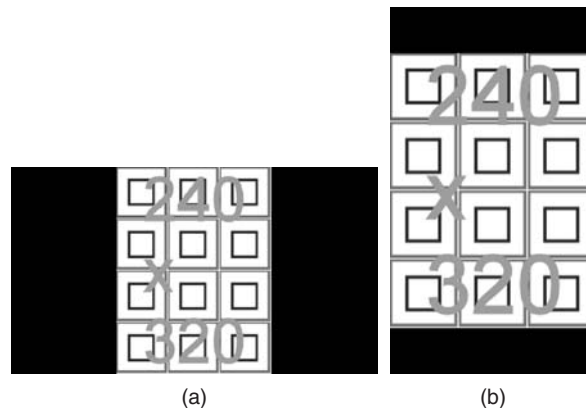


Figure 4.12 Setting a constant MIDlet display area

An additional attribute `Nokia-MIDlet-Canvas-Scaling-Orientation-Switch` indicates if the MIDlet can support the resolution specified in `Nokia-MIDlet-Original-Display-Size` in both portrait and landscape resolutions. For example, you could specify that your MIDlet display area is 240x320 in both portrait and landscape modes by specifying in the JAD:

```
Nokia-MIDlet-Canvas-Scaling-Orientation-Switch: false
Nokia-MIDlet-Original-Display-Size: 240,320
```

² www.forum.nokia.com/document/Java_ME_Developers_Library/

If you add additional attributes that change how the screen is divided (e.g., `Nokia-MIDlet-On-Screen-Keypad` adds a virtual keypad to the screen), there are even more combinations that can be used by your MIDlet. Adding the following JAD attributes adds the virtual keypad and enables orientation switches (see Figure 4.13):

```
Nokia-MIDlet-Canvas-Scaling-Orientation-Switch: true
Nokia-MIDlet-Original-Display-Size: 240,320
Nokia-MIDlet-On-Screen-Keypad: gameactions
```

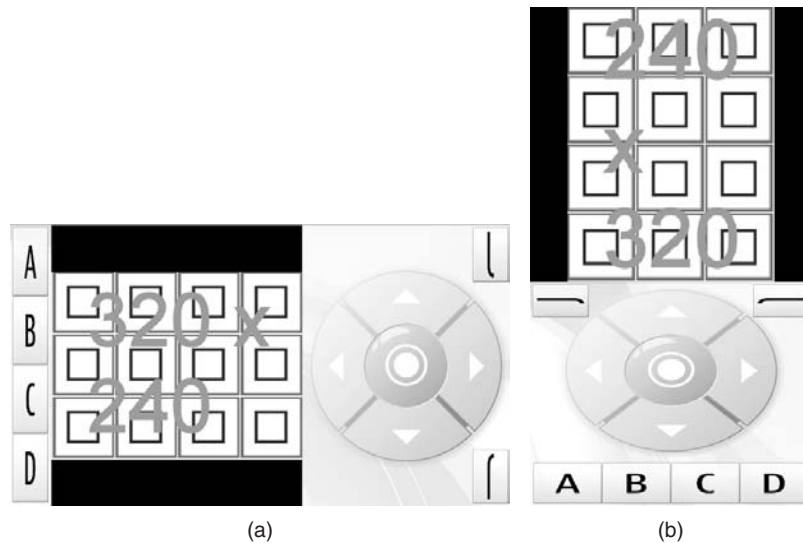


Figure 4.13 Scaling, orientation switches and virtual keypad

An additional JAD attribute in S60 5th Edition, `Nokia-MIDlet-App-Orientation`, allows MIDlets to force either portrait or landscape UI orientation. MIDlets need to add the JAD attribute with the value `portrait` or `landscape`. If the attribute is undefined, the MIDlet is required to support dynamic orientation switching.

4.8.2 Generic Solutions

The screen and display parameters are the size and the color depth. MIDlets should never assume any specific screen size. Instead, they should query the size of the display and adjust their view accordingly.

Here is a snippet of code taken from `DisplayDetectorMIDlet` in the Java ME Detectors suite:

```
public void detectScreenSize(){
    MyCanvas canvas = new MyCanvas();

    // detect normal size
    int width = canvas.getWidth();
    int height = canvas.getHeight();
    form.append("width:" + width);
    form.append("height:" + height);

    // detect full screen size
    canvas.setFullScreenMode(true);
    width = canvas.getWidth();
    height = canvas.getHeight();
    form.append("full width:" + width);
    form.append("full height:" + height);
}
```

Please note the second check of width and height, after setting the screen mode to full screen.

A few of the generic solutions to diversity in screen size are tiling, centering the application view, scaling the view images and deployment of multiple size images.

Using tiling you can use smaller images and render them repeatedly to cover the whole screen or parts of the screen. You could use a single image and render it at various locations or you could use multiple images and tile them according to some policy. For example, you could use three images, rendering two of them at the corners and tiling the third across the screen between the two corners (see Figure 4.14).



Figure 4.14 An image tiled between two others

You could query the screen dimensions and render the image in the center. To fill the gaps left between the image and the edges of the screen, you could use a background color or tiling.

MIDP does not have a library for image scaling. However, you could find such a library or develop it yourself. Then, you need to decide when in the execution of the application is the most appropriate time to do the scaling. For example, you could do the scaling once at the first launch and cache the scaled images in the file system to be loaded for subsequent runs.

If you deploy multiple images of various sizes, your application can choose at run time which one is closest to the required dimensions.

Display detection is less frequently used but is still useful in certain cases. A few display properties can be retrieved using the static `Display` instance, such as the number of colors that can be represented, the supported number of alpha transparency levels, the best image height and width for a given image type, and the color specifiers that match well with the native color scheme.

The following code displays some of those properties, as shown in Figure 4.15:

```
public void detectDisplay(){
    Display display = Display.getDisplay(this);
    form.append("\nnumColors:" + display.numColors());
    form.append("\nalpha:" + display.numAlphaLevels());
    form.append("\nfg color:" + Display.COLOR_FOREGROUND);
    form.append("\nbg color:" + Display.COLOR_BACKGROUND);
    form.append("\nleh:" +
        display.getBestImageHeight(Display.LIST_ELEMENT));
    form.append("\nlew:" + display.getBestImageWidth(Display.LIST_ELEMENT));
}
```

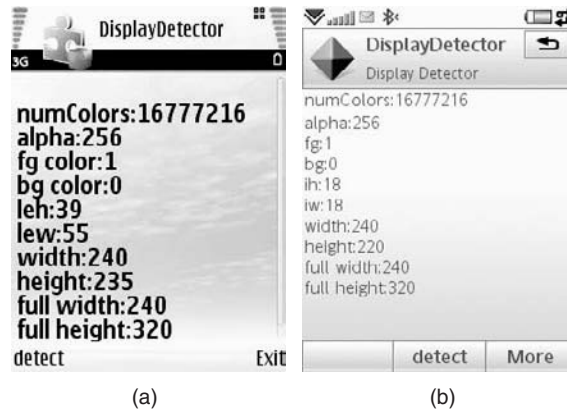


Figure 4.15 Querying `Display` and detecting screen dimensions

Generally the display attributes are useful for creating `CustomItem` objects that match the user interface of other items and for creating user interfaces within the canvas that match the user interface of the rest of the system.

Another point related to screen size is orientation. It can change, for example, when the device slider shifts, flip opens or closes. When the orientation changes, the implementation calls the `Displayable.sizeChanged(int, int)` method. Overriding this method is useful for

subclasses of `Canvas` to scale their graphics in order to fit their contents within the display area. Applications that support dynamic orientation switching also need to consider where the soft keys are located on the device. Once the orientation is switched, the keys have also moved and the user might not be able to use them properly.

4.9 A Last Resort: NetBeans Preprocessing

When there are no other options, consider NetBeans preprocessing. It is far from being the best solution but it is a technological solution that would not exist if there was no engineering need for it.

The NetBeans device fragmentation solution is based on the use of project configurations. A project should have one project configuration for each distribution JAR and JAD that you would like created for that project. The second part of the device fragmentation solution is the ability to specify certain blocks of code in your source files as being specific to one or more configurations. This is accomplished by using actions in the right-click context menu of the editor, or from the Edit, Preprocessor Blocks menu.

Preprocessing modifies the code in your source files before the code is parsed by the compiler. The preprocessor modifies the code according to directives inserted as code blocks into the code. These code blocks are marked visually in the NetBeans Source Editor and are included (or excluded) when you build the JAR for a specific project configuration or ability. You can use these code blocks to create, manage and track code that is specific to one or more project configurations or abilities.

For example, if you are writing an application targeted for several different devices, you can create a project configuration for each device (e.g., S60, UIQ), and a preprocessor block with directives for each project configuration. You can then test the application for each device quickly by changing the active project configuration and running the application.

It is not a recommended solution. You can choose it if there is no better alternative.

4.10 Summary

Any discussion about Java ME is incomplete without referring to the issue of fragmentation. And no discussion about fragmentation can enumerate a comprehensive list of all issues and all solutions.

We discussed various approaches to deal with fragmentation, covered some areas that tend to be diverse on Symbian OS and suggested some possible solutions. You can use the solutions, generalize them,

change them or just take them as a reference. There can be no absolute answers here.

We hope that this chapter has succeeded in scoping the topic as it applies to Java ME on Symbian OS. Having a single common Java ME platform for many Symbian OS UI platforms considerably reduces the problems that plague other platforms, resulting in diversity differences rather than fragmentation. You cannot guarantee all devices out of the box; no one can. However, with some detection, adaptive code and flexible design in your hands, providing a single JAR for the majority of Symbian smartphones is a very realistic proposition.

5

Java ME SDKs for Symbian OS

There are many tools for Java development that have a power and quality that supersedes tools for other development technologies. Java ME development tools leverage the existing Java development tools by integrating Java ME software development kits (SDKs) with the standard Java tools and IDEs. That makes the job of Java developers probably more comfortable and productive than that of other developers.

Symbian OS provides a common Java ME platform for many phone manufacturers but there is no single common Java ME SDK for Symbian OS devices. Instead, each of the phone manufacturers ships its own SDK. The difference between the various SDKs might not be that big. When choosing an SDK, the target devices are a primary factor, along with the number or quality of the accompanying tools and documentation.

Phone manufacturers create a Java ME SDK that includes an emulator which can be integrated with standard Java IDE (using the Unified Emulator Interface). They complement the SDK with additional tools, documentation and examples and take steps to ensure that Java applications can be developed easily for their platform.

As a developer, you can pick from a number of SDKs. This chapter discusses the various phone manufacturers' Java ME SDKs, providing a high-level overview; you can find more detail in the SDKs' manuals and documentation. We assume that you have installed an SDK and used NetBeans or Eclipse in previous Java development so we do not cover installation or running a first program. Instead, we indicate the UI platform for which the SDK is used, whether it is a true emulation of Java ME on Symbian OS or is based on the Java Wireless Toolkit (WTK), features and tools in the SDK, features in the emulator and other interesting things you can find in the SDK from the point of view of Java development targeting Symbian OS devices.

We recommend that after you read the chapter and get a general idea about the variety of available SDKs, you download the SDK which is most appropriate for your project and start experimenting with it.

5.1 Recommended Tooling Approach for Java ME on Symbian OS

Because of the variety of available SDKs for Java ME on Symbian OS (see Figure 5.1), we start with a general discussion on which SDKs to use and when to use them.

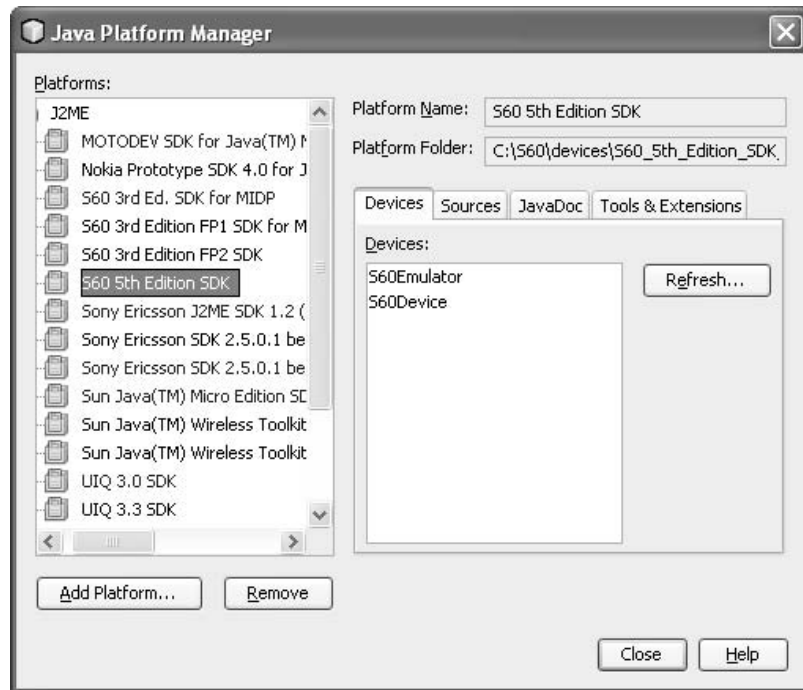


Figure 5.1 Java ME SDKs for Symbian OS

We recommend that you become familiar with several of the SDKs that are covered in this chapter and use the SDK which is most appropriate according to the development stage you are at or the problem you are trying to solve. You could use the WTK, Java ME SDK 3.0 (the WTK successor), Java ME SDKs specific to Symbian OS, your own utilities and, if you develop with C++, you could even get familiar with native development tools for Symbian OS.

The question that you should ask yourself is "Am I using the full breadth of available tools, at the right times?" If you have a big enough toolbox, you might find a tool that you have never used before is just what you need at a particular time. It is safe to assume that most of the time you will use the WTK or its successor, the Java ME SDK 3.0, in conjunction with an SDK specific to a Symbian OS UI. You will connect to the device using

a vendor connectivity suite; use the SDK emulator utilities that apply to JSRs used by your application; use on-device debugging with a Symbian smartphone; add some of your own utilities and, if a very specific need arises, you might resort to a native Symbian OS tool (e.g., Nokia Energy Profiler, native CPU profiler or file explorer). Having a variety of tools at your disposal should help you to develop rapidly for Symbian OS devices.

It is a small thing, but we recommend that you assign one or two days to learn how to use the tools as early as possible, even before a project starts. Learning tools requires a certain amount of time, which might not be available when you get closer to the deadline of a demanding customer.

To help you apply the above approach, we consider various tools and discuss their applicability to different needs so that you can make your own decision on which tool is most suitable for you, in any given situation.

5.2 Generic SDKs: Java ME SDK 3.0 and WTK 2.5.2

At time of going to print, we are welcoming Java ME SDK 3.0 (see Figure 5.2). Although it is too early to say farewell to the WTK, which has been the most popular SDK for quite a few years, let us meet the Java ME SDK 3.0 and go over some of its features.

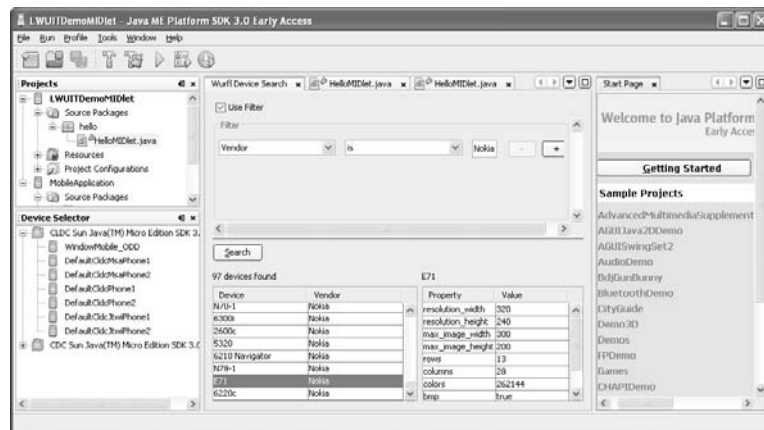


Figure 5.2 Java ME SDK 3.0

The Java ME SDK comes with a new underlying architecture and lots of new features such as:

- full MSA 1.1 stack
- integration with third-party emulators and devices

- on-device debugging (ODD) support
- integration with LWUIT
- WURFL-device search database
- Windows Mobile reference stack with ODD support
- a new framework based on the NetBeans Platform.

What does this mean for Java developers? It means that they can still use the WTK as before but they now have a much more powerful development tool.

What does this mean for Java developers targeting Symbian OS? For many Java ME platforms, development using the WTK or Java ME SDK 3.0 (integrated with an IDE, e.g., NetBeans or Eclipse) is the only option. This approach has the benefit of using standard tools across different devices from different vendors. There are also many good development and monitoring tools (e.g., memory monitoring and network monitoring) available in the Java ME SDK 3.0 which are recommended.

The disadvantage of using generic tools is that the emulation gives you the same Java ME standard JSRs but the execution environment and underlying implementation are different from Java ME on Symbian OS. The Java APIs are obviously the same APIs, with the same methods signature, and are compliant with TCK (a suite of tests, tools and documentation that determines whether or not a product complies with a particular Java technology specification). However, the underlying behavior is different because the Java ME implementation stack that runs on Symbian OS devices and the WTK emulation are two different stacks; this might lead to different behavior between execution on a real device and execution using the generic SDK.

Additionally, a generic SDK environment can only run Java applications – it is not a full-blown phone stack. More specifically, it is not a multi-process environment that hosts multiple run-time environments, unlike Symbian OS. That obviously limits what can be tested using the emulator.

What you get from using the Java ME SDK 3.0 and the WTK is a generic SDK that lets you do most of the development and gives you some excellent tools that you should use as needed. However, we strongly recommend that in conjunction with a generic SDK you use a Symbian OS SDK that is applicable to your project.

5.3 SDKs for the S60 3rd Edition and 5th Edition Platforms

S60 Platform SDKs for MIDP enable Java developers to quickly and efficiently run and test Java applications for devices built on the S60 platform.

The SDK delivers all the tools required to build Java applications and includes an S60 device emulator, various tools, Java API implementations, documentation, and example applications. The S60 SDK is a hybrid: it supports both native Symbian OS and Java development, which is an additional benefit if you are dealing with both.

The S60 Platform SDKs for Symbian OS enable you to take full advantage of the features of the NetBeans IDE with Mobility pack. The first thing to do is to find which SDK is relevant for your target device.

Within the S60 platform, there are various editions (significant updates in Symbian OS functionality) and feature packs (FP). We focus on S60 5th Edition and S60 3rd Edition, which are based on Symbian OS v9.x and MIDP 2.x. Feature packs are incremental improvements within an edition, adding, for example, new JSRs or improving the supported features within an existing JSR. Each combination of edition and feature pack has its own SDK.

To find out which platform, edition, and feature pack your device uses, consult the Forum Nokia Device section at www.forum.nokia.com/devices and check the developer platform section of the technical specifications. Once you know the version and feature pack of your device, download the appropriate SDK. For example, the Nokia 5800 Xpress-Music is based on S60 5th Edition, the Nokia N73 is based on S60 3rd Edition initial release, the Nokia N95 is based on S60 3rd Edition FP1, and the Nokia N96 is based on S60 3rd Edition FP2.

The S60 emulator (see Figure 5.3) provides a *true* emulation of Symbian OS and the MIDP environment delivered on S60 devices, enabling

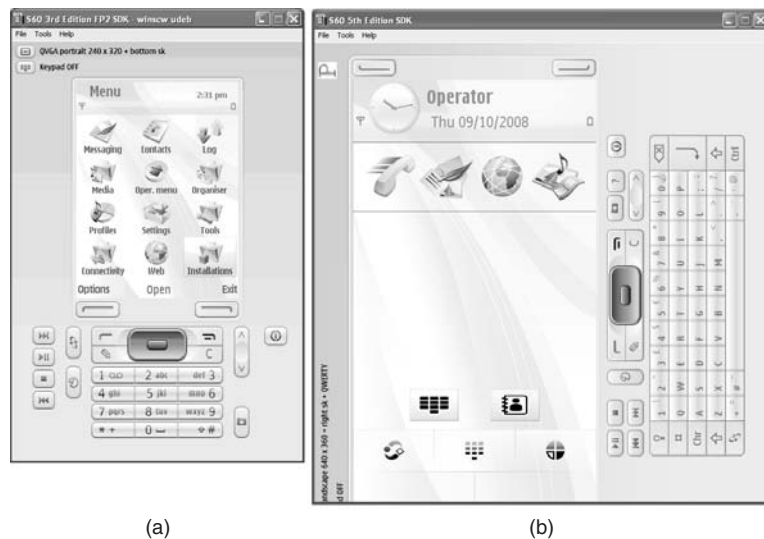


Figure 5.3 Emulators for a) S60 3rd Edition FP2 and b) S60 5th Edition

applications to be better tested on a PC. The emulator mimics the operation of a real S60 device so accurately that applications can be developed even before the required S60 device is available.

The S60 emulator allows you to switch between concurrently running native and Java applications (see Figure 5.4), change the device settings as you would on a real device, use the native S60 browser, insert contacts to the native contacts database, manipulate files on the emulator file system, and much more. This is a powerful feature that cannot be matched by a generic emulator: it is as close as possible to a real S60 device.



Figure 5.4 Using the Task Manager to switch between applications

5.3.1 Running MIDlets on the S60 Emulator

When you launch the MIDlet from NetBeans, the progress dialog in Figure 5.5 appears. If there is more than one MIDlet in your suite, you are asked to choose one of them (see Figure 5.6). The S60 emulator starts an application called DebugAgent whose role is to manage debug sessions. The DebugAgent display shows the launch progress (see Figure 5.7) until the MIDlet becomes active and is sent to the foreground.

In order to speed up the development cycle, you can keep the emulator running throughout your development session; you do not need to restart the emulator for every debug session. The first time you launch an application from NetBeans, the emulator starts the Java DebugAgent running to handle the installation and lifecycle of debugged MIDlets. After terminating the debug session, you can leave the emulator running and the DebugAgent waiting for the next MIDlet launch. Rebuild the debugged application from the IDE and re-launch the application in the emulator. To stop the execution of your application without closing the S60 emulator, press Abort on the progress dialog.



Figure 5.5 Launch progress window

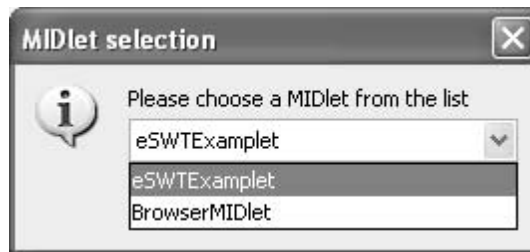


Figure 5.6 Choose a MIDlet from the suite

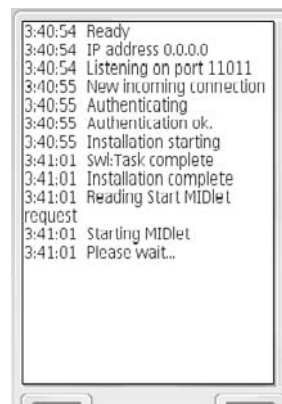


Figure 5.7 DebugAgent display

You can also deploy and run MIDlets manually. You may copy a suite JAR and JAD to the S60 emulator file system to be installed from the S60 Application Manager (see Figure 5.8). Copy the binaries to the SDK Installs directory: `\epoc32\winscw\c\Data\Installs\`. From the main menu in the emulator, select Installations, App. mgr., Installation files and follow the installation process. After the installation is complete, you may launch the MIDlets from the location in which you find all the installed applications.

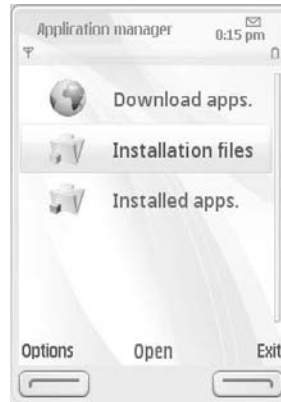


Figure 5.8 Installing a suite from the file system

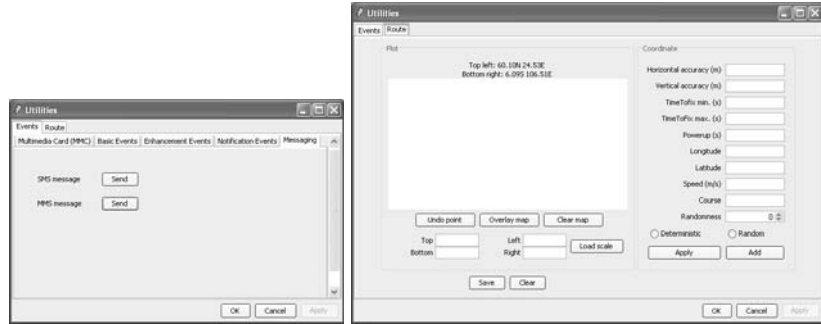
5.3.2 Features of the Emulator and SDK

The S60 emulator is rich in features which you are likely to find useful at different stages and for development using various JSRs. You can switch screen orientation between portrait and landscape modes and you can use the File menu to open various kinds of file.

The emulator's Utilities window contains tools that emulate run-time behavior. It provides a user interface for simulating various different kinds of phone events through the Events tab (see Figure 5.9a), such as incoming phone calls, MMC simulation, grip events, battery events, enhancement events, notification events, SMS and MMS messaging events. The Route tab (see Figure 5.9b) is for developing location-based applications with the SimPSY plug-in. To open the Utilities window, select Tools, Utilities from the emulator menu.

The diagnostics and tracing functionality provided with the SDK enables you to monitor the activities of applications running on the S60 emulator (and on a real device) and resource utilization, such as CPU and memory. To open the Diagnostics window (Figure 5.10), select Tools, Diagnostics from the emulator menu.

In the Preferences window (Figure 5.11), you will find the MIDP Security tab which is used to adjust the MIDlet run-time security sandbox on



(a) (b)

Figure 5.9 Utilities window in the emulator: a) Events tab and b) Route tab

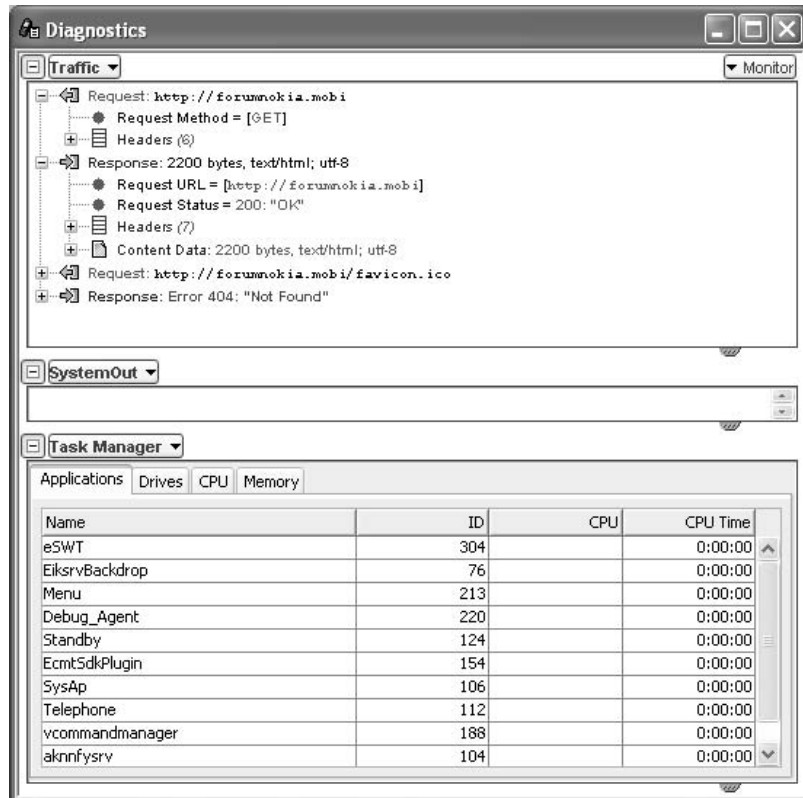


Figure 5.10 SDK Diagnostics window

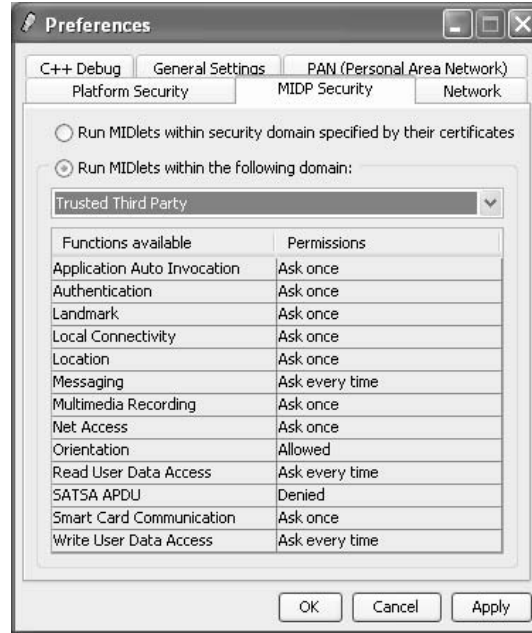


Figure 5.11 Preferences window

the emulator. You can run MIDlets within the security domain specified by their certificates (the default) or select the security environment. To open the Preferences window, select Tools, Preferences from the emulator menu. You can launch the Preferences window from NetBeans by selecting Java Platforms, S60 SDK, Tools & Extensions, Open Preferences.

5.3.3 Using On-Device Debugging

The high-level setup instructions are to connect the device and the PC (by Bluetooth or WLAN), start the DebugAgent on the device, start the Device Connectivity Tool, and run and debug from the IDE just as you do on the emulator. (Depending on the S60 SDK version, the DebugAgent can also be called EcmtAgent.) You set up the DebugAgent in the following way:

1. Install the DebugAgent to your S60 device. The DebugAgent SIS file is delivered with the SDK and is located under the `\S60tools\Ecmt\` directory.
2. Launch the DebugAgent on the S60 device (see Figure 5.12).
3. Select the DebugAgent connection method: select Options, then Settings. In the Bearer view that is displayed, select Options, Change to set the connection method (see Figure 5.13).

4. Start the Device Connectivity Tool on your PC (Start, Programs, S60 Developer Tools, 5th or 3rd Edition SDK, MIDP, Tools, Device Connection).
5. Establish a connection between the S60 device and the Device Connectivity Tool (see Figure 5.14).



Figure 5.12 DebugAgent running on device



Figure 5.13 Choosing connection method

You now have a Bluetooth connection between the PC and the device. You need to set the device in the project properties (see Figure 5.15) to S60Device (or S60Device_over_Bluetooth if it is an older SDK).

From here, you launch and debug an application exactly as on the emulator. The same progress console appears on the PC and progress on the device appears on the DebugAgent display.

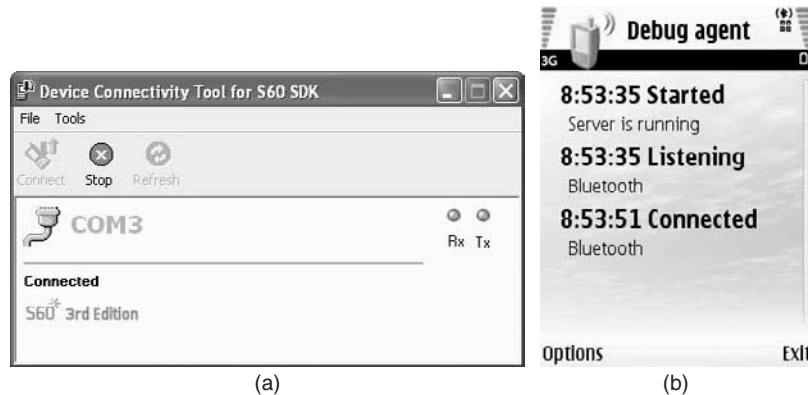


Figure 5.14 Establishing a connection

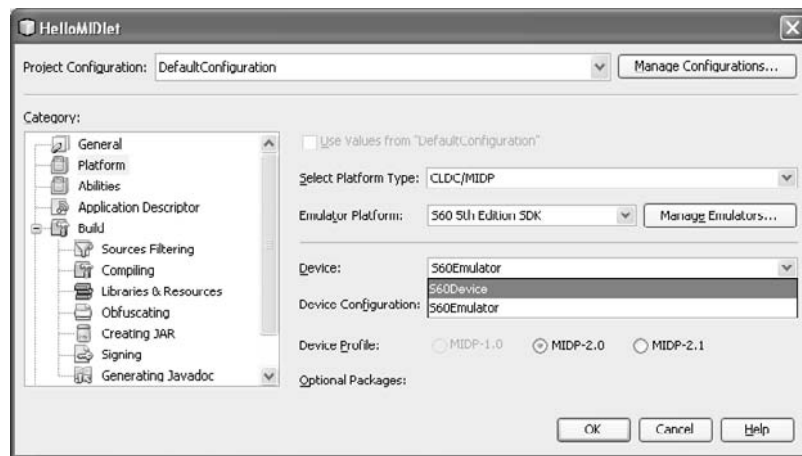


Figure 5.15 Choosing on-device debugging in NetBeans

Please note that setting up the connection between the PC and device is dependent on your machine configuration (e.g., the Bluetooth drivers). If you have configuration issues, you should consult your system administrator. For more detailed instructions and how to set up the connection over WLAN, please refer to the S60 SDK user guide document (located under [SDK_HOME]\docs).

The diagnostics and tracing functionality provided with the SDK enables you to monitor the activities of an application on a real S60 device. Monitoring the activities of an application on an S60 device takes place in the DebugAgent. You establish a connection between the PC and the device, just as when setting up on-device debugging, and then monitor the activities through services provided by the Diagnostics window.

5.3.4 Other Tools

Under the SDK installation directory, you can find a wealth of documentation and code examples. We recommend that you spend some time getting familiar with all the features of the SDK, so that you are aware of the power that is given to you as a developer.

You may also download as a separate bundle the Java ME Developer's Library,¹ a complete resource package with introductory documents, getting started guides, tutorials, and API level information. This library is available online in HTML format, as a standalone version and as an Eclipse plug-in (see Figure 5.16).

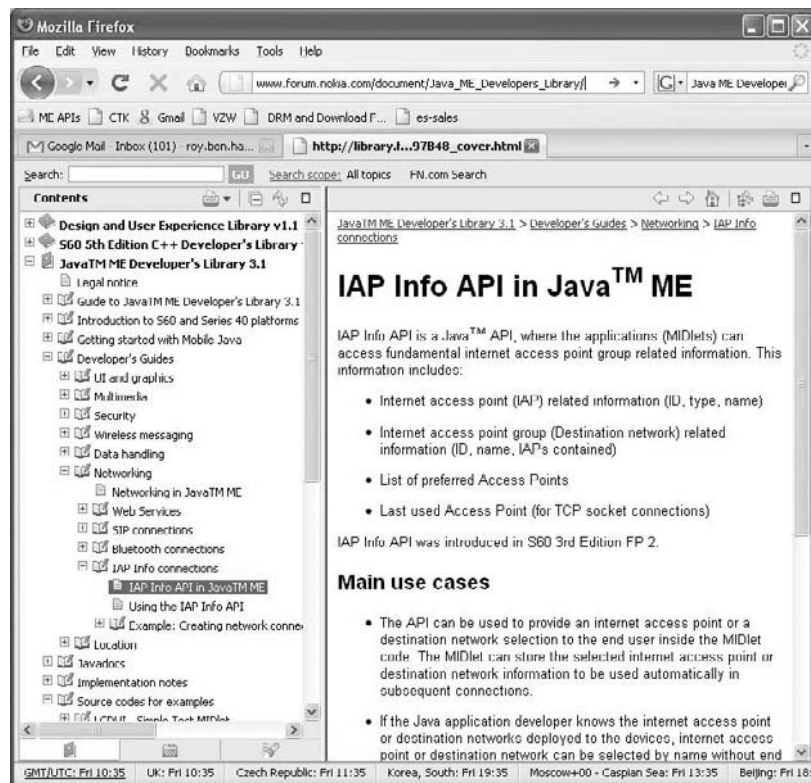


Figure 5.16 Java ME Developer's Library

The SDK contains more tools for Java development for S60. For example, it includes an SVG to SVG-T Converter that allows you to optimize scalable vector graphics for the S60 user interface by enabling you to convert SVG images to SVG-Tiny (SVG-T) format.

¹ www.forum.nokia.com/document/Java_ME_Developers_Library

You can download other tools (such as, the SNAP Mobile SDK discussed in Appendix B) from Forum Nokia. For example, in order to help you profile the power consumption of your applications, Forum Nokia (forum.nokia.com) has released Nokia Energy Profiler, compatible with devices from S60 3rd Edition FP1. Install Nokia Energy Profiler, start it, send it to the background, then start your application and switch back to the Profiler to see how much the consumption has increased. You can then fine-tune the application so it uses low-power algorithms and routines. Figure 5.17 shows Nokia Energy Profiler consumption graphs for the resource-intensive JBenchmark application (see Chapter 9).

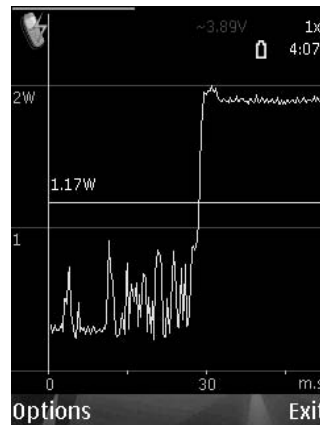


Figure 5.17 Nokia Energy Profiler

The Nokia PC suite is not a developer tool, but can be integrated with NetBeans to allow you to deploy Java applications from NetBeans (see Figure 5.18). Download the PC suite that supports your device from

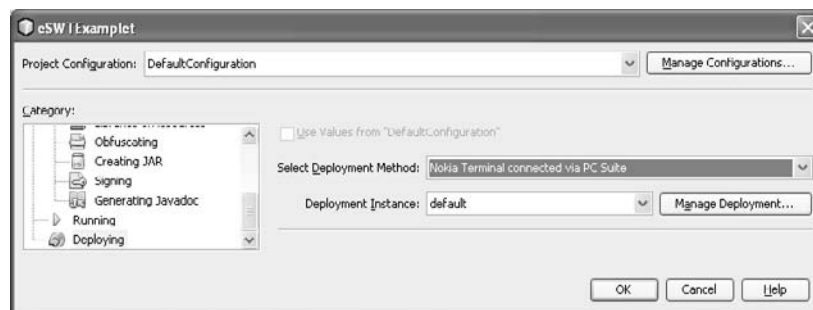


Figure 5.18 PC suite deployment from NetBeans

www.nokia.com/pcsuite and connect to the device. From NetBeans, open the project's Properties dialog. In the Deploying tab, select 'Nokia Terminal connected via PC Suite' as the deployment method. To deploy the application, right click on the project and select Deploy. The application installation on the device starts immediately.

5.3.5 Review

Let's summarize what we have said about the S60 SDKs. We covered the main features of the emulator, the tools that can facilitate development using different JSRs, the use of on-device debugging and diagnostics, and gave pointers to additional tools and resources.

It takes great effort to produce a good SDK and Nokia has certainly succeeded in producing a powerful emulator and a feature-rich SDK. These tools ensure a good developer experience and productive development targeting S60 devices.

5.4 SDKs for the UIQ 3 UI Platform

UIQ device manufacturers may supplement the UIQ 3 Java ME implementation with additional JSRs, proprietary APIs, and their own development tools and SDKs. Developers can take advantage of those opportunities by developing with the additional APIs and using the device manufacturer SDK, tools and add-on packs.

We recommend that you use an SDK from a UIQ device manufacturer. The advantage is that you will benefit from any enhanced toolset, additional development tools and additional APIs provided by the SDK. The drawbacks are that it may provide WTK-based emulation, rather than a Symbian OS emulator. The emulator might look like a Symbian OS device but it is the same emulator as in the WTK with a different skin. True UIQ 3 emulation harnesses the Symbian OS emulator and gives an execution environment as close as possible to a real UIQ 3 device.

Each UIQ version represents significant updates in Symbian OS functionality (i.e., UIQ 3.x equates to Symbian OS v9.x). Within the UIQ 3 platform, there are various minor versions. To find out which UIQ version your device uses, consult the list at **uiq.com/phones**; the Sony Ericsson W960i is a UIQ 3.1 device and the Motorola Z10 is a UIQ 3.2 device.

Once you know the UIQ 3 version, download either the UIQ 3 SDK or the relevant phone manufacturer SDK. We first give an overview of the generic UIQ 3 SDK before moving on to Java ME SDKs for UIQ 3 devices from Sony Ericsson and Motorola.

5.4.1 Generic UIQ 3 SDK

The primary role of the UIQ 3 SDK for Symbian OS is for Symbian C++ development. However, there are two key points that make it very relevant and useful for Java development. The UIQ 3 emulator gives full-blown Symbian OS emulation, which allows you to install, run and manage Java applications as on a real UIQ 3 device. Additionally, the UIQ 3 SDK includes a basic SDK that is compliant with the Unified Emulator Interface (UEI) and can be integrated with NetBeans, allowing you to develop, run and debug MIDlets using the UIQ 3 emulator.

If you do not have the SDK installed, you may download it from **developer.uiq.com/devtools_uiqsdk.html**. (You need to be a registered member of the UIQ Developer Community to be able to download it.)

The major benefit that the UIQ 3 SDK gives you is a single Java ME execution environment which is compatible across the various UIQ 3 platforms (newer UIQ 3 versions may contain additional JSRs; e.g., the UIQ 3.3 SDK supports more JSRs than the UIQ 3.1 SDK). The SDK also gives you real emulation of Symbian OS: for example, you may launch other native applications in the background while you are running your MIDlet, change the device settings, and so on, just as you would do on a real device.

Deploying and Running MIDlets from the Emulator File System

You first need to copy the JAR and JAD to the emulator file system so that you can use the UIQ emulator File Manager to install and run the suite.

1. Launch the emulator from [SDK HOME]\epoc32\release\wincsw\udeb\epoc.exe.
2. In Windows, copy the suite JAR and JAD to the UIQ 3 emulator file system under [SDK HOME]\epoc32\wincsw\C\Media files\other\.
3. In the UIQ 3 emulator, go to Main menu, Tools, File manager and browse to \other\ (see Figure 5.19).
4. Follow the installation process which will eventually start the MIDlet, in this case, the APISDetector MIDlet (see Figure 5.20).

After installing a suite, you can run the MIDlets again from the same location as all installed applications (see Figure 5.21).

As you can see from Figure 5.20, the UIQ 3.3 emulator supports the full set of MSA Component JSRs. In the emulator, you can also find all the Java management operations that are on a real device, run native applications concurrently, switch between applications (see Figure 5.22), and so on.



Figure 5.19 Installing and running from the UIQ emulator file system

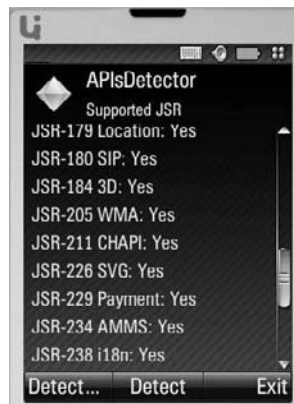


Figure 5.20 The APISDetector MIDlet on UIQ emulator



Figure 5.21 Running a MIDlet on UIQ emulator



Figure 5.22 The Task Manager in the UIQ emulator

Deploying and Running MIDlets from NetBeans

Now we can integrate the NetBeans IDE with the UIQ 3 SDK emulator (which can be integrated with any UEI-compliant IDE).

1. In NetBeans, select Tools, Java Platform Manager. Select the J2ME tab and click on Add Platform.
2. Select the Java Micro Edition Platform Emulator, click Next and click Find More Java ME Platform Folders. You are then presented with the directory browser.
3. Browse to the `\epoc32\tools\java\sei\wincsw\udeb\hmidp92\` folder under the UIQ 3 installation directory, and click on Search. The directory is added to the list of folders under which NetBeans looks for a UEI-compliant SDK.
4. Check the appropriate SDK box, click Next and allow NetBeans to detect the SDK (see Figure 5.23).
5. Click Finish to complete the integration of the UIQ 3 MIDP SDK with NetBeans.

When you create a new Java ME MIDlet project in NetBeans and you reach the Default Platform Selection page (see Figure 5.24), select the UIQ 3 SDK (named Symbian 9.2 SDK) and win32 as the device (this means that you are running the emulator on Windows). You can then click on Finish to complete the setup of the new project.

You may launch the emulator from the Windows program menu. Once it is running, you can start the DebugAgent from the applications menu (see Figure 5.25a). You can also launch the DebugAgent from NetBeans. Click Tools, Java Platform Manager. Select the Tools and Extensions tab

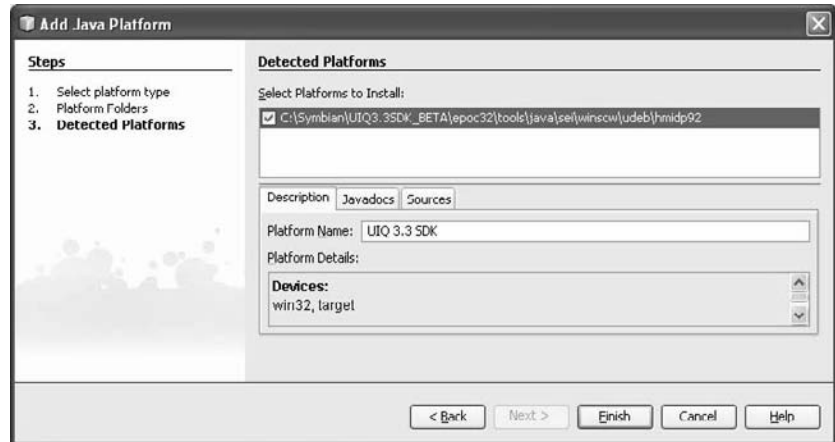


Figure 5.23 Integrating UIQ 3 SDK with NetBeans

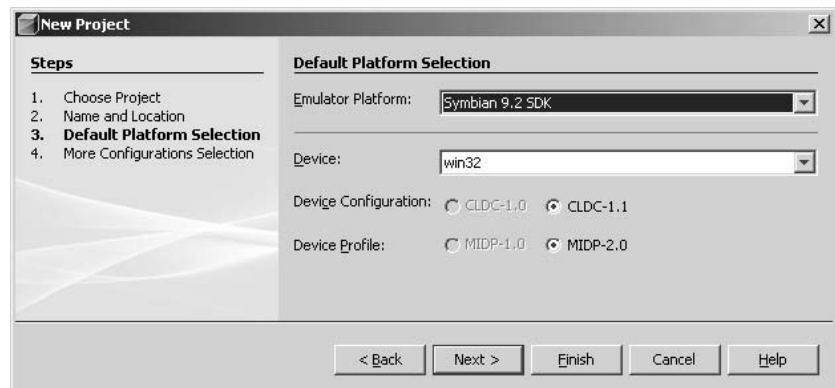


Figure 5.24 Setting UIQ 3 SDK as the default platform

in the UIQ 3 SDK left panel and click Open Utilities. You may launch the emulator and DebugAgent from the Launch Now button.

Ensure the DebugAgent is running and is ready to receive incoming connections.² From the DebugAgent menu, click on More, Start. You will see logs that indicate that the agent is ready to receive incoming requests (see Figure 5.26).

You only have to start the emulator and the DebugAgent the first time you launch a MIDlet. After terminating your MIDlet, the emulator keeps running and subsequent launches take less time to start up. You may then

² Refer to FAQ-117 in www.developer.uiq.com for information on how to enable a TCP server running on the emulator to receive incoming connections.



Figure 5.25 Launching the Java DebugAgent a) from the applications menu and b) from UIQ 3 SDK utilities



Figure 5.26 Starting the Java DebugAgent server

launch the MIDlet, from NetBeans, which pops up the progress dialog shown in Figure 5.27.

To stop the execution of your application, click Abort on the progress dialog. The emulator and the DebugAgent keep running, waiting for the next MIDlet launch request. Do not close the emulator during your development session, to allow quick launching of MIDlets.



Figure 5.27 MIDlet launch progress window

There are no other Java development tools in the UIQ 3 SDK. The role of making Java ME SDKs for UIQ 3 is in the hands of phone manufacturers, who usually have their own developer programs and development tools for their various platforms. Keep in mind that having a true emulation of a Symbian OS device is a major advantage; that alone makes the UIQ 3 SDK a powerful tool.

5.4.2 SDKs for Sony Ericsson SJP-3

Sony Ericsson's Java strategy has adopted a platform approach with two branches: SJP supports phones that use Symbian OS and JP supports phones that do not use Symbian OS. Each Java Platform is used in several phone models to help developers focus on a family of devices rather than on a variety of different products. At the time of writing this book, there are three SJP platforms (SJP-1, SJP-2 and SJP-3), which are implemented through an evolutionary approach in order to ensure forward compatibility between platform versions. The Sony Ericsson SJP-3 SDKs for the Java ME Platform include support for JSRs and proprietary APIs available on Sony Ericsson UIQ 3 phones. For example, the SJP-3 SDK supports all the UIQ 3.0 MIDP JSRs and additional APIs such as JSR-205 (WMA 2.0), JSR-226 (SVG), and the Nokia UI API.

The Sony Ericsson Java ME 2.5.0.1 SDK is a modified version of the Sun Microsystems WTK. After installation, there are two subdirectories, WTK1 and WTK2, which correspond to WTK 1.04.01 and WTK 2.5.0 respectively. The WTK2 subdirectory supports Symbian devices, such as the Sony Ericsson W950, and some Sony Ericsson JP devices. You can choose to use the generic SJP-3 profile emulator (named SonyEricsson_SJP3_240x320_Emu) or an emulator for a specific UIQ 3 device (e.g., SonyEricsson_W950_Emu). There is also support for UIQ

2 devices. As a developer this allows you to concentrate on two variables: the platform and the screen size.

An additional subfolder, `\OnDeviceDebug\`, contains the support for on-device debugging.

The SDK is UIE-compliant and therefore the integration with NetBeans is performed similarly to the integration with the WTK or the UIQ 3 SDK (note that the SDK location is `[SDK path]\PC_Emulation\WTK2\`).

After you have installed and launched a MIDlet, you can emulate pausing and resuming the running MIDlet and change the device orientation. As in the WTK, from the emulator you can manage external events, such as location, file connection and payment transactions. You can also take a snapshot of the LCDUI display area, which could be helpful in closely examining the application UI during development.

The SDK emulator provides a rough approximation of the Java ME platform on real devices. It is not a true emulation of Symbian OS; generally, the emulated navigation and input behavior does not exactly correspond to the actual device (see Figure 5.28).



Figure 5.28 Sony Ericsson SDK with WTK-based emulation

From the SDK (or from the NetBeans Java Platform manager, after integration), you can launch the WTK utilities and preferences tools (see Figure 5.29).

The SJP-3 SDK has powerful integration with real UIQ 3 devices: on-device debugging, Device Explorer and the Sony Ericsson PC suite for Smartphones.

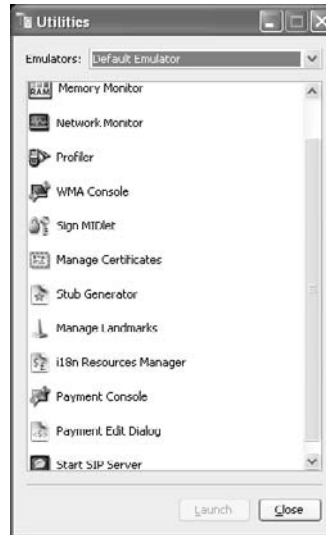


Figure 5.29 Sony Ericsson SDK utilities

To use on-device debugging and the Device Explorer, we need to set up a two-layer connection. The first layer is the Bluetooth layer (physical layer); the second layer is the TCP layer (network layer).

1. Install `\JavaME_SDK_CLDC\OnDeviceDebug\bin\JavaDebug-Agent.SIS` on the device.
2. On the device, go to Control Panel, Internet Accounts and open Ad-Hoc Bluetooth PAN. In the account, select More, TCP and set up the TCP address and subnet mask to be used (see Figure 5.30a).

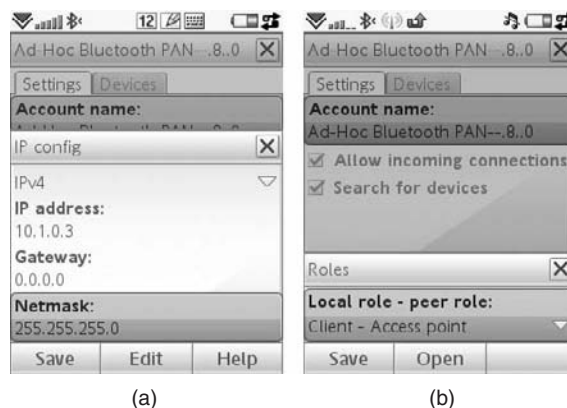


Figure 5.30 Ad-Hoc Bluetooth PAN setup

3. Select More, Roles and configure the role as Client - Access point (see Figure 5.30b).
4. On the PC, go to My Bluetooth places and select View my Bluetooth services. Right click Properties on My Network Access. In the General tab (Figure 5.31a), select Allow other devices to access the Internet/Lan via this computer.
5. Select Configure Network Adapter. In the Internet Protocol (TCP/IP) properties sheet (Figure 5.31b), assign a different IP address in the same subnet and assign the same subnet mask.

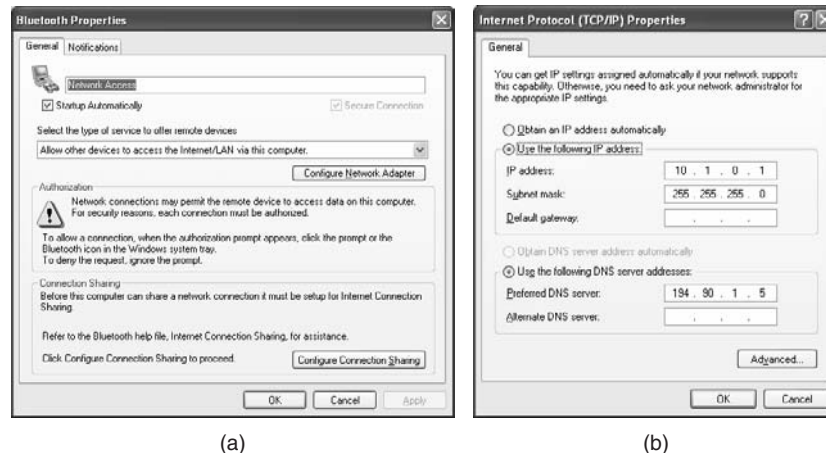


Figure 5.31 PC Network Access setup

6. From My Bluetooth Places, go to Devices in range and select your device. Right-click and select Connect Personal Ad-hoc User Service. Click on Connect.
7. Launch the JavaDebugAgent. In the Settings sheet (Figure 5.32a), assign the PC host IP address.
8. From the main screen, start the server (Figure 5.32b).
9. Open the SDK Connection Proxy utility and press Connect (see Figure 5.33).

Once the connection is established, in the NetBeans project properties (see Figure 5.34), select the Debug On-Device SDK (it is a separate SDK) and SonyEricsson_SJP-3 as the device.

After you have completed the setup, launching and debugging an application is done in exactly the same way as on the emulator. Please note that setting up the connection between the PC and the device is

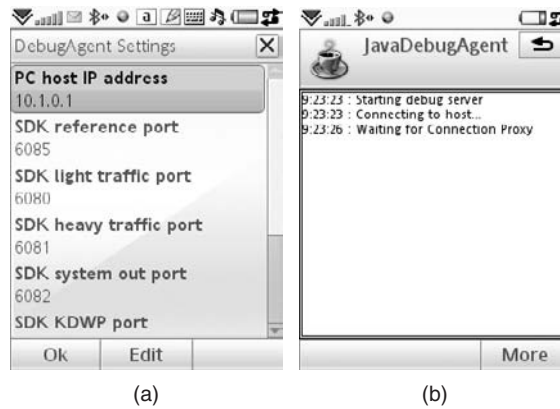


Figure 5.32 JavaDebugAgent setup

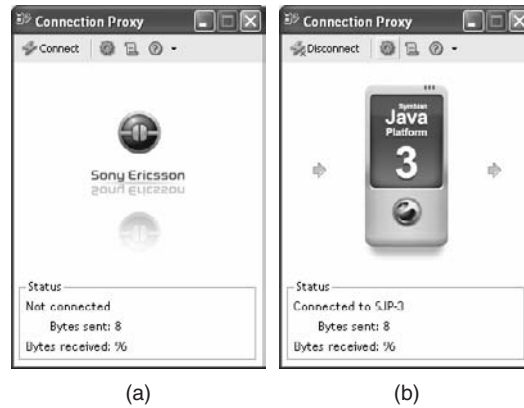


Figure 5.33 Connecting the proxy

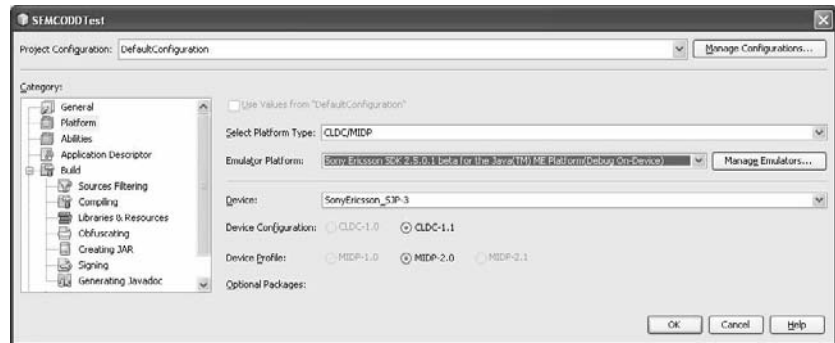


Figure 5.34 NetBeans project setup

dependent on your machine and device configuration. In the case of configuration issues, you should consult your system administrator.

With the SDK's Device Explorer tool (see Figure 5.35), developers can directly manipulate MIDlets and suites from the PC. This includes installation; starting and stopping the execution; getting storage information; and examining trace messages coming from `System.out`.

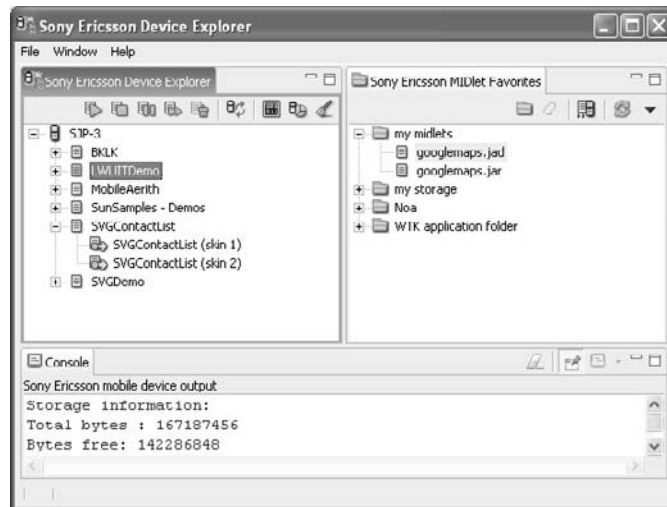


Figure 5.35 Device Explorer utility

The setup and usage of Device Explorer is done using the same mechanism as that used by on-device debugging:

1. Turn on Bluetooth.
2. Pair device with PC.
3. Start the JavaDebugAgent server.
4. Connect the Connection Proxy tool.
5. Launch Device Explorer.

Finally, the Sony Ericsson PC suite for Smartphones (see Figure 5.36) is not a developer tool but it is a useful tool to have. Its features include browsing and exchanging files between the PC and device; installing MIDlet suite and native applications on the device; and updating your phone with the latest firmware.

For more information and to download SJP SDKs, please refer to Sony Ericsson Developer World.



Figure 5.36 Sony Ericsson PC suite for Smartphones

5.4.3 Motorola MOTODEV Studio

MOTODEV Studio for Java ME (see Figure 5.37) provides an Eclipse-based IDE, enhanced with EclipseME. MOTODEV Studio comes with MOTODEV SDK for Java ME, to let you access all the tools and documentation you need from one central place. MOTODEV SDK for Java ME

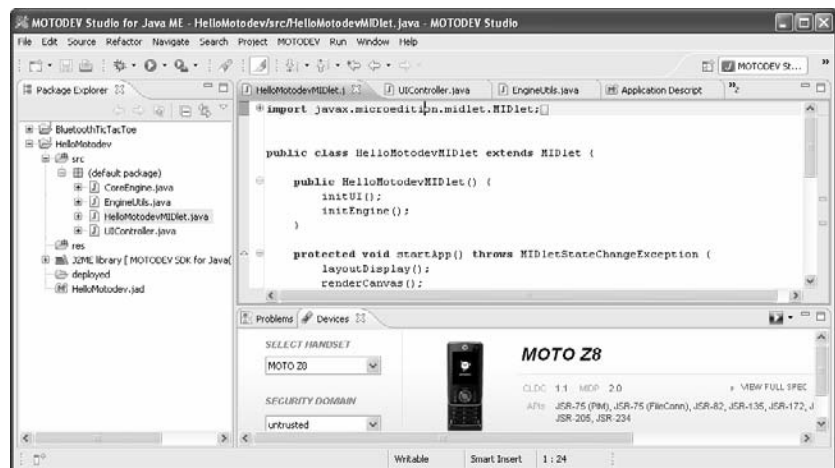


Figure 5.37 MOTODEV Studio



Figure 5.38 MOTODEV emulator with MOTO Z10 skin

version 1.3 supports Motorola UIQ 3 handsets, such as the MOTO Z8 and MOTO Z10 (UIQ 3.1 and UIQ 3.2, respectively).

The MOTODEV SDK Java ME emulator uses an emulation of the MIDlet environment in Motorola handsets and UIQ devices that is based on Java SE, with an interactive ‘skin’ of the handsets (see Figure 5.38). Standard mobile phone features are not supported and it is not a true emulation of Symbian OS. The MOTODEV emulator includes features such as:

- mounting file system roots and specifying the size and capacity
- setting networking configuration
- clearing all Contacts, Events and ToDo PIM entries
- simulating external events such as an incoming call
- clearing all RMS data.

From the main menu of MOTODEV Studio, select Window, Show View, MOTODEV Studio for Java ME, Devices to access Java ME development tools (see Figure 5.39) and supporting services such as:

- Application Signing Tool – automates signing of MIDlet suites
- Bluetooth Remote Control – allows handsets to be controlled using a remote Bluetooth device

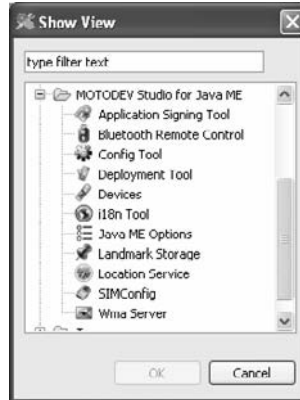


Figure 5.39 MOTODEV Studio tools and services

- Config Tool – supports reading and writing internal Motorola handset configurations
- Deployment Tool – enables deployment of Java ME applications into the device
- i18n Tool – supports internationalization of application content, such as messages and images
- Landmark Storage – supports management of landmark information for the Location API
- Location Service – allows route simulation
- SIMConfig – allows viewing of specific SIM card information
- WMA Server – provides a method for sending and receiving SMS messages between two emulated devices.

How to use each of the SDK tools and services is clearly explained in the SDK documentation which is available from the Help menu.

MOTODEV Studio documentation also includes valuable information on target devices. If you need to know some details about a certain model, for example, MOTO Z10, you can easily get that information without leaving MOTODEV Studio. From the main menu, select Window, Show View, Devices (see Figure 5.40).

To find more information on the device, click on VIEW FULL SPEC; a help window opens with the key features, technical specification and Java ME information of the specific device (see Figure 5.41).

From the Devices view, click on VIEW DEVICE FEATURE MATRIX to compare devices; a window opens with the Motorola device matrix that shows the relationship between the devices and information on each



Figure 5.40 MOTODEV Studio view of devices: MOTO Z10



Figure 5.41 MOTODEV SDK full device specification: MOTO Z10

device such as device family, software platform, supported JSRs and Motorola APIs.

The MOTODEV SDK for Java ME is also available as a separate unit that can be integrated with any UEI-compliant IDE, such as NetBeans. It is also possible to use the SDK as a standalone application, named Launchpad (see Figure 5.42), which gives you access to the same documentation, tools and services as in the MOTODEV Studio integrated version. However, the standalone MOTODEV SDK does not include full support for applications development (e.g., compilation and packaging in the MOTODEV SDK are done from command line, while the MOTODEV Studio is a full blown IDE).

To download MOTODEV Studio for Java ME and MOTODEV SDK for Java ME, please visit the MOTODEV website at ***developer.motorola.com***.

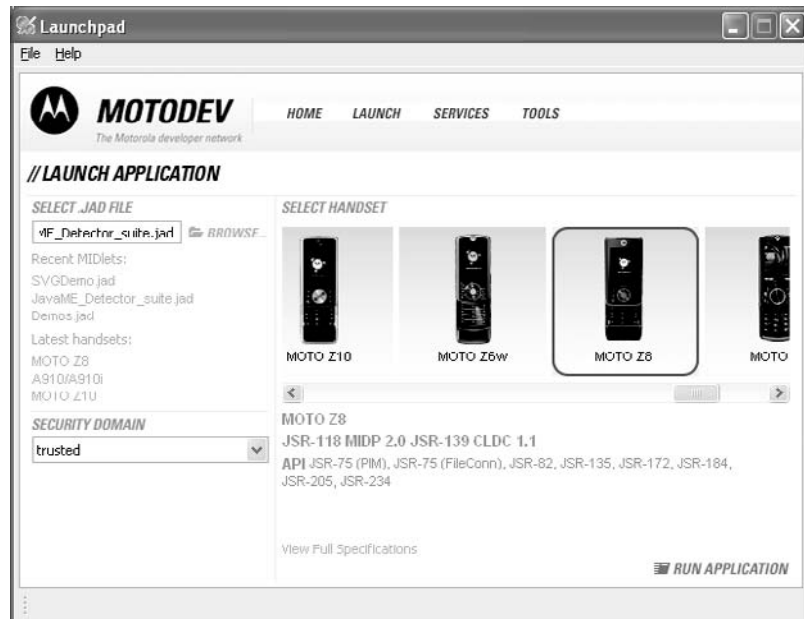


Figure 5.42 MOTODEV standalone SDK for Java ME

5.5 Summary

Let's summarize what we have covered in this chapter about Java ME SDKs targeting Symbian OS devices. For many Java ME platforms, development using the WTK or Java ME SDK 3.0 is the only option. Development with Java ME on Symbian OS offers developers many more SDK options, which target different Symbian OS UI platforms and device families. The approach which we recommend is to become familiar with several SDKs and tools; and use those best suited to the development stage you are at or the problem you are trying to solve.

We covered the SDK emulator, tools, on-device debugging and other features of S60 5th Edition and 3rd Edition FP2 SDKs, UIQ 3.3 SDK, Sony Ericsson SJP-3 SDK and MOTODEV SDK for Java ME version 1.3.

To download the latest SDKs and get more information on developer tools, refer to the developer program of the relevant device manufacturer. Chapter 7 gives information about MOAP development and SDKs.

Part Three

MSA, DoJa and MIDP Game Development

6

Designing Advanced Applications with MSA

JSR-248 Mobile Service Architecture (MSA) is another step in the overall vision of Java technology as a whole and, more specifically, the vision of Java on mobile phones.

We start the chapter by focusing on the MSA specification and trying to answer the major questions that a developer would ask – what is it and what does it consist of? A more pragmatic question – what can I do with it? – follows and we look at various applications that can benefit from MSA. To complement the list of possible applications, we take a specific example of a well-known application, and enhance it using MSA. Finally, we finish with a short section on the next stage of the MSA standardization effort.

6.1 What Is MSA?

It is very likely that you have already heard about MSA but, if you have not read the specification yet, there are many questions to be addressed: is it a new set of API packages? is it a successor to MIDP 2.0? and so on. The word ‘umbrella’ is often used to describe JSR-248 and it describes MSA quite well. Under the umbrella, there are familiar JSRs and no new APIs. The answer to the first question, then, is that MSA is not a new set of API packages; all the APIs scoped by MSA are defined in other JSR specifications. As to the second question, MSA is based on JSR-118 MIDP 2.1 but, unlike the previous MIDP specifications, it also includes other JSRs. Let’s now try to say what MSA is, in more depth and detail. Unless stated explicitly, the discussion in this chapter refers to version 1.1 of JSR-248 MSA.

JSR-248 defines a Mobile Service Architecture (MSA) that is a step in the evolution of Java on mobile phones. It builds a Java ME platform

that is rich in features and reflects the current market needs. MSA broadens the JTWI standard by standardizing support for technologies and features that are already available on many devices (and, specifically, on Symbian devices). The MSA environment is backwards compatible with the old JTWI. Due to the advancements in the mobile industry and Java specifically, MSA is aligning around a broader set of devices which have much more powerful capabilities. However, the intent is still to align on high-volume mobile devices and not to create a challenging standard which would be difficult for low-end devices to comply with. As you may expect, MSA is also designed to be future-compatible with the forthcoming MSA 2.0 environment. (Although an early draft of JSR-249 MSA 2.0 is available for public review at the time of writing, some of the Component JSRs are still in very early stages and therefore MSA 2.0 is not discussed here.)

The MSA specification's primary goal is to reduce fragmentation of mobile Java environments. An aligned mobile Java standard makes life easier for ISVs and developers by giving them a cross-device Java ME platform which is highly predictable and helps them to make sure that their applications work on a wide variety of MSA-compliant devices.

The basis of the MSA environment is the CLDC 1.1 Configuration and the MIDP 2.1 Profile. This means that familiar MIDP MIDlet applications can run on an MSA-compliant device. On top of the MIDP/CLDC environment, MSA defines a normative collection of what it calls 'Component JSRs': the JSRs that are referred to in the JSR-248 specification and which live under the MSA umbrella.

Looking at MSA only as a collection of JSRs is misleading. There are other important elements in JSR-248, such as clarification requirements in the Component JSRs to improve predictability and interoperability. MSA also includes additional new mandatory requirements related to JTWI, security, supported content formats, and so on. MSA also adds optional recommendations that should help those implementing Java ME platforms to create an optimal MSA-compliant implementation.

It is important to note the structured approach to fragmentation management in the MSA specification, which actually defines two MSA platforms. When vision meets reality, it is always challenging, especially when considering the big capabilities gap between smartphones and mid-range devices. To ensure consistency for both high and low device segments, JSR-248 defines two platforms: full MSA targets high-end devices and MSA Subset targets mid-range devices. Both have well-defined conditions for features that may not be available on all MSA-compliant devices. This enables the MSA specification to be used across the widest possible variety of horizontal markets and customer segments.

As for CDC-based platforms, both MSA and MSA Subset can be implemented on top of CDC as long as the underlying implementation complies with JSR-248. However, although MSA with CDC is an interesting topic to

discuss, there is still some way to go until the industry standards leapfrog to a more powerful CDC configuration (or possibly a mobile platform similar to Java SE), and so CDC is not in the scope of the chapter. We discuss only MSA implemented over the CLDC 1.1 Configuration. For more information on MSA and CDC, refer to the Runtime-Execution-Environment descriptor attribute definition in MIDP 2.1 specification.

We now take a closer look at the Component JSRs in both the full MSA and MSA Subset platforms. We also give a few examples of Component JSR requirements that were clarified or added. Figure 6.1 shows MSA and MSA subset.

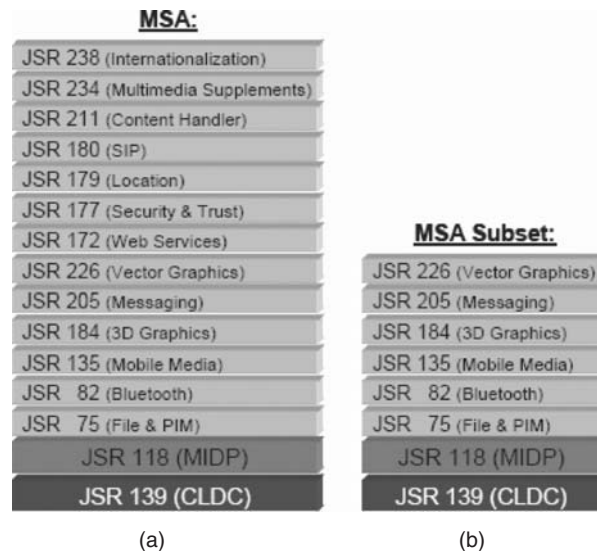


Figure 6.1 JSR-248 MSA Component JSRs: a) full set and b) subset

An MSA-compliant platform may include a newer version of a Component JSR, as long as the newer version is backward-compatible with the version specified in JSR-248 (however, application developers are encouraged to develop to the specific versions of the Component JSRs defined in JSR-248).

For each supported JSR, an MSA-compliant platform should comply with all the additional requirements for that JSR that are specified by the MSA. A JSR can be mandatory or conditionally mandatory (see Table 6.1). An MSA-compliant platform must support all mandatory Component JSRs that are part of the full MSA or MSA Subset. MSA provides clarifications that describe the conditions under which the conditionally mandatory Component JSRs and their optional packages are to be supported. For example, JSR-179 Location must be supported if the target device has a GPS receiver that is able to deliver geographical coordinates to other

run-time environments using another method, such as an accessory location device.

Table 6.1 MSA 1.1 Component JSRs

JSR	Name	Optionality
JSR-139	CLDC 1.1	Mandatory in MSA and MSA Subset
JSR-118	Mobile Information Device Profile 2.1	Mandatory in MSA and MSA Subset
JSR-75	PDA Optional Packages 1.0	Mandatory in MSA and MSA Subset
JSR-82	Java APIs for Bluetooth 1.1	Conditionally mandatory in MSA and MSA Subset
JSR-135	Mobile Media API 1.2	Mandatory in MSA and MSA Subset
JSR-172	J2ME Web Services Specification 1.0	XML parsing is mandatory in MSA
JSR-177	Security and Trust Services API 1.0: <ul style="list-style-type: none"> • CRYPTO • APDU • PKI • JCRMI 	Mandatory in MSA Conditionally mandatory in MSA Conditionally mandatory in MSA Not part of MSA or MSA Subset
JSR-179	Location API for J2ME 1.0.1	Conditionally mandatory in MSA
JSR-180	SIP API for J2ME 1.1.0	Mandatory in MSA
JSR-184	Mobile 3D Graphics API for J2ME 1.1	Mandatory in MSA and MSA Subset
JSR-205	Wireless Messaging API 2.0	Mandatory in MSA and MSA Subset
JSR-211	Content Handler API 1.0	Mandatory in MSA
JSR-226	Scalable 2D Vector Graphics API 1.1	Mandatory in MSA and MSA Subset
JSR-234	Advanced Multimedia Supplements 1.1	Mandatory in MSA
JSR-238	Mobile Internationalization API 1.0	Mandatory in MSA

MSA provides developers with a wider synergy of APIs, predictability of APIs on devices, ease of development and porting, and less testability. The advantage of MSA for device manufacturers is that more applications

means more demand for new devices that can run the new applications. More applications also increase the usage of services provided by network operators and hence increase their revenues. MSA satisfies the requirements of enterprises and IT departments because of the improved compatibility and manageability of secure applications. Finally, MSA helps to give consumers, who have more influence on technology than formerly, an enhanced and consistent user experience and improved service quality. Everyone benefits!

In broad terms, MSA is a standardization of existing JSRs, which was seen as both essential and feasible within a short timescale, relative to the time required for mass-market deployment. With that alignment came some clarifications and some additional requirements, which were required but not ground breaking.

To understand JSR-248 MSA well, the usual rule applies: go to ***www.jcp.org***, download the specification and read it. Interested readers may refer to [Knudsen 2007], which is a comprehensive guide to MIDP application programming and MSA for CLDC that provides practical examples and coverage of all the APIs that comprise MSA for CLDC.

6.2 What Can I Do with MSA?

In this section, we explore a few example applications that benefit from an MSA-compliant platform. We also see how we can improve an existing MIDP application if we assume it runs on an MSA-compliant Symbian phone.

The principle of how to use MSA is simple yet powerful. MSA provides you, as a developer, with an aligned and powerful set of standard Java ME APIs, so that you can focus on the main task of your application, which is the mobile service that the Java application enables. Your application lives in its natural mobile habitat of CLDC 1.1 and MIDP 2.1, which determines your application model and how the application is packaged, discovered and deployed. You throw into the melting pot some capabilities such as Advanced Multimedia, Location, PIM and File Connection. Dazzle the user with a 3D or SVG UI. Reach out of the device by using web services, short-link connectivity or remote connectivity. You need to make sure that the user can pay for the service, using SATSA. And, since it is a truly global application, it uses the simple but beneficial Mobile Internationalization API.

Implementing a real application is obviously not that simple. We might have added some fake glamour to a design process which obviously requires work and planning, but this is indeed how MSA looks from the feature perspective. This is what the power of MSA Component JSRs enables you to do. So let's zoom in again to the technical perspective and explore a few examples of applications that enable services with real value.

6.2.1 MusicMate

MusicMate is a cool and trendy name for a light and trendy mobile application whose main purpose is to be your assistant for music-related functionality. The first thing to do is to focus on the service you are trying to enable, as the main task of the application. Its most obvious function is to play music tracks. Where do those tracks come from? MusicMate can locate existing music files on your file system and it can help you purchase new songs or albums from various web services. If one of your friends likes a song you play to him, MusicMate can beam it to his mobile (whether he has MusicMate installed or not).

We have defined the main application task well enough for a high-level example, so let's move to the next stage. CLDC 1.1 and MIDP 2.1 are the environment in which the mobile application lives, so MusicMate must be a MIDlet, packaged in a JAR file and a signed JAD file. As we noted in Chapter 3, Java ME on Symbian OS does not impose fixed limits on the available computing resources, so while we should consider the download time and keep the JAR to a reasonable size, there is no need to be strict with packaged resources, such as background images. The same applies to other computing resources that are redefined by MSA, so you can focus on good design with reasonable resource utilization rather than on managing constraints.

Now we need to match the functionalities of MusicMate to the available MSA Component JSRs. MusicMate must find tracks on your mobile and provide some persistence mechanism for downloaded songs. We do that using JSR-75 FileConnection. At the user's request, possibly the first time it is launched, MusicMate looks in all the subdirectories under all the roots for existing content:

```
FileConnection fc = (FileConnection) Connector.open(path, Connector.READ);
Enumeration filesList = fc.list(".*", true);
String fileName;
while (filesList.hasMoreElements()) {
    fileName = (String) filesList.nextElement();
    fc = (FileConnection)Connector.open(path + fileName, Connector.READ);
    if (fc.isDirectory()) {
        // TODO: search under folder
    }
    else {
        // TODO: if is MP3 file, add to list
    }
}
```

Now that we have discovered all the tracks, we would like to play them. Multimedia capabilities are an essential feature of MusicMate and that obviously means that it uses JSR-135 Mobile Media API to control audio and multimedia resources and provide support for multimedia features, such as volume control. The following snippet illustrates playing a track from the file system:

```

FileConnection fc = (FileConnection)
                        Connector.open(mp3Path, Connector.READ);
InputStream is = fc.openInputStream();
player = Manager.createPlayer(is, "audio/amr");
player.realize();
volumeControl = (VolumeControl) player.getControl("VolumeControl");
if (volumeControl != null) {
    volumeControl.setLevel(100);
}
player.prefetch();
player.start();

```

To enrich our application with more advanced capabilities, we could also use JSR-234 Advanced Multimedia Supplements, which extends JSR-135 MMAPI and defines support for advanced multimedia functionality. For example, MusicMate could use the `EqualizerControl` which is an audio `EffectControl` and can be used to tune the sound. Two other examples for JSR-234 supplements are 3D audio support and audio radio support.

JSR-135 is a mandatory Component JSR in both full MSA and MSA Subset; JSR-234 is a Component JSR in full MSA only. That might not be a limitation, since good planning should include identifying the target audience and the types of device they have. That task should be completed before the coding begins and prevent you from using features that are not available on the target device (see Section 4.4).

This is sufficient to let us play tracks that are already on the user's phone but it's not enough for our trendy and cool MusicMate. We'll throw in JSR-172 Web Services, which provides a standard Java ME API to support web services and XML parsing. MusicMate uses it to let the user search for existing content in websites. The user types in the search criteria and MusicMate sends the query to the portal and displays the content that is available for purchase (with your business partner you need to agree on how to expose the web service at the server end).

MusicMate can send tracks to friends using a short-link connection, such as JSR-82 Bluetooth. Internationalization can be managed using JSR-238, which defines a common API that enables you to isolate localizable resources from the application source code and to access those resources at run time. It also provides support for a user-selected or device-selected locale, recognizing cultural conventions, such as date and time formats. The importance and benefit for developers of JSR-238 Internationalization is tremendous when you consider that mobile applications tend to be used in many countries by people who speak many different languages.

The following snippet gets a localized string and image, using the `javax.microedition.global.ResourceManager` class:

```
String baseName = ...
String[] locales = ...
res = ResourceManager.getManager(baseName, locales);
StringItem desc = new StringItem(res.getString(STRING_MY_DESC), "");
byte[] imageData = res.getData(MY_IMAGE);
Image img = Image.createImage(imageData, 0, imageData.length);
```

MusicMate must also let the user pay for downloaded tracks. JSR-229 Payment API was a mandatory part of the MSA 1.0 specification but was later removed from the MSA 1.1 maintenance release. This is because many platforms do not have the infrastructure required to support JSR-229 and including it would impose a requirement that could not be satisfied by many manufacturers. At the time of writing, it is also not included in MSA 2.0. However, JSR-177 SATSA API defines APIs that provide security and trust services for mobile devices and MusicMate can rely on these features to handle user authentication and payment.

So here we have a high-level plan for MusicMate. Figure 6.2 illustrates MusicMate's usage of MSA Component JSRs.

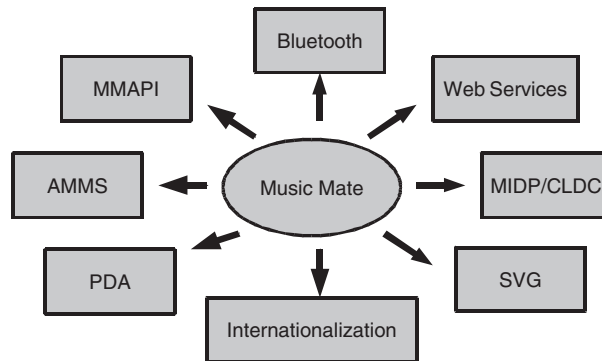


Figure 6.2 Usage of Component JSRs by MusicMate

A major item on our task list is to create a compelling user interface for MusicMate. The audience which this application is targeting will not be impressed with an interface based on LCDUI widgets. Perhaps we should go for an LCDUI Canvas-based UI? You could implement your own library, use a free library (e.g., LWUIT) or use a third-party library.

MSA 1.1 offers you two advanced graphics JSRs that are mandatory Component JSRs in both Full MSA and MSA Subset: JSR-184 3D Graphics (see Chapter 8) and JSR-226 Scalable 2D Vector Graphics. Because of the crucial importance of the user experience on mobiles, we now examine how to leverage JSR-226 SVG. JSR-226-SVG was initially specified by the JCP and later became a mandatory part of both full MSA and MSA Subset.

Recently, SVG has emerged as a prominent solution for creating rich user experiences, with SVG content creation being in the domain of visual

designers rather than software engineers. Let's admit it, visual designers do a much better job of creating rich UI than software engineers. In the current triangular model of mobile application development are the engineer, the visual designer and the content author. JSR-226 SVG enables the skills to be combined: the visual designer generates static and animated SVG images for the user interface, which the engineer includes in the mobile application.

An API is not enough to promote SVG technology so Nokia, Sun and Ikivo are working to ensure there are powerful tools to create mobile SVG content. The engineer's side of working with SVG is well covered by the NetBeans Mobility Pack and the Visual Mobile Designer gives you an intuitive way to incorporate SVG images and animations into your mobile applications. There are three steps:

1. Create static SVG files.
2. Animate the SVG.
3. Integrate the SVG content with the Java application.

The first step is performed by the visual designer, using a tool such as Adobe Illustrator¹ or Inkscape². The visual designer can create complete components (such as SVG menus or splash screens) or individual components (such as buttons). The components are saved in SVG basic format (more specifically, the DTD for saving should be SVG basic 1.1).

The static SVG files can be animated by Ikivo Animator.³ The visual designer opens a background SVG image, turns on animation mode, and drags an individual component such as a label or a button. The animation is created by adding a key point for the starting position, a second key point for the end of animation, and the movement path. The animated file can be saved as SVG-T 1.1 format. (For compatibility, you might need to edit the `animateTransform` and `animateMotion` SVG XML properties to ensure the animation works correctly.)

The final step is to integrate the SVG content with the Java application. Copy the SVG files to the project resources folder and open NetBeans Visual Designer, which provides number of SVG components such as `SplashScreenComponent`, `SVGMenu`, `SVGPlayer`. After you select which of those components to use, you can drag the SVG-T files to the SVG component and set attributes such as time increments and full-screen mode. Your Java code can also influence the SVG images, for example, by adding an `SVGEventListener` and code that increments the SVG timer.

A major advantage of using JSR-226 SVG is the simple interfaces between SVG-T and JSR-226. When the static SVG files are created, you

¹ www.adobe.com/products/illustrator

² www.inkscape.org

³ www.ikivo.com/animator

can add and name hooks which are accessible using the JSR-226 API. Using those hooks, any SVG graphic can be easily replaced as long as it uses the same hook to which the Java code can attach.

The task of creating a rich and compelling UI is highly important. Expectations of a rich and compelling user interface have recently been raised significantly. The user experience can have a significant effect on the sales and popularity of an application compared to other applications that provide the same functionality.

There is a wealth of examples and online resources regarding JSR-226 SVG on the various developer program websites.

6.2.2 Geotagging

Geotagging is the process of adding geographical information, which usually consists of latitude and longitude coordinates, to various forms of media, such as photographic images. The uses for geotagging can vary from finding location-based news to finding images taken near a given location by using the coordinates in a geotagging-enabled website (e.g., Flickr).

So let's consider an application that lets you take a picture and send it to your Flickr account, geotagged with the location in which the picture was taken.

The application can take a picture using JSR-135 MMAPi or JSR-234 AMMS. Initially, we need a handle to the camera player:

```
// get the camera player
Player camera = Manager.createPlayer("capture://video");
```

Then we take a snapshot using JSR-234 AMMS or JSR-135 MMAPi. In AMMS, we use `SnapshotControl`:

```
snapCtrl = (SnapshotControl) camera.getControl(
    "javax.microedition.amms.control.camera.SnapshotControl");
snapCtrl.start(1);
```

In MMAPi, we use `VideoControl`:

```
VideoControl videoCtrl = (VideoControl) camera.getControl("VideoControl");
byte[] snapData = videoCtrl.getSnapshot("encoding=jpeg");
```

In such an application, the camera capabilities play a central role so we are very likely to use some of the new controls that JSR-234 AMMS introduced. For example, we would want to set the exposure mode of

the camera, set the camera flash on with red-eye reduction and set the optical zoom:

```
p = Manager.createPlayer("capture://video");
p.realize();
p.start();
cameraControl = (CameraControl)
    p.getControl("javax.microedition.amms.control.camera.CameraControl");
cameraControl.setExposureMode("night");
flashControl = (FlashControl)
    p.getControl("javax.microedition.amms.control.camera.FlashControl");
flashControl.setMode(FlashControl.FORCE_WITH_REDEYEREDUCE);
focusControl = (FocusControl)
    p.getControl("javax.microedition.amms.control.camera.FocusControl");
if (focusControl.isAutoFocusSupported()) {
    focusControl.setFocus(FocusControl.AUTO);
}
zoomControl = (ZoomControl)
    p.getControl("javax.microedition.amms.control.camera.ZoomControl");
int maxOpticalZoom = zoomControl.getMaxOpticalZoom();
zoomControl.setOpticalZoom(maxOpticalZoom);
```

We get the current longitude and latitude location values using JSR-179 Location API:

```
// get location provider
LocationProvider locPvdr = LocationProvider.getInstance(null);
// get the current location
QualifiedCoordinates qc = locPvdr.getLocation().getQualifiedCoordinates();
double currentLongitude = qc.getLongitude();
double currentLatitude = qc.getLatitude();
```

In the MusicMate example, we used JSR-172 Web Services to communicate with remote servers. If you are using a RESTful Web Service, you could use simple HTTP requests and responses,⁴ using MIDP 2.0 `HttpConnection`. The URL is built as follows:

```
String restURL = "http://api.flickr.com/services/upload/?tag=";
...
url += currentLongitude;
url += currentLatitude;
```

An HTTP connection is opened to send the image:

```
HttpConnection hc = (HttpConnection)Connector.open(restURL);
hc.setRequestMethod(HttpConnection.POST);
```

⁴ www.en.wikipedia.org/wiki/REST

This example uses a more limited combination of MSA Component JSRs and capabilities, which is sufficient to provide the basis for a geotagging application. The main APIs used for such an application are JSR-234 AMMS and JSR-179 Location. Both APIs are part of the full MSA (JSR-179 Location is conditionally mandatory). This example does not fully exploit the power of these Component JSRs but it still illustrates the richness of Java applications that target a full MSA-compliant Java ME platform. (To clarify, both JSRs are supported by S60 5th Edition, S60 3rd Edition FP2 and UIQ 3.3.)

6.2.3 Radio Tuner

In the previous example, we used JSR-179 and JSR-234. You could also use those Component JSRs to create a location-based radio tuner, which can discover and listen to radio stations in the current location.

To discover local radio stations, we can query a remote server using JSR-172 Web Services or, as in the previous example, an alternative protocol such as REST. Again, JSR-234 AMMS provides interfaces for tuner features, such as audio radio playback and access for Radio Data System (RDS) features.

`TunerControl` is the main class for controlling a tuner. It is used to control, for instance, the frequency and modulation used. The channel presets of the device are also accessed via `TunerControl`. The core functionality is the radio tuner. Having retrieved the tuner control, you use it for tuning the radio frequency as in the example below:

```
Player radioPlayer = Manager.createPlayer("capture://radio");
radioPlayer.realize();
radioPlayer.addPlayerListener(new MyPlayerListener(this));
tunerControl = (TunerControl) radioPlayer.getControl
    ("javax.microedition.amms.control.tuner.TunerControl");
tunerControl.setStereoMode(TunerControl.STEREO);
rdsControl = (RDSControl) radioPlayer.getControl
    ("javax.microedition.amms.control.tuner.RDSControl");
rdsControl.setAutomaticTA(true);
radioPlayer.start();
int freq = tunerControl.seek(966000, TunerControl.MODULATION_FM, true);
tunerControl.setFrequency(freq, TunerControl.MODULATION_FM);
```

We need to provide a rich and compelling user experience. Again, we can implement it using JSR-226 SVG, or JSR-184 3D for additional effects. The following code example illustrates the combining of a compelling UI and data; it uses JSR-226 SVG to display channel information retrieved from JSR-234 AMMS:

```

SVGElement svgChannelInfo = ...
// get tuner information
String channelName = rdsControl.getPS();
String radioText = rdsControl.getRT();
String programmeType = rdsControl.getPTYString(true);
String infoStr = ... // build the display string from the above data
svgChannelInfo.setTrait("#text", infoStr);

```

In this example, we went beyond MSA Subset and used JSR-234 AMMS to provide the core radio-tuner support, JSR-179 Location to give the user a location-sensitive context, JSR-172 Web Services to discover locally available radio stations and JSR-226 SVG to build a compelling UI. This shows how a Java application using MSA Component JSRs can create a compelling, practical and personal mobile experience for the user.

6.2.4 More Examples

Let us consider a few more types of application and briefly associate them with the Component JSRs they use. Messaging applications can access telephony messaging features such as SMS and MMS using JSR-205. The application registers with the Push Registry to be activated when an incoming message addressed to it is received (an SMS is associated with an application by a port number; an MMS is associated by an application ID). JSR-180 SIP can be used to extend the messaging support for integration with instant messaging (IM) clients (e.g., MSN Messenger) and by using JSR-177 SATSA to secure and authenticate the transmitted instant messages. Going further, JSR-172 Web Services can be used to integrate with social networking websites (e.g., LinkedIn) to make the user's remotely stored contact information available locally in the phone address book, using JSR-75 PIM.

Games that run on an MSA-compatible device could use JSR-184 for a 3D UI, JSR-135 for multimedia and sounds, and JSR-82 for local peer-to-peer (P2P) connections over Bluetooth to play against another user or JSR-180 for P2P gaming sessions over the network. Games can significantly benefit from JSR-234 AMMS, which enables you to create an audio scene in which different sound sources are located in a virtual acoustic space (see Chapter 8).

Mapping applications would use JSR-179 to get the current location, storing and retrieving landmarks and points of interest that are shown over an SVG map rendered by JSR-226. The data would be retrieved using JSR-172 Web Services and be cached locally using JSR-75 FileConnection. JSR-238 Internationalization is absolutely mandatory for mapping applications that are used by travelers from across the world.

Meshed web applications that combine data from several services would use JSR-172 to access and parse data from various remote web services, JSR-211 Content Handling to launch the appropriate content player MIDlet, and could store the data locally using JSR-75 FileConnection.

A mobile financial terminal would let you view graphs and charts using JSR-226 SVG, perform secure and authenticated bank transactions using JSR-177 SATSA and JSR-172 Web Services and store important reminders using JSR-75 PIM.

We could find more examples and spend time going into detail for each of them, but the point is clear – a world in which MSA is the Java ME standard is a much more consistent and predictable world, which allows the same application to run easily and more predictably on a wider variety of MSA-compliant phones. After that, it is only natural that a more predictable and consistent platform would enable more mobile services to emerge and we are likely to see more and more advanced Java applications that let us do more on our phone.

6.3 Spicing up Legacy MIDP Applications

If you have a successful Java application that runs on MIDP- or JTWI-compliant platforms, it is quite likely that you need different versions of the application that target different devices according to their capabilities and the Java specifications which they support. This fragmentation can be minimized by using MSA, so you spend less time on version management and code maintenance.

This is similar to the transition from MIDP 1.0 (which did not have gaming APIs and games had to use proprietary APIs from the manufacturer or custom-made APIs) to MIDP 2.0 – existing games were upgraded to use the new MIDP 2.0 gaming API. The consistency of the games increased by using a standard API available across compatible Java platforms.

So what can you do with MSA, to help you as a developer and your company? First, your application can become even better if you decide to add capabilities to it and leverage MSA's strengths. You could include use of additional Component JSRs that are not part of JTWI and enrich your application with features such as SVG, location, and so on. To illustrate how that can be done, let's take an example of a well-known existing application, MobileAerith, and see how we can improve it using MSA.

MobileAerith is a demonstration photo management system (see Figure 6.3). The user can take a picture with a phone camera, manage albums of photos, view thumbnails of the photos in the album, and, when a thumbnail is chosen, view the bigger image. MobileAerith runs on MIDP 2.0; it also requires JSR-135 MMAPI and uses JSR-226 SVG extensively for the creation of a slick user interface that is a joy to behold!

Figure 6.4 shows the JSRs used by MobileAerith before it is enhanced with MSA.

By now you probably want to go through the application code and run it on your device. MobileAerith is part of the ME Application Developers

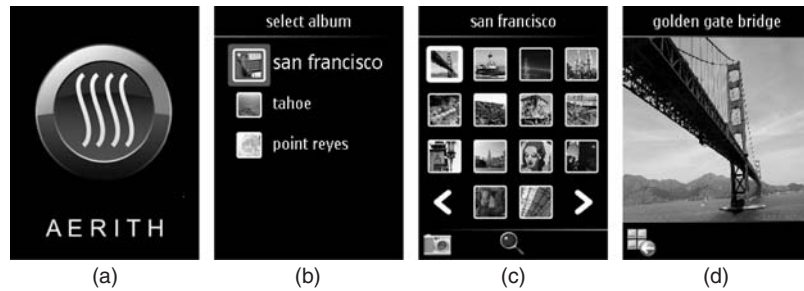


Figure 6.3 MobileAerith screen shots: a) splash screen, b) album selection menu, c) album thumbnails, and d) image

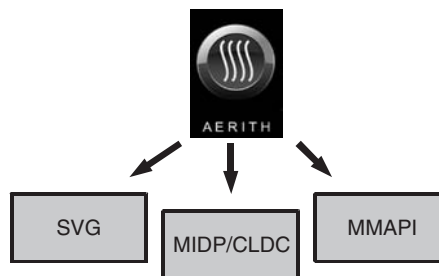


Figure 6.4 MobileAerith before MSA 1.1

project,⁵ which is a great source of advanced applications, useful tips, information on the usage of JSRs and other help for application developers. There is also a wiki with more good information and links. All the sources and resource files for MobileAerith are available under the MobileAerith directory. You can read more and browse the source code from meapplicationdevelopers.dev.java.net/uiLabs/MobileAerith.html.

To get the source, you can access the MobileAerith sample from the NetBeans 6.1 Update Center (see Figure 6.5). From NetBeans, choose Tools, Plugins, Available Plugins, Aerith Mobile Photo Album Sample, Install.

To open the sample after installation, choose File, New Project, Samples, Mobility, MIDP, Aerith Mobile Photo Album Sample.

Now that you have MobileAerith running on your Symbian phone and you have been through the code in NetBeans, just stop and think how many more features we can add to MobileAerith! We could upload the images to an image-hosting website account (e.g. Flickr). We could have the images geotagged, to show where the picture was taken. Both of these features are possible using MSA. So let's start planning how we add those features as we take a tour through MobileAerith.

⁵ meapplicationdevelopers.dev.java.net

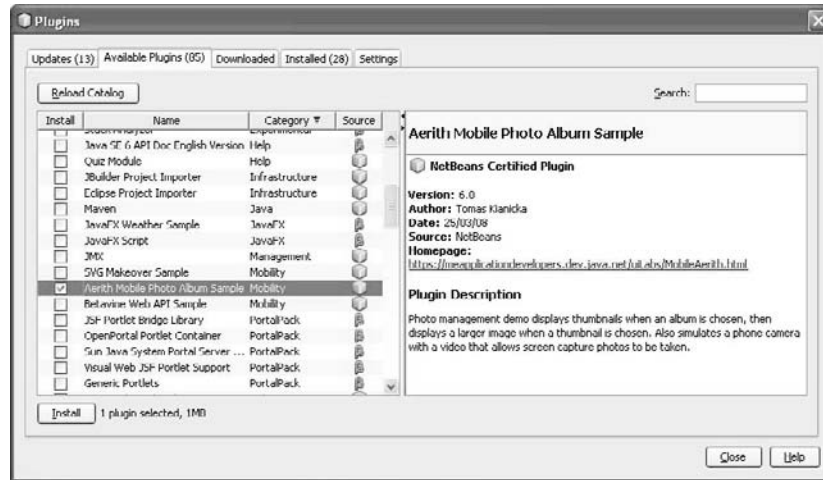


Figure 6.5 Getting MobileAerith using NetBeans Update Center

We open up the application and see a great SVG UI. We would like to browse some of our private pictures on our image-hosting website. That has two technical requirements – user authentication when signing in and interoperability with a remote web service.

The precondition to uploading and downloading pictures from the image-hosting website is signing in, which we can achieve using JSR-177 SATSA. To avoid passing a readable text password from client to server, you can send an MD5 – a message digest (a small ‘fingerprint’ of a larger set of data) – of the password; if this matches the one on the server, the client is authenticated.

The following code demonstrates how to create an MD5 message digest using JSR-177 SATSA:

```
// MD5 digest length is 128 bits
byte[] digest = new byte[16];
java.security.MessageDigest md =
    java.security.MessageDigest.getInstance("MD5");
md.update(password, 0, password.length);
md.digest(digest, 0, digest.length);
```

JSR-172 Web Services give us the interoperability with the web service. There are other ways to communicate with a remote service, for example, JavaScript Object Notation (JSON), a text-based data interchange format, and REST, but a major benefit of using JSR-172 is that under MSA it becomes a standard JSR. There is no need to maintain proprietary code or third-party packages.

We have seen the stored images and would like to take a new picture and upload it to the website with the location information. First, we get our current location using JSR-179:

```
LocationProvider lp = getLocationProvider();
Location loc = lp.getLastKnownLocation();
// TODO: check not null, check timestamp etc..
QualifiedCoordinates qc = loc.getQualifiedCoordinates();
double lat = qc.getLatitude(),
double lon = qc.getLongitude(),
```

Next, we get a map of the current location from a geo site using JSR-172 Web Services, which retrieves the map as an SVG image. We use the NetBeans SVGPlayer utility class, which can be used as a component in NetBeans Visual Designer and encapsulates SVGAnimator:

```
Display display = getMIDletDisplay();
InputStream in = getMapFromServer(); // get input stream to SVG map, using
// Web Services
SVGImage svgMap = (SVGImage)SVGImage.createImage(in, null);
svgPlayer = new SVGPlayer(svgMap, getDisplay());
svgPlayer.setStartAnimationImmediately(true);
svgPlayer.setTitle("Current location map:"); // set title
SVGElement labelElement = getSVGMapLabelElement(svgPlayer);
// Set the SVG element trait
labelElement.setTrait("#text", "Picture taken here!");
display.setCurrent(svgPlayer.getSvgCanvas());
```

We can then take a picture using JSR-234 AMMS, which provides enhanced access for camera-specific controls, such as brightness, contrast, flash lights, lighting modes and zooming. These enable us to have fine-grained control over the camera and the resulting picture. The following example demonstrates some of the controls that JSR-234 AMMS adds on top of JSR-135 MMAPI:

```
player = Manager.createPlayer("capture://video");
...
snapshotControl = (SnapshotControl) player.getControl
    ("javax.microedition.amms.control.camera.SnapshotControl");
flashControl = (FlashControl) player.getControl
    ("javax.microedition.amms.control.camera.FlashControl");
zoomControl = (ZoomControl) player.getControl
    ("javax.microedition.amms.control.camera.ZoomControl");
focusControl = (FocusControl) player.getControl
    ("javax.microedition.amms.control.camera.FocusControl");
exposureControl = (ExposureControl) player.getControl
    ("javax.microedition.amms.control.camera.ExposureControl");
imageFormatControl = (ImageFormatControl) player.getControl
    ("javax.microedition.amms.control.ImageFormatControl");
...
```

```

cameraControl.setExposureMode("landscape");
flashControl.setMode(FlashControl.OFF);
zoomControl.setDigitalZoom(350);
...
snapshotControl.setFileSuffix(".jpg");
snapshotControl.setFilePrefix("aerith");
imageFormatControl.setFormat("image/jpeg");
imageFormatControl.setParameter(FormatControl.PARAM_VERSION_TYPE, "JPEG");
imageFormatControl.setParameter(FormatControl.PARAM_QUALITY, 80);
...
snapshotControl.start(1);

```

Before we upload the picture using JSR-172, we can also apply some nice image effects using the image-processing capabilities of JSR-234 AMMS, such as effects and transformations. For example, we could convert the image to monochrome:

```

// get media processor for jpeg format
MediaProcessor processor =
    GlobalManager.createMediaProcessor("image/jpeg");
// get processor controls
ImageEffectControl imageEffectCtrl = (ImageEffectControl)
    processor.getControl
        ("javax.microedition.amms.control.imageeffect.ImageEffectControl");
...
// set the new preset to convert the image to monochrome image
imageEffect.setPreset("monochrome");
imageEffect.setEnabled(true);
...
// Wait until the processing has been completed
processor.complete();

```

With these features implemented, we would probably include the option to beam pictures to our friends from the locally-cached album, using JSR-82 Bluetooth. We'd also internationalize our revamped MobileAerith with JSR-238 Internationalization.

We could add even more: JSR-75 FileConnection to cache the images; JSR-205 WMA to send SMS to our friends once we take the picture; registration in the Push Registry to be activated when a similar message from a friend is received. But it looks like we have already achieved a significant improvement, so let's put the all the new pieces together and see what we have now.

The original MobileAerith (before MSA), as shown in Figure 6.4, uses JSR-226 SVG to provide a great UI and JSR-135 MMAPI to take snapshots. We added JSR-172 for uploading and downloading pictures from the account, after having signed in using JSR-177 SATSA, got our current location using JSR-179 Location and used that information and JSR-172 again to retrieve a map from a geo site. Advanced camera operation,

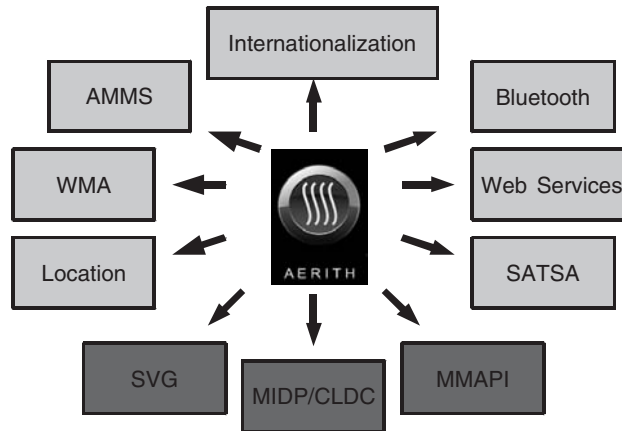


Figure 6.6 MobileAerith enhanced with MSA capabilities

picture scaling and processing was done using advanced features of JSR-234 AMMS that are not in JSR-135 MMAPI. Figure 6.6 shows the new version of MobileAerith after the big facelift, using MSA Component JSRs.

MSA's synergy of APIs has enabled us to give added value to an existing mobile application, whilst at the same time ensuring that it runs on many more MSA-compliant devices with less version management overhead

6.4 Beyond MSA 1.1: MIDP 3.0 and MSA 2.0

So far we have discussed what MSA 1.1 is and shown some examples of Java applications that use MSA 1.1. There is even more to expect in the future. The MSA standardization efforts are intended to be ongoing activities to define the evolving Java platforms to meet the latest market requirements.

Apart from JSRs that handle specific technologies and are currently being worked on, we can already look ahead to JSR-249 MSA 2.0 and JSR-271 MIDP 3.0, which are being drafted at the time of writing this book.

Their release is determined by the time required for mass-market deployment and the readiness of the relevant Java standards which themselves depend on technology standardization in other relevant standardization organizations. Any future standard will strive to maintain backward compatibility as much as possible and plan ahead to be forward compatible with future standards.

The advantages and disadvantages of the focus on high-volume devices with limited capabilities will remain. A predictable application execution environment across the industry means that dealing with high-end requirements is not likely to be a primary requirement of MSA in the immediate future.

So what can we expect to see in MSA 2.0 and MIDP 3.0? The structure of the JSR-249 MSA 2.0 specification is similar to that of MSA 1.1 (Component JSRs, clarifications, recommendations, etc.). The current scope is for three categories of MSA 2.0 platform:

- MSA 2.0 Advanced Platform (AP) – backward compatible with full MSA 1.1
- MSA 2.0 Standard Platform (SP) – backward compatible with MSA 1.1 Subset
- MSA 2.0 Entry Platform (EP) – backward compatible with JTWI 1.0.

As you can see from Figure 6.7, there are new JSRs coming and all of them will bring more opportunities for services and developers. (Note: As we write this, JSR-249 is not finalized so there may be some changes in the final release.)

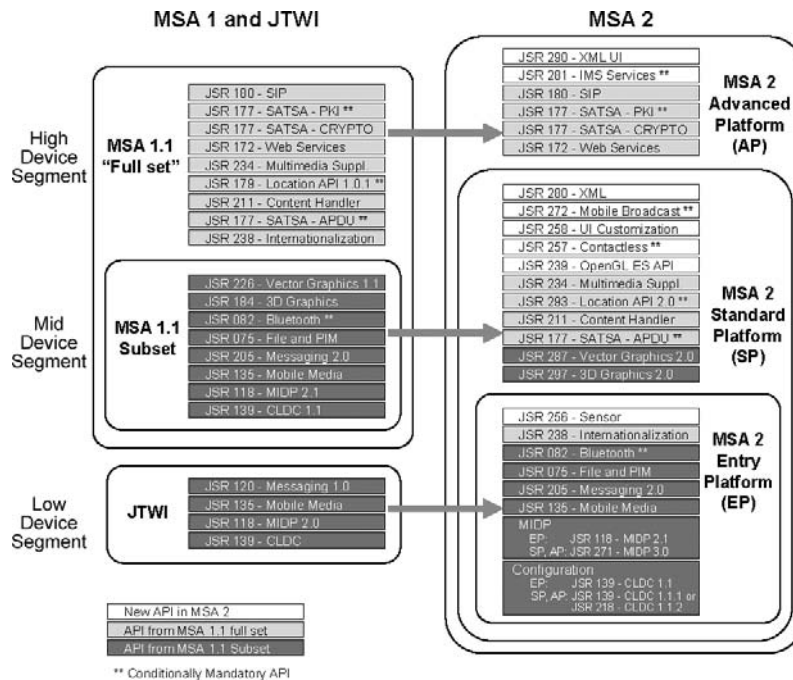


Figure 6.7 From JTWI and MSA 1.1 to MSA 2.0

While MSA 2.0 will increase the technology support, JSR-271 MIDP 3.0 will bring great advancements to the environment in which Java applications run. The specification defines:

- Shared Java libraries (LIBlets)
- Receiving more system events using the `javax.microedition.event.*` package
- Inter-MIDlet Communication (IMC)
- CLDC 1.1 and CDC 1.1 support
- MIDlet concurrency
- Idle screen MIDlets
- Packaging, deployment, installation, and security.

MIDP 3.0 also refers to other ongoing JSRs (e.g. JSR-258 Mobile User Interface Customization and JSR-307 Network Mobility and Mobile Data API).

The world of Java applications in a few years from now will be powerful and different. Imagine a scenario in which two concurrently running Java applications (MIDlets, or possibly another model, e.g. Xlets) are both using a shared LBS LIBlet and communicate with each other using IMC. They will be using advanced features such as JSR-256 Sensor API, JSR-293 Location API 2.0 and will be able to respond to many more system events. When they exit, the idle screen MIDlet will start running. This is definitely something to wait for!

6.5 MSA and Symbian OS

At the time of writing this book, the first S60 5th Edition smartphones have been announced and S60 3rd Edition FP2 devices are already in the shops.

Two of the major goals of JSR-248 MSA are alignment to higher standards and defragmentation. Symbian's strategy has always been to enable a powerful Java ME run-time environment as required. That means that there is ongoing progress and new JSRs are added in each Symbian OS release, either by Symbian or by Symbian OS licensees. The set of APIs that we detected in Chapter 3 already exceeds the list of Component JSRs in MSA Subset. As a matter of fact, the detected devices are closer to full MSA than to MSA Subset.

So you can safely assume that generally the direction of Java ME on Symbian OS is towards full MSA and beyond. More than that, Java ME on

Symbian OS is not going to stop once it reaches full MSA but will keep progressing and evolving because of the strategic importance of having a powerful Java ME platform for Symbian OS.

We discussed consistency in Chapters 3 and 4, where we showed that having a common implementation for many devices from many manufacturers ensures consistency in a remarkable way. Different models, from different phone manufacturers have the same underlying implementation. That is consistency.

Other MSA clarifications simply do not apply to Symbian OS. In areas such as supported protocols, Java ME on Symbian OS exceeds the requirements (e.g., by supporting TCP server sockets). In areas of computing resources such as heap size and number of threads, Java ME on Symbian OS does not set any fixed limits. As Java ME is hosted on a powerful native operating system, application developers are free to focus on the application task.

6.6 Summary

MSA supersedes JTWI and defines the next generation of Java ME platform for mobile handsets based on MIDP/CLDC. It comprises a collection of Component JSRs and additional clarifications and requirements that define two MSA platforms: full MSA and MSA Subset.

MSA lets developers focus more on the main task of the application, on the mobile service which the Java application enables, by providing a rich and predictable Java ME environment.

We discussed building MusicMate, an application that uses the richness of MSA 1.1. Then we showed a few more examples and moved to enhance an existing application, MobileAerith, with features that are standardized in the MSA 1.1 specification.

The message should be clear, start writing great applications using MSA!

7

DoJa (Java for FOMA)

This chapter introduces the DoJa profile from NTT DoCoMo which is used in millions of handsets worldwide and especially in Japan. We start with some mobile application history in the Japanese market and then examine what the DoJa profile is all about. Following on from that, we configure the Eclipse IDE for DoJa development and run through some application code. Then we have a look at the differences you need to be aware of when porting from MIDP applications to DoJa and have a brief word about game development.

In Section 7.8, Sam Cartwright from Mobile Developer Labs provides an in-depth look at the latest DoJa profile available to developers based in Japan, giving us a rare insight into this exciting part of the mobile development world. He walks us through some of the more advanced features available on Japanese devices – including biometrics, gesture readers, compasses and virtual purchasing.

7.1 In the Beginning. . .

. . .someone created the mobile phone. Then some bright spark built the first mobile phone game and a decade later we have a multi-billion dollar industry led by Japan. Right now, there are three major operators in Japan – KDDI, NTT DoCoMo and Softbank (also known at various times as J-Phone, Vodaphone KK and Vodaphone Japan). These three companies pretty much control the mobile phone market in Japan although NTT DoCoMo easily holds the balance of power with a 47% market share which amounts to over 53 million subscribers.¹ When you consider that

¹ KDDI has over 30 million subscribers and Softbank has over 18 million. So in Japan, four out of every five people you meet has an account with one of these three companies – this is very big business! See www.nttdocomo.co.jp/english/corporate/ir/binary/pdf/library/annual/fy2007/p10.e.pdf.

the population of Japan is just over 127 million this is a very large chunk of the market.

The first Java enabled phones were shipped by NTT DoCoMo² in early 2001 and the subsequent success of these caused a type of Java-based arms race between these competitors. From the start, Softbank and KDDI decided to leverage the Kauai Virtual Machine (KVM) provided as part of the JBlend platform supplied by Aplix³ and based their offerings on CLDC 1.0 and MIDP 1.0 with customized extensions.

NTT DoCoMo, however, went their own way, jumped in at the deep end, and developed a proprietary Java ME profile called DoJa. In taking a path that at first glance appeared to be fraught with risks, they nevertheless managed to grab the lion's share of the market. This is largely due to the fact that they have never been bound by the limitations of MIDP either in business strategy or in the actual technology.

DoJa is analogous to MIDP and sits on top of CLDC just as MIDP does. In fact there is so much overlap that many class methods in the respective APIs even have the same names. However they are vastly different profiles designed with very different aims in mind – MIDP is a general and minimal specification to provide some kind of common baseline that is available across a wide range of mobile devices; DoJa is a business solution to a specific market opportunity aimed at creating, maintaining, and increasing revenue streams in the Japanese mobile subscriber market.

By choosing custom development, NTT DoCoMo effectively kept the cross-platform abstractions provided by CLDC while neatly side-stepping MIDP development altogether (and therefore all inherent technical and business risks associated with it). By creating their own profile they managed to achieve complete control over the technology platform they were using and are therefore not dependent on any third party whether it be the supplier of a KVM implementation or the creation of new standards, such as the publicly-driven JCP.

NTT DoCoMo is a leader in research and development creating both the W-CDMA protocol and the world's first 3G service that uses it, Freedom of Mobile Access (FOMA), in 2001. Until fairly recently, FOMA phones have run Symbian OS v8.1b; this gives them the benefits of the EKA2 kernel but avoids having to deal with the platform security model introduced in Symbian OS v 9.

This has tremendous benefits to NTT DoCoMo – since the platform is closed, there's no risk of third parties installing native applications after market. And since they control all native development in-house, they can apply their own quality assurance processes and therefore don't need to get these applications Symbian Signed. However, things are

² www.nttdocomo.com

³ www.aplixcorp.com/en/index.html

changing because some of the latest handsets released in 2008, such as the SH905iTV from Sharp and Fujitsu's F905i, run Symbian OS v9.3.

NTT DoCoMo went a step further and created their own equivalent of the S60 and UIQ platforms, called Mobile Oriented Applications Platform (MOAP).⁴ MOAP has variants for both Symbian OS and Linux-based mobile phones. Again, there are no MOAP APIs available to external developers.

Symbian's licensees have sold over 40 million handsets in Japan so far⁵ – a very big number and one which is growing all the time. It's a pretty exciting (but restrictive) part of the mobile development spectrum. Native C++ development is limited to in-house personnel and there is no support for MIDP, so the only way for third-party developers to write applications for FOMA devices is to develop using the DoJa profile (see Section 7.2).

There can be a bit of confusion here as sometimes NTT DoCoMo handsets are referred to as 'i-mode' handsets. i-mode is an always-on Internet service provided to NTT DoCoMo subscribers accessible from anywhere in Japan. These devices have an i-mode start button which allows access to up to 12,000 official websites (and many more unofficial ones) with information services for sports, ticketing, weather, and games as well as providing email access.

The i-mode service was extremely successful in Japan and spread across the world although not with the same results. In some countries (e.g. the UK and Australia), the service has been dropped due to poor subscription levels, but in others, such as Israel, it has done extremely well. An always-on service across the whole country is possible because the service in Japan uses a realistic pricing model and has the infrastructure to support it. It's extremely inexpensive as users are charged on data volumes sent or received rather than connection air time. There are also family plans, fixed cost plans and various other discount plans which have all contributed to the success of i-mode in Japan.

As we'll see throughout this chapter, the creation of the DoJa profile and its design and implementation boundaries are the direct result of a cohesive business model. By imposing limitations on what the API exposes, it has been possible to define *how* technical solutions must be realized, in a manner that maximizes revenue streams for the operator, NTT DoCoMo. For example, limitations on the size of local storage mean that large assets need to be stored on remote servers and accessed over the air (remember charges are calculated on a volumetric basis); the lack of access to Bluetooth meant that any simple messaging mechanism between devices must go via HTTP (Bluetooth support was introduced in DoJa 5.0). As can be seen, this means that DoJa applications tend to generate much more traffic than the equivalent MIDP applications.

⁴ en.wikipedia.org/wiki/MOAP

⁵ www.symbian.com/news/pr/2008/pr200811003.asp

In addition, NTT DoCoMo control the intellectual property rights to content and can ensure the security and, more importantly, the homogeneity of their technical implementations. This means that they do not have to plan mitigation strategies for the kinds of fragmentation problems seen in Java ME mainstream profiles to date. Furthermore, NTT DoCoMo can produce new versions of their DoJa profile to their own time schedule to meet the changes in customer demands as new technologies evolve. To put that into perspective, there are currently some seven or more versions of the DoJa profile for use inside Japan, as well as another two overseas editions. Compare this to MIDP, where the 3.0 specification is still being finalized.

Given all of this, it is little wonder that NTT DoCoMo has taken a winning lead in this highly competitive market and has managed to stay there so far. And while it's all very new and foreign to us, looking at what's happening in Japan now gives us a view into our probable future. Many of the mobile device technologies and infrastructures being used in Japan may very well become commonplace over the next decade in cities such as London, New York, and maybe even Sydney! So take this chance to get familiar with this part of the world and consider the opportunities available because with DoJa, you can sell your applications to a technologically savvy customer base of more than 50 million people, many of whom regularly spend money and time on mobile games.

7.2 DoJa – the Basics

DoJa applications are small programs, written in Java, known as 'i-Appli'. To get started developing with DoJa, a free toolkit, not unlike Sun's Wireless Toolkit, is available for download from NTT DoCoMo at **www.doja-developer.net/download**. This is a very simple tool that supports basic i-Appli development using a text editor, such as Notepad. It also lets you set run-time configuration parameters and various network settings and comes with an Eclipse plug-in (available when you choose the Custom installation option, as shown in Figure 7.1).

Detailed use of this toolkit is discussed in [Stichbury 2008, Section 10.4], so we won't go any further with this here. Also, developers usually use an IDE such as Eclipse for effective DoJa development. However, the toolkit is still a great way to get started if you want to use it. We walk through setting up Eclipse in Section 7.5.

Like MIDlets, i-Appli are deployed as JAR files with an accompanying metadata file called the Application Descriptor File (ADF) which has a role analogous to the JAD file in MIDP. A DoJa profile implementation

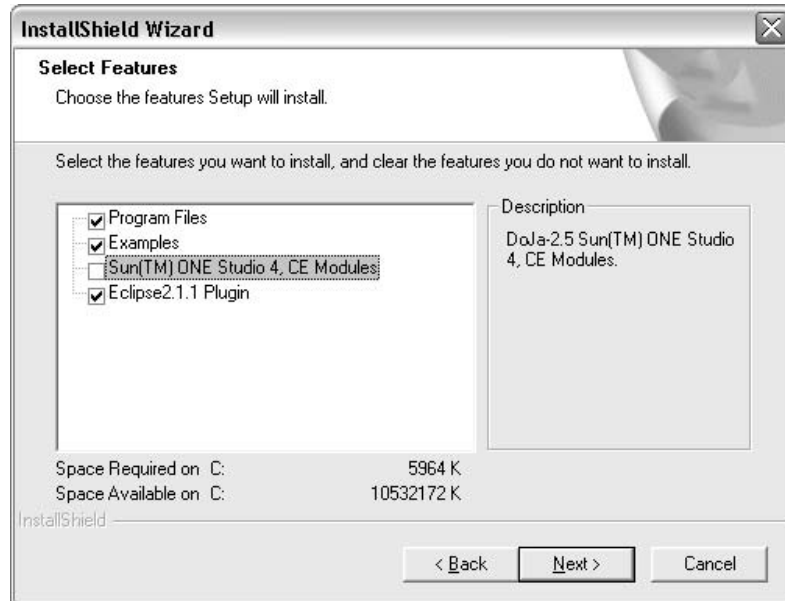


Figure 7.1 DoJa Toolkit Custom Installation

sits astride CLDC 1.1 so all of the standard low-level libraries are there as well as the same limitations:

- the generic connection framework
- no finalization
- bytecode pre-verification
- reduced thread functionality, compared to Java SE
- reduced `java.util` and `java.io` package functionalities, compared to Java SE.

Most versions of the DoJa profile are designed for use in Japan only. There are two overseas versions – DoJa 1.5oe and DoJa2.5oe – for the rest of us to use but they expose much less functionality to the developer than the DoJa versions that are deployed in Japan. This is because gesture readers, barcode scanners, virtual banking and e-wallets require a wide variety of physical and social infrastructure changes to happen before they can realistically be supported. We're probably due for a new 'oe' release, but, at time of writing, we don't know the schedule for this.

It should be obvious that there are no JSRs in the DoJa world. There are simply versions of DoJa with more or less functionality as the case requires – as mentioned above, fragmentation is far less of an issue when you control all aspects of the manufacturing process! For the same reason, and unlike with MIDP, new versions of the DoJa profile do not necessarily have to be backwards compatible; indeed, it is very common to have classes or packages re-named, classes moved to another package or simply removed entirely between one version release and another!

Most readers are already familiar with Java ME development using some combination of MIDP and CLDC. This makes learning DoJa a fairly simple task once you understand what the limitations are. To give some context, Sections 7.3, 7.4, 7.5, 7.6 and 7.7 specifically discuss DoJa2.5oe – the latest overseas edition available. Many of the functional limitations outlined in the list below are removed or relaxed in the versions internal to Japan (such as Bluetooth APIs being available) but for those of us who don't live in Japan, the 2.5oe API is all we get.

With that in mind, let's look at some of the main restrictions that can tend to frustrate newcomers from an MIDP background:

- There is no multitasking so only one application may be running at any one time.
- The size of the actual JAR file can be no more than 30 KB – larger JARs are simply not installed.
- There is no record management system (RMS) to speak of – the nearest equivalent is a binary random access file, called the Scratchpad, limited to a maximum of 200 KB in size.
- You may open sockets over OBEX or HTTP but HTTP sockets can only be opened to the same server from which the application was originally downloaded.
- When you have an HTTP connection, data transfer is limited to 5 KB upload and 10 KB download.
- There is no access to PIM data, such as the Calendar, and although you can *add* new phonebook entries, you can't *read* from the phonebook.
- There is no access to Bluetooth.

These restrictions can be a bit daunting at first, but when you consider that thousands of i-Appli have already been written and sold creating a multi-billion yen industry,⁶ it's clear that developers have devised a number of workarounds over time. Probably the biggest hurdle is the JAR size limitation, which often directly affects application architecture as well as porting strategies.

⁶ report.cesa.or.jp/english/whitepaper/index.html

A small JAR means that large assets such as video clips, high-resolution images and audio clips need to be stored and retrieved from remote locations. That's not too hard but it may require significant re-engineering of your code base when you're trying to port a MIDlet across to DoJa. Incidentally, it's worth noting that the latest DoJa 5.x versions in Japan allow up to 1 MB to be shared between the JAR file and the Scratchpad so, if you are working with these versions, it is obviously less of an issue.

One of the more unfortunate consequences of this size limitation is that developers must take dramatic steps to reduce the size of their executable. They usually adopt aggressive coding strategies that result in a poor code base which tends to be almost unreadable and extremely hard to maintain. I'll talk more about this in Section 7.6.

On the other hand, DoJa2.5oe has a very good game API – one that rivals the MIDP game API (and for now, mobile development is all about games). With support for the Mascot plug-in, you can get high-performance 3D graphics in DoJa pretty quickly and what's more this will take advantage of any hardware acceleration on the device. It also has a fully featured GUI implementation for thick-client applications. This is in many respects very similar to the Abstract Windows Toolkit (AWT) model from Java SE so the learning curve is not a steep one.

DoJa also has some features that MIDP doesn't have – applications can be notified of system events such as the 'ticking' of the system clock as real time passes, the folding of the handset (most DoJa phones tend to be clamshells) and activate or suspend events. You can also find out the reason an application was last suspended – allowing your code to resume with some context.

There is also a fixed mapping to the left and right softkeys so you don't have to hardcode magic numbers in your event handlers to work out which one was pressed – a simple mapping which MIDP *still* doesn't have (you can see an example of this in ***GhostPong*** in Chapter 8). And when an application is suspended, the framework automatically stops any audio playback in progress – given the number of MIDlets (mainly games) that either don't do this at all or do it sporadically, this is a *really* nice feature.

You can also create screen savers easily using the standby functionalities of the `MApplication` class and ADF configuration options. If you don't see any value in that, check out any Java ME forum – everyone wants to be able to do this in the MIDP world and, although it's part of the MIDP 3.0 proposal, they can't do it just yet.

Again, note the difference between DoJa, a profile designed to meet the specific demands of a particular customer market, and MIDP, a profile that tried to be all things to all people by staying generic. DoJa focuses on opportunities and product a lot more than mainstream Java ME but that's no surprise as they sprang from very different sources.

DoJa2.5oe consists of seven core packages (see Table 7.1). They are very similar to the standard MIDP packages with the notable exception of the more functional `com.nttdocomo.ui` package for GUI applications.

Table 7.1 DoJa2.5oe Core Packages

Package	Description
<code>com.nttdocomo.lang</code>	<code>UnsupportedOperation</code> and <code>IllegalState</code> exception classes
<code>com.nttdocomo.io</code>	Connection classes and exceptions for OBEX and HTTP
<code>com.nttdocomo.util</code>	Timers, events, encoders, and a phone utility class for making calls
<code>com.nttdocomo.net</code>	URL encoder and decoder
<code>com.nttdocomo.system</code>	Classes for sending SMS messages and adding new entries to the phonebook (although reading from the phonebook is unsupported)
<code>com.nttdocomo.ui</code>	Core widget classes for GUI development, sprite classes, and another phone utility class exposing vibration and backlight functionality
<code>com.nttdocomo.device</code>	Camera class, allowing pictures and video to be taken using the onboard camera

7.3 I Love JAM

Every DoJa project has a predefined directory structure as shown in Table 7.2. Although you haven't built anything yet, it is generated by Eclipse or the DoJa toolkit when you create a new project.

Table 7.2 DoJa Project Directory Structure

Directory	Description
<code>bin</code>	Holds the ADF file (<code>.jam</code>) and JAR executable
<code>res</code>	Location for any image, video or audio resources to be distributed with the application
<code>sp</code>	Location for the Scratchpad file
<code>src</code>	Source files location

In this section, we're concerned only with the Application Descriptor File (ADF) located in the `bin` project subdirectory. The ADF is the heart

of a DoJa application. It inherits its suffix (`.jam`) from the original Java ME days when it was tied in with the Java Application Manager (JAM) which evolved into the Application Management Service (AMS). The ADF is completely standardized and integrated into the execution framework and contains a large number of parameters most of which are optional. These primarily address security issues and can be thought of as taking the place of the API permissions in protection domains in MIDP.

DoJa is designed to be secure. If any of the parameters contain invalid values, the default behavior of the launch sequence is to prevent the application from running altogether. Most of these parameters can be left at their default values, but there are a few core parameters worth outlining here, as described in Table 7.3. For a comprehensive overview, consult the DoJa Developers Guide.⁷

The ADF provides a secure contract for DoJa applications. As can be seen above, some of the more interesting things that can be done include configuring an i-Appli to start periodically, in response to an incoming SMS or when the device goes into stand-by mode. The ADF drives and defines all of these behaviors in an easy-to-understand and consistent manner.

Table 7.3 ADF Parameters

Parameter	Description
PackageURL	The URL for the application binary; communication by the application is only possible to the host and port defined in this field
AppSize	JAR size in bytes, set during compilation
SPsize	Size of the Scratchpad in bytes; the maximum value here is device-dependent
AppParam	Start-up parameters (i.e. command-line input)
UseNetwork	Specifies if network communication is used; the only valid value is <code>http</code>
UseSMS	Specifies if SMS is used; the value is either blank or the string 'Yes'
LaunchBySMS	If MSISDN is set here, application launch via SMS is permitted
LaunchAt	Used when applications need to be launched automatically at regular intervals; the format is <code>I h</code> where <code>h</code> is the interval in hours.

⁷ www.doja-developer.net/downloads/index.php?node=42

7.4 DoJa Basic Operations Manual

You might now be wondering about some of the details involved in this whole process. When you get started, it's always good to know where to look when things don't work. In this particular case, it's difficult because most of the DoJa documentation has been translated from Japanese into English and tends to be a little terse.

Probably the best place to start is at the DoJa Developer Network (www.doja-developer.net), which has a large number of articles as well as beginner and advanced tutorials. Some are targeted at MIDP developers specifically. If you're serious about DoJa development, this is the place to start (look under the Features menu) and the best way to become effective quickly is to do all of the tutorials so that you can understand the idioms and terms used.

The last thing to mention about getting started is access to device hardware for testing. This is going to be difficult in most Western countries unless your operator has them and you want to sign up for one. It's a problem with no easy solution at this time but, given the stability and coherence of the platform, you can do most development with the emulator to start with. Perhaps in time we'll see a virtual testing lab online, much like Nokia has provided,⁸ to get around this issue.

7.5 Eclipsing DoJa

We'd better build something or I'll lose your interest, so let's start by setting up Eclipse for DoJa development using 2.5oe. The first thing to do is to install Eclipse, if you haven't already (you'll need at least version 2.1.1). If you chose the custom installation option when installing the DoJa toolkit, you've probably also installed the Eclipse plug-in. And that's all you need to do.

As an introduction to DoJa, and in the spirit of public health, we'll build a simple application for tracking body weight. Since mobile phones are often referred to as cell phones, we clearly need to call it **Cellulite**. Open Eclipse and choose New, Project from the File menu. Under the Java node, you should see a project template for DoJa 2.5oe as shown in Figure 7.2. Click on Next and enter the project name as shown in Figure 7.3 and then click Finish.

The DoJa plug-in requires version 1.1 class file compatibility, so the next step is to right-click the project and select Properties. Select the Java Compiler node and check the 'Enable project specific settings' option. Now uncheck the 'Use default compliance setting' and set the 'Generated

⁸ www.forum.nokia.com/main/technical_services/testing/rda_introduction.html

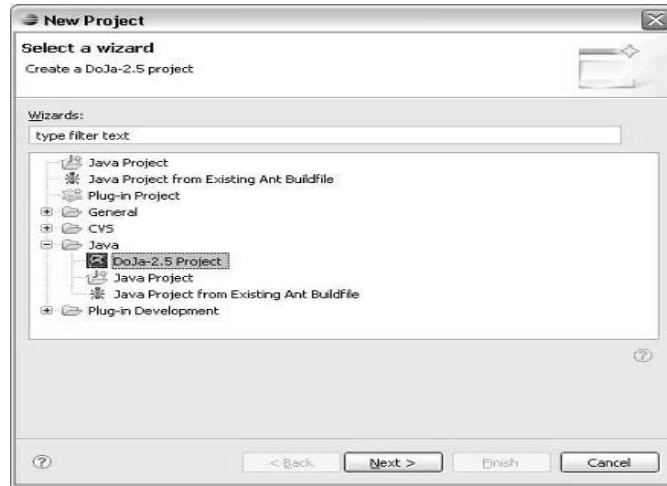


Figure 7.2 Choose the DoJa Project Template in Eclipse

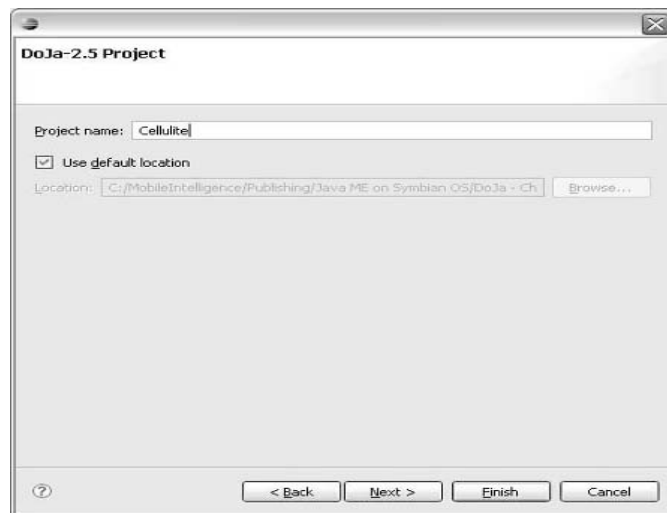


Figure 7.3 Creating the *Cellulite* project

.class files compatibility' option to 1.1. You also need to set the 'Source Compatibility' to 1.3, as shown in Figure 7.4.

You configure the ADF for a DoJa project by choosing DoJa-2.5 Setup from the Project menu (the Project node must be selected in the explorer or this option is grayed out). As you can see in Figure 7.5, there are a large number of configuration options available but for this project we only

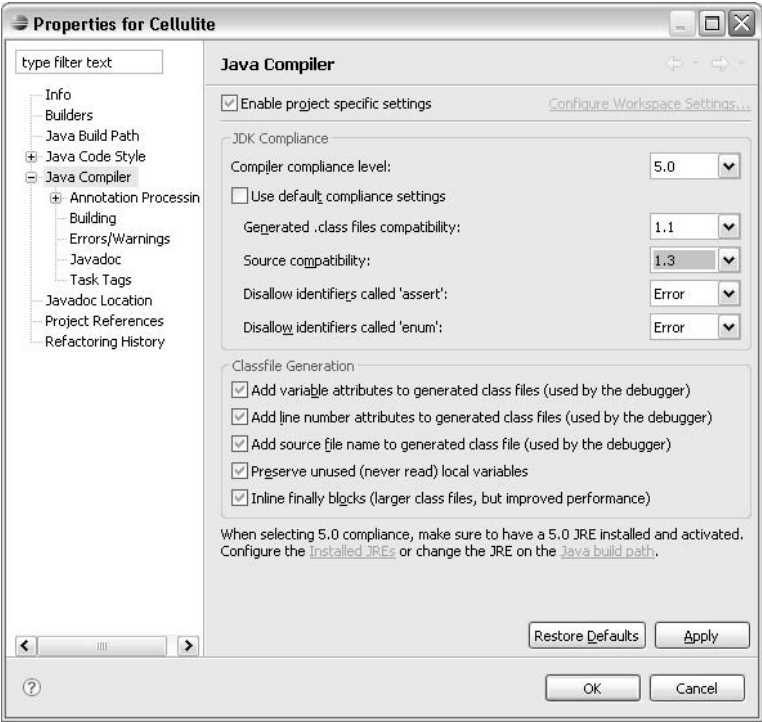


Figure 7.4 Compliance Settings for Cellulite

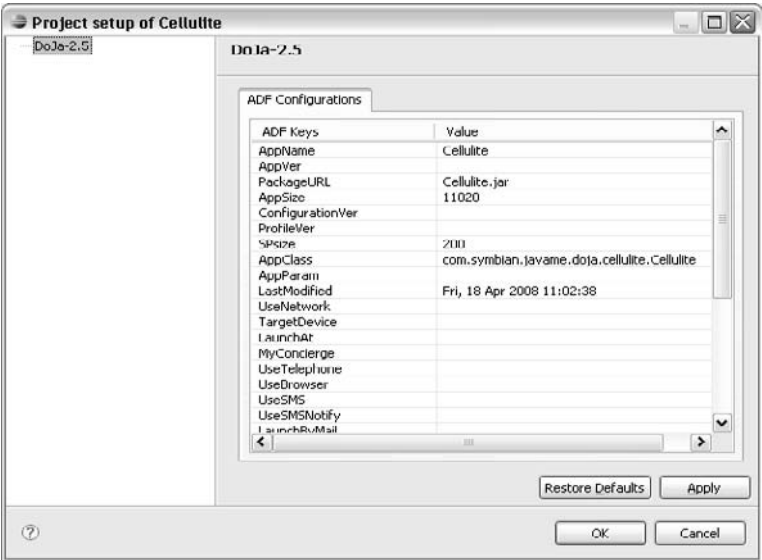


Figure 7.5 Configuring the ADF for Cellulite

need to ensure that the AppName and AppClass fields have the correct values if they haven't been set already.

You also need to be able to specify network settings for when your application uses sockets over OBEX or HTTP. We aren't going to use this here, but when you do need it, it can be hard to find, so select Preferences from the Window menu and select the DoJa-2.5 Environment node on the left to see the screen in Figure 7.6. This is where you specify the base URL for your application's ADF. Remember that i-Appli can only open sockets back to the server where the ADF is located (from which the application was downloaded) so you must set this value correctly to open connections successfully.

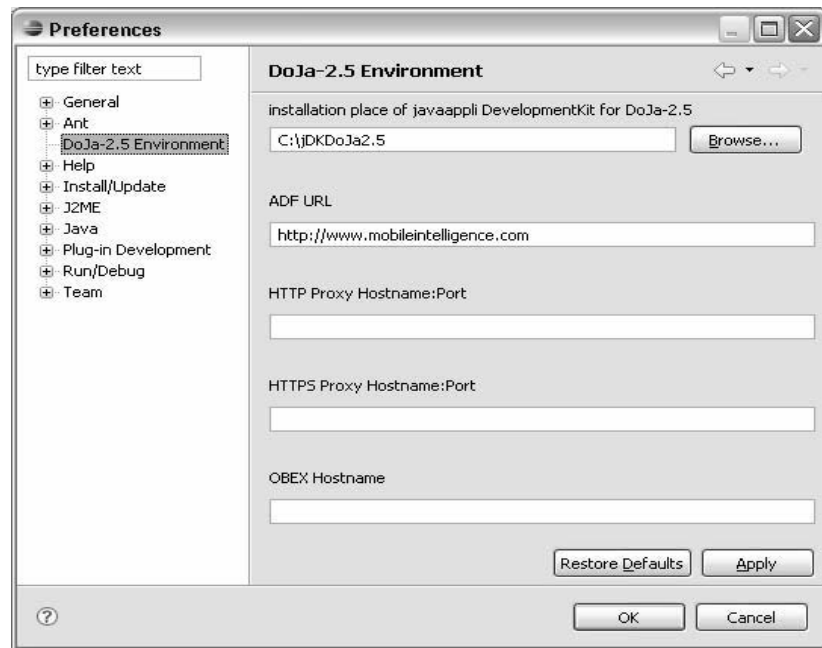


Figure 7.6 Project Network setup

7.6 Dirty Hands

Cellulite allows you to enter a series of data records containing a date and a weight in kilograms into a local 'database' (we'll use a `java.util.Vector` for this, so obviously it won't persist). You can add new records or edit existing ones and you can also produce a graph to display the trend. The application consists of three views (see Figure 7.7).



Figure 7.7 Application views for *Cellulite*

In this section, we run through some of the more relevant portions of the code but there isn't room for all of it, so you will need to download the source code from the website for this book.⁹

7.6.1 High-level API

Cellulite uses the high-level API for the main and data entry screens and the low-level API for drawing on the graph screen. The application consists of a main class that extends the `IApplication` abstract class as required by the framework:

```
package com.symbian.javame.doja.cellulite;
import com.nttdocomo.ui.*;
import java.util.*;
public class Cellulite extends IApplication{

    //constants
    private static String APP_TITLE = "Cellulite";
    private static final float WEIGHT_MIN = 1.0F;
    private static final float WEIGHT_MAX = 200.0F;

    // view ids
    private static final int VIEW_MAIN      = 0;
    private static final int VIEW_EDIT_ENTRY = 1;
    private static final int VIEW_GRAPH_DATA = 2;

    // data members
    private WeightEntry currentEntry;
    private Vector database;
    private int nextRecordId;
```

⁹ developer.symbian.com/javameonsymbianos

```
// from IApplication
public void start() {
    nextRecordId = 0;
    database = new Vector();
    // add some default data to the database
    database.addElement(new WeightEntry(nextRecordId++, 26,2,85.5F));
    database.addElement(new WeightEntry(nextRecordId++, 27,2,84.8F));
    SwitchView(VIEW_MAIN);
}
```

As a matter of good practice, all view switching is centralized in the `SwitchView()` method where each view is implemented by a specialization of either the `com.nttdocomo.ui.Canvas` class or the `com.nttdocomo.ui.Panel` class:

```
// switch between high-level and low-level screens based on viewId
private void SwitchView(int viewId){
    Frame view = null;
    switch(viewId){
        case VIEW_MAIN:{
            view = new MainView();
            break;
        }
        case VIEW_EDIT_ENTRY:{
            view = new EditWeightView(currentEntry);
            break;
        }
        case VIEW_GRAPH_DATA:{
            view = new GraphView();
            break;
        }
    }
    Display.setCurrent(view);
}
```

Error handling isn't comprehensive here – the main application class has a single method for it, which also shows how easy it is to provide user information using the built-in `Dialog` class:

```
// handle any errors in a consistent manner
private void showError(String message){
    Dialog dialog = new Dialog(Dialog.BUTTON_OK, APP_TITLE);
    dialog.setText( (message != null) ? message : "Unknown exception");
    dialog.show();
}
```

The main application view displays the list of records using DoJa widgets. This works in a similar manner to the AWT component model from Java SE – a class that uses widgets should subclass the `com.nttdocomo.ui.Panel` class, which acts as a container for a group of high-level controls.

When using this approach, the class should implement the `ComponentListener` interface, which allows your code to handle events raised by key presses, item selection events, and so on. Softkey events are handled slightly differently by implementing the `SoftKeyListener` interface. In this case, the `MainView` class acts as its own listener for both types of event, keeping all code localized as much as possible:

```
private class MainView extends Panel
    implements ComponentListener, SoftKeyListener{
    private ListBox listBox;

    public MainView(){
        setTitle(APP_TITLE);
        // populate list from database
        int count = database.size();
        listBox = new ListBox(ListBox.SINGLE_SELECT);
        listBox.append("Add New");
        for(int i = 0; i < count; i++){
            listBox.append( ((WeightEntry) database.elementAt(i)).toString());
        }
        listBox.select(0);
        add(listBox);
        // set self as listener
        setSoftLabel(Frame.SOFT_KEY_1, "Graph");
        setSoftLabel(Frame.SOFT_KEY_2, "Exit");
        setSoftKeyListener(this);
        setComponentListener(this);
    }

    // from SoftKeyListener
    public void softKeyPressed(int softKey){}
    public void softKeyReleased(int softKey) {
        try{
            if(softKey == Frame.SOFT_KEY_1){ // left SK
                if(database.size() == 0)
                    throw new Exception("No data to graph");
                SwitchView(VIEW_GRAPH_DATA);
            }
            else
                IApplication.getCurrentApp().terminate();// right SK : shutdown
        }
        catch(Exception e){
            showError(e.getMessage());
        }
    }

    // from ComponentListener
    public void componentAction(Component source, int type, int param){
        if(source == listBox){
            int index = listBox.getSelectedIndex();
            if(index == 0)
                currentEntry = new WeightEntry();// add new
            else // edit existing
```

```

        currentEntry = (WeightEntry) database.elementAt(index - 1);
        SwitchView(VIEW_EDIT_ENTRY);
    }
}
}

```

In the code fragment above, notice that `WeightEntry` is a simple private utility class used to encapsulate a data record – for simplicity, it doesn't have accessors or mutators, just public fields (i.e. it is basically a Java version of a C struct).

The `EditEntryView` class is similar to the main view as they're both high-level screens, however in this case a number of controls are used to allow the user to edit an existing record or to create a new one. Notice how this class shows that you don't have to implement the `ComponentListener` interface. For a simple input screen like this, we can cheat a bit and use the softkey events to handle data validation and updating the database:

```

private class EditWeightView extends Panel implements SoftKeyListener{

    private ListBox cboMonths, cboDays;
    private TextBox txtWeight;
    private int entryId;

    public EditWeightView(WeightEntry weightEntry){
        entryId = weightEntry.Id;
        // populate date controls
        int lastDay = Utilities.getLastDayOfMonth(weightEntry.Month,
            Calendar.getInstance().get(Calendar.YEAR));
        cboDays = new ListBox(ListBox.CHOICE);
        for(int i = 1; i <= lastDay; i++){
            cboDays.append(" " + i);
        }
        cboMonths = new ListBox(ListBox.CHOICE);
        for(int i = 0; i < 12; i++){
            cboMonths.append(Utilities.monthName(i));
        }
        cboDays.select(weightEntry.Day - 1);
        cboMonths.select(weightEntry.Month);
        txtWeight = new
            TextBox(Float.toString(weightEntry.Weight), 5, 1, TextBox.NUMBER);

        setTitle("Weight Data");
        add(new Label("Day"));
        add(cboDays);
        add(new Label("Month"));
        add(cboMonths);
        add(new Label("Weight (kg)"));
        add(txtWeight);
    }
}

```

```

// set self as listener
setSoftLabel(Frame.SOFT_KEY_1, "Save");
setSoftLabel(Frame.SOFT_KEY_2, "Back");
setSoftKeyListener(this);
}

// from SoftKeyListener
public void softKeyPressed(int softKey){ }
public void softKeyReleased(int softKey) {
    try{
        if(softKey == Frame.SOFT_KEY_1) { // Save
            float weight = getInputWeight();// may throw exception
            int day = Integer.parseInt(
                cboDays.getItem(cboDays.getSelectedIndex()));
            int month = cboMonths.getSelectedIndex();
            if(entryId == -1) // new record
                database.addElement(
                    new WeightEntry(nextRecordId++, day, month, weight));
            else // update database with specified values
                updateWeightRecord(entryId, day, month, weight);
            SwitchView(VIEW_MAIN);
        }
        else if(softKey == Frame.SOFT_KEY_2) // Back
            SwitchView(VIEW_MAIN);
    }
    catch(Exception e){
        showError(e.getMessage());
    }
}
}

```

In the listing above, the methods `getInputWeight()` and `updateWeightRecord()` are not shown – these are private methods in this class. Their full implementation is included in the downloadable source code. For convenience, a number of methods not shown above that are useful for date information management have been included in a separate static class called `Utilities`.

7.6.2 Low-level API

Let's move away from the high-level API now and have a look at using the low-level one. There's nothing particularly special about this and certainly the basics are very similar to what you are used to with MIDP 2.0. The main difference to be aware of when you get started is in how double buffering is handled. In MIDP, you can do it yourself or, if you're using a Symbian implementation, the `Canvas` class is double-buffered by default. It's a little different in Doja – you use two methods on the graphics context called `lock()` and `unlock()` and place your drawing code between them, as you'll see below.

The `GraphView` class subclasses `com.nttdocomo.ui.Canvas` and draws a simple line graph representing the distribution of weights in the

database. I've cheated a bit here and actually graphed weight against record number rather than date (otherwise the records would need to be kept sorted) but it conveys the general idea.

When you work with a canvas, event handling is more generalized – all events are passed to the `processEvent(int type, int param)` method of the canvas and you handle or ignore them as required. The `type` parameter is a static field from the `Display` class and the `param` parameter varies with the event type. In other words, you do not need to implement an interface to handle events in DoJa at a low level – you simply override the `processEvent()` method of the `Canvas` class.

Looking at the source, you'll see a large amount of code in the constructor – this is simply used to pre-calculate as much as possible (such as range mapping and where labels need to go, for example) in one hit and it isn't relevant to this discussion. However take note in the listing below that `Canvas` still supports softkey events and, in fact, that's how we handle navigating back to the main screen from here:

```
private class GraphView extends Canvas{

    private int[] xpoints, ypoints;
    private int ox,oy,xmax,ymax;
    ...

    public GraphView(){
        dataCount = database.size();
        ...
        xmapFactor = (float) (xmax - ox) / (float) dataCount; // never zero
        ymapFactor = (float) (oy - ymax) / (rangeMax[0] - rangeMin[0]);
        xpoints = new int[dataCount];
        ypoints = new int[dataCount];
        ...
        setSoftLabel(Frame.SOFT_KEY_1, " ");
        setSoftLabel(Frame.SOFT_KEY_2, "Back");
    }

    public void paint(Graphics g){
        g.lock();// double buffered drawing sequence
        // draw axes
        g.setColor(Graphics.getColorOfName(Graphics.BLACK));
        g.drawLine(ox,oy,ox,ymax);
        g.drawLine(ox,oy,xmax,oy);
        // draw vertical axis labels
        g.drawString(ymaxValue, maxLabelX, maxLabelY);
        g.drawString(yminValue, minLabelX, minLabelY);
        // draw graph
        g.setColor(Graphics.getColorOfName(Graphics.RED));
        g.drawPolyline(xpoints, ypoints, dataCount);
        g.unlock(true);
    }

    public void processEvent(int type, int param){
        if(type == Display.KEY_RELEASED_EVENT){
            if(param == Display.KEY_SOFT2){ // Back
```

```
try{
    SwitchView(VIEW_MAIN);
}
catch(Exception e){
    showError(e.getMessage());
}
}
```

7.6.3 The Big Squeeze

The biggest hurdle with DoJa development for those of us who use the overseas versions is keeping your JAR size below 30 KB. It turns out that JAR size varies directly with the number of classes used, the number of methods in those classes and the length of the identifiers used for methods, fields and classes (yes, really!). This is where the use of an obfuscator such as ProGuard¹⁰ can be a real life saver.

Originally designed to help protect intellectual property, an obfuscator can shrink classes dramatically by shortening names of packages, classes, methods, fields and variables. They can also add a number of optimizations while allowing you to work from a code base that can be understood.

In terms of your code base, don't fall into the trap of trying to manually optimize your classes – the rule of thumb should be to let an obfuscator do this for you. Re-architecting an application to satisfy an external constraint such as this moves your focus away from realizing a correct software solution and starts to reduce your approach to the level of a hack.

As an example, if you configure Eclipse to use ProGuard as an obfuscator, the executable size drops down to 8.79 KB from 10.7 KB – a reduction of just over 17%. For a simple application, this is more than enough and, in many cases, reductions of up to 20% are common.

You can further reduce the size of *any* Java binary by optimizing resource management – for example, large resources can be downloaded from remote servers when required rather than being bundled in with the JAR file. If you need to keep resources locally accessible, you can look at reducing their sizes – for example, an image may still be acceptable in your application with less color depth or at smaller dimensions.

7.7 A Safe Port

There are a number of issues to be addressed when porting an application from MIDP across to DoJa. While they're both built on top of CLDC, the

¹⁰ proguard.sourceforge.net

DoJa API differs from MIDP in many ways. Further, as we've seen, DoJa introduces a number of external technical limitations on functionality that will require a careful porting strategy. In some cases, it may even be necessary to significantly re-architect your application altogether. In most cases, the workarounds involve moving local data to a server and downloading it as required.

7.7.1 Application Lifecycle

In MIDP, the AMS calls the `startApp()`, `pauseApp()` and `destroyApp()` methods of the `MIDlet` class in response to operating system events or requests from the `MIDlet` to change its state.

In DoJa, the `IApplication` class that all applications must extend simply has the `resume()` method for lifecycle management. There is no pausing – an application is either running or it is suspended. You can override the `resume()` method as your application requirements dictate. When a suspended application is resumed, the operating system calls this method as well as sending a `Display.RESUME_VM_EVENT` event to the currently visible screen.

This has a direct effect on porting the pause–resume cycle used in MIDlets. Normally on a pause event, you do tasks such as persisting application state and releasing resources. You then code your `MIDlet` so that it re-acquires these resources when the application re-starts. In DoJa however, the application is already in the same state that it was when it was interrupted. For example, any sound will resume playing from its current location. Clearly, the lifecycle differences between MIDP and DoJa applications complicate the porting process and require careful attention and planning.

7.7.2 Local Storage

Applications that use the RMS for local persistence (particularly database-style applications) need to be changed to use the Scratchpad which has a maximum size of 200 KB. Apart from this size limit, there is no high-level support for the concept of a record since the Scratchpad is a flat, random-access file. Depending on your requirements, you may need to work out your own method for storing, searching and updating records which can be a fair amount of work. Alternatively, you could implement a hybrid solution and store the data on a server and cache local data in the Scratchpad.

Many MIDlets use the RMS as a local cache for large binary assets such as audio, video or images. If your assets are less than 200 KB, they can be stored in the same manner as byte streams, using the Scratchpad. If not, either reduce the size or move them to a server and pull them down as required.

7.7.3 Double Buffering

As we've already seen, double buffering is supported by using the `lock()` and `unlock()` methods of the `Graphics` class. The Symbian OS implementation of the MIDP `Canvas` class is double buffered by default, or you use an offscreen image context to create your own double buffer. This is not a major problem and changing your code base to deal with it should be fairly easy.

7.7.4 Dialogs

DoJa uses the `Dialog` class in place of MIDP's `Alert` class. A `Dialog` is a fixed-function, very simple modal dialog and the following MIDP functionality is not available:

- automatic re-direction to a specified `Displayable` when the dialog is dismissed
- playing an appropriate (and device dependent) alert sound on display
- using your own images – DoJa dialogs do not have any images
- adding your own command buttons and listeners
- timeouts – all dialogs are modal
- activity indicators.

All you can do is simplify your alert management code to address this. Applications that rely on post-dismissal re-directions are likely to require the most re-engineering.

7.7.5 Softkeys

DoJa allows any screen to specify up to two softkeys and there is no equivalent of the `Command` class, therefore there is no support for menus. Depending on your application, this may require a careful re-design of your application's navigation flows.

7.7.6 Text Positioning

The anchor point concept used extensively in MIDP is not used in DoJa. When drawing text on a graphics context at a specified point, that point is always relative to the left edge of the text and its baseline. Working around this is a simple exercise in mathematics.

7.7.7 Key Presses

The `Canvas` class in MIDP supplies the `keyPressed()`, `keyRepeated()` and `keyReleased()` methods. They do not exist in DoJa, where all event handling is handled by the `processEvent()` method of the canvas. You need to explicitly check for the type of the event (e.g. `Display.KEY_PRESSED_EVENT`) and respond to it appropriately.

7.7.8 Sockets

In DoJa, connections can only be opened to the server specified in the application's ADF. DoJa also limits these to HTTP and OBEX connections. MIDP applications that use raw sockets or datagram connections need a lot of work to address this one.

The other issue to remember here is that there are size limits for both uploading (5 KB) and downloading (10 KB) over these connections. This can be problematic as an attempt to download, for example, a 50 KB image does not fail – you just get the first 10 KB.

There are a number of workarounds for this, such as incremental downloading and local caching. The DoJa Developer Network (DDN) has a good tutorial¹¹ that discusses a variety of techniques commonly used to deal with this issue.

7.7.9 Bluetooth Wireless Technology

Applications that rely on Bluetooth are in trouble unless you're using at least DoJa 5.0. There is no easy way around this limitation.

7.7.10 Audio Playback

In DoJa, the `AudioPresenter` class is used instead of MIDP's `MMAPI Player` concept. It uses a different lifecycle and does not support the WAV audio format – audio files in DoJa are either in the Melody For i-mode (MFi) format or Standard MIDI File (SMF) format. The MFi format (contained in `.mld` files) is the sound format used in Flash animations, however the general development community avoids using it due to the fact that MFi generation software is not publicly available and there are a wide variety of versions to manage.¹² The easiest approach to take here is to simply use MIDI.

¹¹ www.doja-developer.net/_up/features/feature1.7/index.php?id=44

¹² There is a good review of this topic on the MDL website: groups.google.com/group/mobiledevlab/web/creating-sounds-for-i-applis?hl=en

Porting audio is tricky, particularly if your MIDlet has extensive reliance on player lifecycle events. Also, there is no support for pitch, tempo or volume controls and the current state of the presenter cannot be queried either.

7.7.11 Game Development

Although you can build altruistic little GUI applications, such as *Cellulite*, the real money is in game development. It was the games market that drove mobile development technologies in the first place and it all started in Japan. Needless to say, this market is the largest and most dynamic in the world. For a great review of how easy it is to build DoJa games, check out the game development tutorials on the DDN.¹³ We talk a lot more about games in Chapter 8, so we will say no more here.

7.8 DoJa 5.1 Profile

The FOMA 905i series of handsets, released in November 2007, was the first handset series to implement the new DoJa 5.1 profile. Series prior to this (specifically 903i, 904i, 703i, and 704i) used the only slightly older DoJa 5.0 profile.

One of the impressive features of NTT's FOMA 3G technology is its high-speed data communications. The network communication size for HTTP connections is 80 K up and 150 K down (as opposed to the 10 K up and 20 K down for Personal Digital Cellular [PDC] handsets – a 2G standard developed and used exclusively in Japan)¹⁴ with download speeds of up to 3.6 Mbps.¹⁵ This, combined with NTT's flat-rate data charges, means that mobile application developers targeting this market have the freedom to design applications that take full advantage of high-speed, low-cost data transmission.

The i-Appli development environment comprises four layers of APIs. The first layer is the Java ME CLDC API, on which DoJa is built. This is a point of commonality with the MIDP architecture where there is a larger English-speaking development community. The remaining layers are the i-Appli Basic API, the i-Appli Optional API, and the i-Appli Extension API (see Figure 7.8).

The exact boundaries between the Optional and Extension APIs are not clear, and it is usual for the DoJa documentation to lump these two

¹³ www.doja-developer.net/features/index.php?node=62

¹⁴ See [NTT DoCoMo 2007]

¹⁵ 3.6 Mbps is the maximum receiving data rate when using a FOMA HIGH-SPEED compatible handset in a FOMA HIGH-SPEED area (see www.nttdocomo.co.jp/english/service/function/high_speed/).

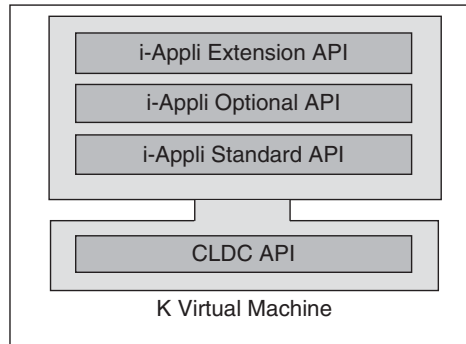


Figure 7.8 i-Appli Runtime Environment Architecture¹⁶

APIs together when referring to them. It is at the handset manufacturer's discretion as to whether or not classes or methods in the Optional and Extension APIs are implemented, so in practice there is no difference between the two. In the API Java documentation, any class or function that is not part of the Basic API is clearly marked 'optional' and throws an `UnsupportedOperationException` if it is invoked on unsupported hardware.

DoJa 5.0 increased the size of the i-Appli binary to 1 MB. Additionally, it removed the separate size limitations on the JAR and the Scratchpad; now the only limit is that the combined size is less than 1 MB. This way of treating the code and resources – having a combined maximum file size – is identical to the method used in MIDP. While there are still many other differences between the two specifications, the removal of the JAR size restriction greatly increases the ability to port applications between the two specifications.

Many of the exciting new features implemented in the Fujitsu 905i and Fujitsu 906i handset series, which implement the DoJa 5.1 profile, exist within the optional parts of the API. While some features are common to all of these handsets, other features (such as the acceleration sensor and speech recognition) are only implemented by some handsets. Some additional features, such as biometric fingerprint authentication, are specific to individual handsets.

Along with the new profile come new developer tools. The emulator for DoJa 5.1 includes support for the Net Beans 4.1 and Eclipse IDEs. There are also various third-party tools to facilitate the creation of 3D models to use with the Mascot Capsule Micro3D Engine and sound creation utilities. The 3D-modeling tools can be downloaded for free directly from the Mascot Capsule website,¹⁷ but many of the official tools

¹⁶ www.nttdocomo.co.jp/english/service/imode/make/content/iappli/about/index.html#architecture

¹⁷ www.mascotcapsule.com/toolkit/docomo/en/index.php

for creating sounds in the various formats used on the handsets are only available to certified content developers. (A handset may not necessarily be able to play sounds created for another handset.)

7.8.1 Packages

This section explains each of the sub-packages included in the `com.nttdocomo` package. Table 7.4 lists the core packages in DoJa 5.1.

Table 7.4 DoJa 5.1 Core Packages

Package	Description
<code>com.nttdocomo.io</code>	This package includes classes to stream text efficiently and other complementary classes. DoJa uses the underlying CLDC 1.1 framework for creating input and output stream connections. This package uses a URL format to connect to network resources via HTTP and HTTPS, and to create OBEX connections via the infrared port or UART connections via serial cable.
<code>com.nttdocomo.net</code>	This package contains the <code>URLEncoder</code> and <code>URLDecoder</code> utility classes for HTML form encoding and decoding.
<code>com.nttdocomo.util</code>	This package contains various utility classes, including the <code>Phone</code> and <code>JarInflater</code> classes.
<code>com.nttdocomo.lang</code>	This package includes the <code>XObject</code> and <code>XString</code> classes that allow native phone data management. These classes are used to allow access to various pieces of user data while restricting operations on that data so that, for example, a user's phonebook data cannot be uploaded to a server.
<code>com.nttdocomo.ui</code>	This package contains classes used for creating application user interfaces.
<code>com.nttdocomo.ui.graphics3d</code>	This package has been available since the DoJa 4.0 profile. It contains a number of classes that enable 3D rendering.
<code>com.nttdocomo.ui.graphics3d.collision</code>	This package is new to the core DoJa 5.0/5.1 API. It provides classes to facilitate calculation and collision detection within 3D coordinate space.
<code>com.nttdocomo.ui.sound3d</code>	Included since DoJa 4.0, this package allows 3D coordinate data to be associated with a sound, which is played as if from a source in 3D space.

Table 7.4 (continued)

Package	Description
<code>com.nttdocomo.ui.util3d</code>	Included DoJa 4.0, this package contains utility functions, such as math and vector arithmetic, for 3D graphics and sound.
<code>com.nttdocomo.ui.opengl</code>	This package implements the OpenGL ES1.1 API.
<code>com.nttdocomo.device</code>	This package contains the classes for controlling the phone's peripheral devices. In DoJa 5.x, optional sub-packages provide classes to control and query the gesture reader, acceleration sensors and FeliCa wireless chip.
<code>com.nttdocomo.system</code>	This package contains classes and interfaces for utilizing the phone's native functionality, including the phone-book, scheduler, and bookmarks. Many of these classes are only enabled for trusted i-Appli.
<code>com.nttdocomo.security</code>	This package contains classes to manage security certificates and digital signatures. Many of these classes have been newly added to the DoJa 5.1 profile.

Unlike the optional classes and methods in the core API packages listed in Table 7.4, all classes and methods in the packages listed in Table 7.5 are optional and they may not be present in all handsets. When one of these packages is included in the Extension API, its operation may be different depending on the manufacturer. Any call to a method that is not supported on the target hardware causes the framework to throw an `UnsupportedOperationException`.

Table 7.5 Optional Packages

Package	Description
<code>com.nttdocomo.opt.device</code>	This package provides classes to control optional phone devices which are handset specific, for example, a second camera or fingerprint authenticator.
<code>com.nttdocomo.opt.ui</code>	This package contains classes to create application user interfaces, extending the functionality of similar classes in the base API. For example, the <code>Canvas2</code> class allows for changing the canvas orientation. Also included are classes to control optional phone features, such as the sub-display.
<code>com.nttdocomo.opt.ui.j3d</code>	This package provides extended 3D graphics rendering and 3D calculation capabilities.
<code>com.nttdocomo.opt.ui.opengl</code>	This package implements the OpenGL ES1.1 extension API.

7.8.2 2D Animation

Video playback is supported as part of the basic API. GIF animation playback and 2D animation playback is supported by means of the `VisualPresenter` class.

The file format for animation data is not specified, meaning that the supported animation data formats vary among the different handset manufactures. The formats supported by each handset, and the ways to create animation data, are released separately by the manufacturer.

The only new thing for the DoJa 5.x profile is the addition of the attributes to set the `VisualPresenter` for full-screen playback – either by forcing full-screen or setting a preference for full-screen playback. This is part of the Optional API and is handset dependent.

7.8.3 3D Graphics

The 3D graphic features in DoJa 5.0 allow for the loading and display of external 3D model data. Previous DoJa profiles split the 3D-graphics-rendering API into high-end and low-end APIs, each with their own specification. Since DoJa 4.1, the low-end API has been removed and new functionality has been added in the 5.x profile.

Support for the Mascot Capsule Micro3D Engine has been included since DoJa 4.0. All phones supporting the DoJa 5.1 profile support both versions 3 and 4 of the Mascot Capsule Engine.

There are two `Graphics3D` classes included in the API. One is in the `com.nttdocomo.ui.graphics3d` package and the other is in the optional `com.nttdocomo.opt.ui.j3d` package. The latter has functionality for ambient and directional lighting, texture mapping, affine transformations, 3D vector math, and other advanced 3D graphics operations. Both packages use the Mascot engine for rendering.

Using the Mascot Capsule 3D-graphics-rendering functions is a relatively simple process. A call is made to the `Canvas.getGraphics()` function to retrieve the current `Graphics` object. This is then cast to an object of the `com.nttdocomo.ui.j3d.Graphics3D` interface, which uses the Mascot Capsule Micro3D Engine for rendering.

```
Graphics g = Canvas.getGraphics();  
Graphics3D g3D = (com.nttdocomo.ui.j3d.Graphics3D) g;
```

This could be considered the ‘default’ method for 3D rendering. Free modeling tools and documentation can be downloaded from the Mascot Capsule website.

OpenGL ES support is new to DoJa 5.x. The ES specification consists of a well-defined subset of the standard OpenGL profile, which should further assist developers in porting existing applications to DoJa devices.

You can use the same casting procedure to create a `com.nttdocomo.ui.opengl.GraphicsOGL` interface object, which uses the OpenGL ES engine for rendering:

```
Graphics g = Canvas.getGraphics();
GraphicsOGL g3D = (com.nttdocomo.ui.opengl.GraphicsOGL) g;
```

Further documentation is available from the Khronos Group;¹⁸ it not only describes each method of the `GraphicsOGL` class in much greater detail than the DoJa API, but it is also in English. Unfortunately, there is a notable lack of any coded examples for DoJa.

7.8.4 Sound

The DoJa 5.0 profile builds on previous versions by allowing playback of iMotion audio tracks. Support is also enabled for the simultaneous playback of four MFi/Phrase format sounds and PANPOT (stereo-sound-effect positioning).

3D sound, including PANPOT, has been a feature implemented in DoJa since version 4.0. Supported on handsets with top and bottom speakers, as well as left and right speakers, these functions provide fine-grained directional output. The three-dimensional sound playback features allow sounds to be played from a virtual sound source, giving the illusion of orientation and direction. 3D coordinate data can be specified and updated either dynamically at run time or by embedding coordinate data in the sound.

PANPOT allows sound to be directed through either the left or right speakers. The attribute value ranges from 0 to 127, with 64 being the central position. It can only be used on sounds in the MFi/Phrase format, and only where supported by the handset.

The following short code extract demonstrates how to set the location of a sound using the `CartesianListener` to specify the position and orientation of the sound:

```
MediaSound mediaSound;
AudioPresenter audioPresenter;
Audio3D audio3d;
...
// acquire a new audio presenter and set the media sound
audioPresenter = AudioPresenter.getAudioPresenter();
audioPresenter.setSound(mediaSound);
// set up the 3d audio to be controlled by the application
audio3d = audioPresenter.getAudio3D();
audio3d.enable(Audio3D.MODE_CONTROL_BY_APP);
```

¹⁸ www.khronos.org/opengles/1.X

```
...
CartesianListener listener = new CartesianListener();
listener.setLookAt(new Vector3D(0f, 0f, 0f), new Vector3D(0f, 0f, -1f),
                    new Vector3D(0f, 1f, 0f));
cartPos = new CartesianPosition(listener);
audio3d.setLocalization(cartPos);
```

7.8.5 Serial Communications (UART)

Some FOMA handsets are equipped for serial communications via a cable (Universal Asynchronous Receiver/Transmitter, UART) connection. For supported handsets, data can be transferred to a PC via USB cable by opening a connection using the `Connector.open()` methods and passing in a URL in the following form:

```
comm:/0 [ ; parameter, ... ]
```

For example:

```
InputStream in = Connector.openDataInputStream("comm:/0;baudrate=38400");
int flag = in.read();
```

7.8.6 DTV and Radio Tuner

For handsets equipped with digital television tuners, the DoJa 5.x API includes DTV launch integration features, as well as allowing for the registration of recording timers.

Handsets may optionally be equipped with radio tuners, which can also be controlled via the DoJa 5.x API. This is new functionality added in DoJa 5.0. The radio can be used in stand-by applications and is set to continue playback even when the phone is in a dormant state.

7.8.7 Acceleration Sensor and Electronic Compass

The electronic compass is an optional feature added in the DoJa 5.0 profile. It provides the ability to collect the kind of location information associated with a traditional compass.

Perhaps one of the more exciting new features of the DoJa 5.1 profile is the acceleration sensor. This sensor allows for the collection of acceleration data along the x, y, and z axes, as well as pitch and roll information, and screen orientation. This functionality appears to be clearly directed towards game developers and allows for some quite engaging and innovative applications.

The following simple program uses the acceleration sensor to determine the orientation of the phone. The program draws an arrow on the

screen that always points to the top of the screen. If you turn the phone on its side, the program determines which side is now the top and updates the arrow accordingly.

```
import com.nttdocomo.ui.*;
import com.nttdocomo.device.location.*;
public class AccelDemo extends IApplication {
    public void start() {
        MainCanvas canvas = new MainCanvas();
        new Thread(canvas).start();
        Display.setCurrent(canvas);
    }
}

class MainCanvas extends Canvas implements Runnable {
    AccelerationSensor sensor;
    MainCanvas() {
        setSoftLabel(SOFT_KEY_1, "END");
        setBackground(Graphics.getColorOfName(Graphics.BLUE));
        sensor = AccelerationSensor.getAccelerationSensor();
    }
    int[] angle = null;
    public void run() {
        Graphics g = this.getGraphics();
        while(true) {
            sensor.start(sensor.getIntervalResolution());
            try {
                Thread.sleep(100);
            }
            catch(Exception e){}
            sensor.stop();
            AccelerationData data = sensor.getData();
            angle = data.getScreenOrientation();
            sensor.disposeData();

            redraw(g);
        }
    }
    public void paint(Graphics g) {
    }

    public void redraw(Graphics g) {
        g.lock();
        g.clearRect(0, 0, Display.getWidth(), Display.getHeight());
        g.setColor(Graphics.getColorOfName(Graphics.WHITE));
        if(angle != null){
            g.drawString(" "+angle[0], 10, 15);
            switch(angle[0]){
                case 0:
                    g.drawLine(120, 120, 120, 0);
                    g.drawLine(120, 0, 130, 10);
                    break;
                case 90:
                    g.drawLine(120, 120, 0, 120);
                    g.drawLine(0, 120, 10, 130);
                    break;
                case 180:
                    g.drawLine(120, 120, 120, 240);
```



```

        g.drawLine(120, 240, 130, 230);
        break;
    case 270:
        g.drawLine(120, 120, 240, 120);
        g.drawLine(240, 120, 230, 130);
        break;
    }
}
g.unlock(true);
}
public void processEvent(int type, int param) {
    if (type == Display.KEY_RELEASED_EVENT) {
        if (param == Display.KEY_SOFT1) {
            (IApplication.getCurrentApp()).terminate();
        }
    }
}
}
}

```

7.8.8 ToruCa

ToruCa is a feature that was introduced in the FOMA 902i series of phones (Doja 4.1 and later). ToruCa are electronic ‘business cards’ that transfer information that can be captured by an *osaifu-keitai* (e-wallet mobile phone).

Information such as business cards or restaurant fliers and promotional coupons can be downloaded onto the phone by waving it in front of a ToruCa terminal at a store. Information is automatically stored on the phone and can be browsed, searched, or exchanged via infrared or mail.

Businesses offering coupons or information via ToruCa in Japan include ANA and JAL Airlines, Tower Records, East Japan Railway Co, and many others.

The Doja API allows an application developer to access ToruCa information stored on the phone or to launch applications via commands executed when ToruCa information is received by the handset.¹⁹

7.8.9 FeliCa

An *osaifu-keitai* is equipped with an embedded wireless chip. FeliCa is the technology, originally developed by Sony, which enables contactless, highly secure, two-way data transfer between the FeliCa chip in the handset and a compatible terminal. In short, it allows the handset owner to pay for goods and services using the phone, and be billed on their monthly statement.

The FeliCa functionality in the Doja 5.x API enables application developers to create interactive i-Appli that allow online registration and

¹⁹ See www.nttdocomo.co.jp/english/service/osaifu/toruca.html and www.nttdocomo.co.jp/service/osaifu_shopping/toruca/image/index.html

deletion of services to the FeliCa chip, and reading and writing data within the FeliCa chip.

```
try {
    Felica.open(); // open the FeliCa chip
    FreeArea fa = Felica.getFreeArea(); // obtain a Free Area object
    int[] index = new int[1];
    index[0] = 0;
    String writeData = new String("test data");
    // write the data to the free area
    fa.write(index, writeData.getBytes());
    //read data from the free area
    String readData = new String(fa.read(index));
    Felica.close(); // close the FeliCa chip
} catch (FelicaException fe) {
    fe.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

The API is, naturally, rather complex, but a good overview of the technology is available at www.sony.net/Products/felica/abt/index.html. There is more information about FeliCa on the NTT DoCoMo website.²⁰

7.8.10 Other Features

DoJa 5.x implements a few new security features and continues to provide support for others.

The fingerprint authenticator, although a novel security feature, is only implemented by Fujitsu handsets. Application developers are able to authenticate the identity of the user by using the fingerprint authenticator via the `com.nttdocomo.opt.device.FingerprintAuthentication` class.

Interestingly, the fingerprint sensor can also be used to capture navigation data. The sensor can capture the position and direction of movement of a user's finger as it moves across the sensor. The sensor is also used by the phone's native browser to allow the user to scroll up and down a web page.

Included in the Basic API and enabled on all DoJa 5.1 devices are several classes and interfaces that enable speech recognition. Speech recognition has been available in DoCoMo devices for several years – the pre i-Mode N208s released back in 1999, for example, could search for a name in the address book when the user said the name. However, support is geared towards Japanese – which has a much smaller phonetic alphabet than English.

²⁰ See www.nttdocomo.co.jp/english/service/imode/make/content/felica and www.nttdocomo.co.jp/english/service/osaifu/index.html

Finally, the `com.nttdocomo.security` package contains several classes, many optional, that provide encryption and decryption functionality. The common PKCS #5 (RFC2898) Password-Based Encryption specification is supported. Using these classes, data can be safely exchanged between an i-Appli and an external server.

7.9 Summary

There has been a lot of new information in this chapter, so we'll keep this short. We talked about the market in Japan and some of the history of mobile technologies. We saw that even though the world is excited today about 3G networks, Japan had the first 3G network (FOMA) way back in 2001. We've also learned how to develop applications using DoJa 2.5oe and Eclipse as well as what to keep in mind when trying to port MIDlets across to the DoJa environment.

Sam Cartwright has given us a rare glimpse of some advanced mobile phone technologies that we can hope to see emerging in the rest of the world over the next decade or so. He also detailed the functionality available in the very latest DoJa 5.1 handsets. It doesn't get more cutting edge than this.

DoJa is a whole area of Java ME technology that we can only view at a distance from outside Japan – but what they're doing over there now makes the future look all the more exciting for the rest of us. Whatever it entails, you can be sure that Java ME technologies will form the nucleus of the next generation of mobile games and applications.

8

Writing MIDP Games

The Game API that came with MIDP 2.0 made mobile phone game development a thing of beauty. Although hundreds of games were written, sold and shipped on MIDP 1.0 devices, ushering in (and indeed creating) an exciting new industry for mobile games, it was MIDP 2.0 that elevated art into science.

At the start of this chapter, we investigate some general game concepts as well as those specific to mobile phones, look at how the MIDP 2.0 Game API makes Java ME an easy choice for mobile game development and the benefits that Symbian OS brings to the whole picture. We also throw together a very quick and simple game to demonstrate some Game API basics. A brief analysis of this first game prepares us for the more advanced concepts and techniques discussed in Section 8.4.

Section 8.4 focuses on more advanced game development on Symbian OS using the Java ME optional JSR libraries which are now standard on Symbian OS as part of MSA support. In this section, we discuss the design, planning and implementation of a game which aims to demonstrate a wide variety of techniques you can add to your toolkit. Many of these are only feasible on Symbian OS as many other mobile platforms have built-in constraints on MIDlets.

In addition to showing correct use of the core Game API, we also aim to show how to use the Mobile Media API (JSR-135) for audio effects, see how the Scalable 2D Vector Graphics API (JSR-226) can be used to build a compelling menu system, review the use of the Mobile 3D Graphics API (JSR-184) for the game world and see how we can use the Bluetooth API (JSR-82) to add multiplayer functionality to our game so that two players can play it against each other in a head-to-head match.

Throughout the chapter, only the most relevant sections of code are shown as there is only limited space available and a lot to cover. To that end, we recommend downloading the source code from the book's website before proceeding to get the most out of this chapter.¹

¹ developer.symbian.com/javameonsymbianos

8.1 What Is a Game?

Games have been around for at least 5000 years in one form or another and there are many opinions and definitions floating around on what constitutes a game.² These tend to focus on challenges, skills, goal-oriented activities, educational stimulation, competition, and so on. These are all valid attributes of all types of game but what we want to think about now is mobile phone games so we will try to sharpen that definition up a little first.

There are obviously many types of game: games of chance such as roulette, turn-based games such as chess, role-playing games (RPGs) such as Dungeons and Dragons, and real-time strategy (RTS) games where players compete against each other or against the computer at a strategic rather than a tactical level (*Age of Empires* is a great example). The 1980s spawned the classic 2D arcade-style games *Space Invaders*, *PacMan*, *Galaga*, *Phoenix* and, of course, *Atic-Atac*³ amongst many others. There are first-person shooter (FPS) games such as *Quake* and *Doom*, as well as massively multiplayer, online, role-playing games (MMORPGs) such as *World of Warcraft*. With fast and relatively cheap broadband Internet access, each of these can take on an online flavor and these days it is nothing to play against opponents from all over the globe.

I hope you like acronyms because the world of game development has many of them. With the exception of a few enthusiastic hobbyists, most people write games for mobile phones with the idea of turning a profit. So in the interests of commercial reality and simplicity, let's be just a little cynical and adopt an operational definition we can work with as developers:

A mobile phone game is any piece of interactive mobile phone content that is entertaining, easily built and that people will buy.

So we can immediately eliminate ringtones, wallpapers and screen savers since they aren't interactive. Strictly speaking, it also eliminates business applications on phones since almost none are all that entertaining.

8.1.1 The Game Loop

Now that we know what a game is (and isn't), we need to look at how games are structured. With the exception of turn-based games (which tend to be event-driven), the heart of a game is the 'game loop'. At

² en.wikipedia.org/wiki/Game

³ en.wikipedia.org/wiki/Atic-Atac

its most abstract level, a game loop can be thought of as a continuous process of checking for user input, updating the game world and then repainting the screen to display the new state of the world to the user.

Unlike mainstream game platforms, mobile phone games must execute on a large and diverse range of target hardware. So in reality there are a couple more steps that need to occur to keep things running smoothly. Not all books include these steps, and some change the order or absorb some steps into others. Figure 8.1 shows the main steps in the game loop as described in [Stichbury 2008].

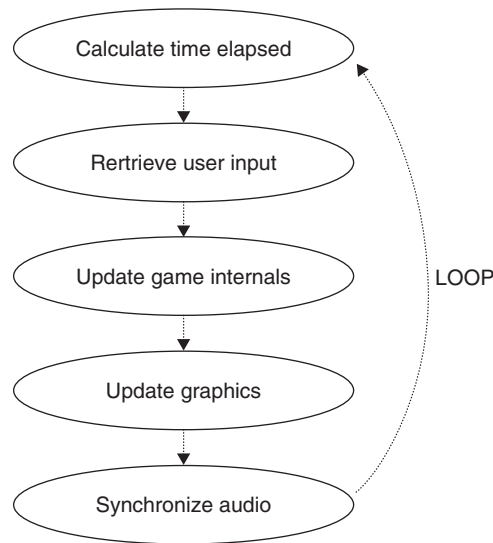


Figure 8.1 The Game Loop from [Stichbury 2008]

In this chapter, we vary this very slightly. Notice that when laid out step by step the process of building a game loses some of its mystery and actually becomes quite mechanical.

In pseudo-code, our generic game loop is:

```
while (game not stopped)
  check if game over
  retrieve and handle user input
  update game world internals (physics/AI)
  update graphics
  synchronize events before next loop
```

Jumping ahead a bit, the implementation then becomes pretty straightforward – you simply create a game thread and these steps translate directly into a series of method calls in the `run()` method of the `Runnable` object.

The astute reader may ask why the game loop checks the ‘game over’ condition as the first step, rather than the last as is normal with a `while` loop. You don’t have to do this – this is simply in the nature of a tip which gives you some added flexibility later on. Many games keep their rendering loops, music and world integrations running even when the game is over. This gives you the chance to show some game stats, upload results to a remote server, run some specific local tasks, and so on. This is very hard to add in after everything else is written, so decide early on if you want to do this.

TIP: Keep the ‘game over’ condition and the ‘game loop’ termination condition separate.

8.1.2 Terminology

Before we go much further there are some terms that need to be defined (see Table 8.1). If you want more information on any of these terms, it can easily be found on Wikipedia⁴ but this should be enough to get by for now. Don’t worry if they don’t make sense just yet – they will by the end of the chapter.

8.1.3 Mobile Games

Designing games for mobile phones requires consideration of a number of factors not found in other areas of the games industry. By virtue of its design, a mobile phone is a communications device and therefore must give priority to this function. When writing mobile games, we need to realize that at any point game play can be interrupted by an incoming call or SMS, a scheduled alarm, sudden loss of battery power, removal of the battery altogether or the game being moved into the background by the user or the operating system. You should always be asking yourself, ‘what happens if the device fails *now*?’

Moreover the capabilities of mobile phones are limited by a wide range of factors. Power budgets, heat dissipation, net handset weight and case dimensions put limitations on available screen real estate, RAM (and therefore available process heap) and processing power. These are not major considerations when building games for mainstream console game decks. Many games, and 3D games in particular, require extensive use of floating-point arithmetic and it is only recently that we are starting to see mobile devices with dedicated FPUs or GPUs emerging onto the market. To date, it has been far more common to emulate floating-point operations in software, which requires a lot more processing power and adversely impacts performance.

⁴ en.wikipedia.org

Table 8.1 Game Terminology

Term	Definition
Culling	An optimization technique used to avoid processing objects that fail to meet certain criteria; examples include viewport culling, polygon culling, and back face culling
Double buffering	A technique used to avoid flicker: drawing is done to an offscreen buffer which is then copied to the display all at once
Field of view (FOV)	A vertical angle defining the portion of the game world currently viewable by the camera
Frames per second (FPS)	The number of times the game screen is repainted each second; television runs at around 25 FPS; an acceptable minimum for mobile games is about 15 FPS although this depends on the type of game
Floating-point unit (FPU)	Hardware that provides fast implementations of floating-point operations: square root, division, addition, etc.
Frame rate	See frames per second
Graphics processing unit (GPU)	Dedicated graphics hardware that provides fast implementations of common graphics primitives specifically for use in games
Heads-up display (HUD)	The area of a game screen where player information is displayed (score, health, weapons, etc.)
Level of detail (LOD)	The amount of detail in a scene: a complex model shown at a distance from the viewer can sometimes be replaced by a simpler one without obvious loss of quality
Mesh	A description of an object in 3D space. It consists of vertices, faces, surface normals, etc. and is usually produced by a modeling tool such as 3D Studio and imported into the game code base where it can be manipulated and rendered
Refresh rate	The frequency at which the frame buffer is flushed to the display; this is outside the control of the game and is a property of the display hardware measured in Hz
Save point	A point in a game where the complete game state is saved; it can later be loaded and the game resumed
Simulation rate	The number of times per second that the physics model of the game world is updated, measured in Hertz (Hz); it is independent of the frame rate and is based on real time

(continued overleaf)

Table 8.1 *(continued)*

Term	Definition
Sprite	Small pre-rendered images used to represent objects in a game; they can be moved, animated and rotated
Viewport	A 2D area representing the currently visible area of the game world

Mobile phones also tend to offer quite limited multimedia capabilities. Until recently many phones only supported simple tone generation sequences and MIDI playback. This has changed though and these days powerful device hardware is common and easily accessible via APIs for onboard cameras, the microphone, digital audio, the display and video playback, and so on. User input techniques are also a lot more limited on mobile devices than on dedicated games hardware. Phones tend to be limited to either a five-way jog dial using standard Java ME game key mappings or touch screen input with a stylus (as on UIQ phones from Sony Ericsson).

Games for mobile phones need to be designed for the unexpected. They need to be ready to react almost instantaneously to a change in the operating environment. In terms of Java ME, games need to be particularly sensitive to the MIDlet lifecycle (see [de Jode 2004]). Research from Nokia [2006b] has shown that many mobile games are played in short bursts by casual gamers. Mobile games are most often played while waiting for a bus, on a queue, or in a lecture or business meeting.

Given this, they need to be designed with a completely different execution model in mind. They need to start quickly, be easy to play and quick to engage. Unlike mainstream games, we need to avoid long title sequences with intro movies or background stories during start up. They need to be easily stopped, re-started, paused and resumed, and support multiple short bursts of play. A great technique for aiding this is to use automatic save points throughout the game. That way if game play is unexpectedly terminated, the player can re-start at some well-defined point.

Mobile games need to be able to quickly adapt to a user's changing operating environment on demand. A well-designed mobile game should allow the user to disable all multimedia effects, including sound, audio or video sequence playback, vibration, and camera usage, at any time, even during game play, so the ability to change game settings should always be available to the player. The best way to do this is by careful design of the game pause menu. For an excellent in-depth treatment of this topic please refer to the article⁵ from Forum Nokia. Basically, you

⁵ See [Nokia 2006a]

should design your game so that settings can be changed when it is in its paused state.

TIP: Allow the user access to the game settings when the game is paused.

Research has shown (and it's common sense anyway) that one of the key indicators of customer satisfaction is the start-up time of a mobile game. One of the best ways to reduce the *perceived* start-up time is the use of a splash screen. The basic idea is to show something to the user as soon as possible after they have started the application. Usually this is a screen grab from the game itself, although in the mobile games world it is often a high-resolution image created by a third-party graphic artist that bears a somewhat dubious relationship to the actual game quality. It can be an animation, a short movie video or title music – whatever is appropriate. However it is done, the point is to show something that catches the eye and entertains as soon as possible while performing loading tasks in the background on a separate thread. These tasks can do anything – commonly they are used to load meshes from the local store, pre-fetch audio or video streams, download remote assets over a socket, or pre-load and cache level data. Whatever the tasks are, the start-up time of the game needs to be as fast as possible so feel free to cheat in any way you can to achieve this. As a rule of thumb, game start-up times should be no more than about three seconds.

TIP: Use a splash screen and background threads to reduce perceived start-up time.

Once a game is up and running, it is the frame rate that determines perceived performance to the end user. Generally speaking, we need to aim for a frame rate between 15 and 25 FPS. Any more than 25 is probably a case of diminishing returns and any less than 15 makes objects appear to move in discrete jumps rather than smoothly. There are a variety of techniques for maintaining a constant frame rate and we discuss them in Section 8.4.1.

It is also essential to keep the rate at which you update the objects in your world (the simulation rate) separate from the frame rate. The simulation rate is usually dictated by the game physics engine. One of the most common mistakes made by beginners in mobile game development is to simply update the physics model once every game loop iteration. This means that on fast hardware, integration of the physics model occurs faster so objects move faster. It also means that as the frame rate varies

so does the speed of world objects. We show how to dissociate the simulation rate and the frame rate when we review our demo game in Section 8.4.1.

TIP: Keep the simulation rate and frame rate independent of the game-loop frequency.

8.1.4 Mobile Graphics

Games are essentially about polygons and pixels. In many 3D games, objects are represented by a mesh. This is usually a set of vertices (points) in 3D space which define a set of adjacent polygons (usually triangles) describing the surface of an object.

It should be no surprise that an object made up of a lot of triangles is more expensive to render than one made up of relatively few triangles. This has been the case for years and high-end graphics cards usually address this using a variety of techniques including caching geometry directly in graphics memory on PC video cards to avoid unnecessary traffic over the bus.

In addition, performance is affected by the actual number of pixels that need to be updated between one frame and the next. The amount of pixels an object needs to occupy in screen space is a direct function of the size of the polygon, its distance from the observer and the angle it makes to the line of sight. This is why polygon-fill rates are key performance indicators of graphics cards for PCs. On mobile phones, processors are very limited and can easily form a performance bottleneck for a game. Consequently, it is quite possible for a complex object made up of many small polygons to render faster than an object built with a smaller number of large polygons, because it requires fewer pixels. This directly affects the frame rate and game performance.

TIP: On mobile phones, the number of pixels to be updated has a large effect on resultant frame rate due to slower processors.

Lastly, a technique used in mainstream game development is the concept of varying levels of detail (LOD). The idea leverages the fact that an object in the distance is not as clear and sharp as one that is closer to us. Imagine we want to render a tank that moves around in a 3D world. We could have three models for the tank – one to use when it is off in the distance, which would consist of the fewest triangles, a model of intermediate complexity for intermediate distances, and a high-quality (or high triangle count) mesh when the tank is close to the point of view.

These concepts were all born in the PC game development industry over the last few decades and are still relevant to mobile games today. The core theme is always to get away with as much as you can without decreasing the user experience.

TIP: Do not draw what cannot be seen and cheat as much as you can!

The world of mobile game development is specialized and extremely interesting. If you want to know more about general games concepts, please refer to three excellent books: [Stichbury 2008] is dedicated purely to games development on Symbian OS and is an excellent read. [Astle and Durnil 2004] and [Malizia 2006] focus on 3D graphics while exploring a wide range of topics in mobile games development and are highly recommended if you are serious about learning more.

8.2 Building a Simple MIDP Game

Now that's all great, but let's face facts – writing and playing games are the fun bits. However before we get into the advanced stuff let's pretend that all we have is Java ME without the frills and see what we can do. The point to be demonstrated here is that the MIDP Game API was developed for the feature phone market. In this market, there are limits on heap size, thread pools, JAR size, few or no optional JSRs and minimal multimedia capabilities. Given all of that it is amazing what can be done in a very short space of time. So let's hook straight in with a simple game that we can throw away just as a quick demonstration of how much the MIDP 2.0 Game API does for you.

Figure 8.2 shows a screen shot for a demo game, called **GhostPong** in honor of the original arcade game Pong.⁶ It's less than 150 lines in length and took just over an hour to write (it should have taken less but I kept making stupid typing mistakes...).

The point of the game is that basically the little ghost sprites fall from the top of the screen and you have to smash them out of existence with the brick paddle at the bottom which moves left and right. You get 10 points for each one you get and you lose 10 points for each ghost you miss. And as for the name – well there's no excuse for that.

What this shows is that you don't need complex architectures and reusable libraries in order to build effective games. With MIDP 2.0 and the Game API you can jump straight in and just get on with it. The game is only 6 KB in size and has no external requirements other than a dependency on MIDP 2.0. This means that it would quite possibly run today on almost a billion mobile phones.

⁶ en.wikipedia.org/wiki/Pong


```

26     pong = new Sprite(Image.createImage("/pong.png"));
27     pongWidth = pong.getWidth();
28     pongHeight = pong.getHeight();
29     pongX = (getWidth() - pongWidth) >> 1;
30     pongY = getHeight() - 20 - pongHeight;
31     pong.setPosition(pongX, pongY);
32     tonePlayer = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
33     tonePlayer.realize();
34     toneControl = (ToneControl) tonePlayer.getControl("ToneControl");
35     toneControl.setSequence(soundSeq);
36     layerManager = new LayerManager();
37     layerManager.append(pong);
38 }
39
40 public void start(){
41     Thread thread = new Thread(this);
42     thread.start();
43     running = true;
44     startTime = System.currentTimeMillis();
45     lastGenTime = startTime;
46 }
47
48 public void run(){ // the game loop
49     while(running){
50         try{
51             generateGhosts();
52             movePong();
53             moveGhosts();
54             draw();
55             flushGraphics();
56             Thread.sleep(50);
57         }
58         catch(Exception e){}
59     }
60 }
61
62 private void draw(){
63     graphics.setColor(0x000000);
64     graphics.fillRect(0, 0, getWidth(), getHeight());
65     graphics.setColor(0xff0000);
66     graphics.setFont(font);
67     graphics.drawString("'" + points, getWidth()>>1, 10,
68                                     Graphics.HCENTER | Graphics.TOP);
69     layerManager.paint(graphics, 0, 0);
70 }
71 // every second there's an 80% chance of generating new ghosts
72 private void generateGhosts(){
73     long elapsed = System.currentTimeMillis() - lastGenTime;
74     if(elapsed > 1000L){
75         int i = genRandom(1, 10);
76         if(i <= 8){
77             int numGhosts = genRandom(1, 4);
78             for(int g = 0; g < numGhosts; g++){
79                 Sprite ghost = new Sprite(ghostImage);
80                 int ghostX = genRandom(0, getWidth() - ghost.getWidth());
81                 ghost.setPosition(ghostX, 0);

```

```

82         if(i < 4) // occasionally flip the sprite for variety
83             ghost.setTransform(Sprite.TRANS_MIRROR);
84         layerManager.append(ghost);
85     }
86 }
87     lastGenTime = System.currentTimeMillis();
88 }
89 }
90
91 private void moveGhosts(){
92     int fallSpeed = 10;
93     int numGhosts = layerManager.getSize() - 1; // exclude the pong
94     for(int i = numGhosts; i >= 1; i--){
95         Sprite ghost = (Sprite) layerManager.getLayerAt(i);
96         ghost.move(0, fallSpeed);
97         if(ghost.collidesWith(pong, true)){
98             points += 10;
99             layerManager.remove(ghost);
100             try {
101                 tonePlayer.start();
102             }
103             catch(Exception e){
104                 e.printStackTrace();
105             }
106         }
107         else if(ghost.getY() > this.getHeight()){
108             points -= 10;
109             layerManager.remove(ghost);
110         }
111     }
112 }
113
114 private void movePong(){
115     int keyState = getKeyStates();
116     int dx = 15;
117     if( (keyState & GameCanvas.LEFT_PRESSED) != 0){
118         pongX -= dx;
119         if(pongX < 0) pongX = 0;
120     }
121     else if( (keyState & GameCanvas.RIGHT_PRESSED) != 0){
122         pongX += dx;
123         if(pongX > (getWidth() - pongWidth))
124             pongX = getWidth() - pongWidth;
125     }
126     pong.setPosition(pongX, pongY);
127 }
128
129 // shut down on right softkey (S60 & UIQ)
130 public void keyPressed(int keyCode){
131     if(keyCode == -7 || keyCode == -20){
132         midlet.die();
133     }
134 }
135
136 public int genRandom(int min, int max){
137     return (Math.abs(random.nextInt()) % max) + min;
138 }
139 }

```

Now we are going to have a quick look at what we have managed to cover in under 150 lines of Java ME code.

8.2.1 What Has It Got?

No argument – this game won't be on any best-seller list but let's see some of the things this simple game demonstrates:

- Loading and displaying an image from a JAR file (lines 25 and 26)
- Creating a Sprite instance from an image file (line 26)
- A basic game loop using a thread (lines 48–60)
- Using a full screen GameCanvas (lines 20 and 21)
- Examining key states to respond to user input (lines 115, 117 and 121)
- Moving and rotating Sprites (lines 79–83)
- A simple way to generate random numbers over a fixed interval (lines 136–138)
- Basic layer management using the `LayerManager` class (lines 95, 99–109)
- Basic MIDP 2.0 Media API tone sequences (lines 14–17, 32–35 and 100–105)
- Responding to softkey events (lines 130–134)
- Collision detection between Sprites (line 97).

It has to be said that, once you get used to the Game API, you will see that almost all of these features came for free. The bulk of this game is made up of simple code snippets that glue together pre-packaged MIDP 2.0 game functionality.

8.2.2 What Has It Not?

At first glance it isn't too bad for a first pass. However there are a number of things that this game lacks. Some of the missing features can be classed as 'nice to have' but quite a few are core and represent common errors made in mobile game development.

Obviously it doesn't have menus, a splash screen, score tracking or advanced media. It doesn't use any particular architecture, would be hard to maintain and is a little casual with its memory management. Here are some other problems:

- It does not handle pausing of the game in any way.

- It has no handle to the game thread so this cannot be explicitly managed.
- The flag variable `running` is set *after* the call to `Thread.start()` – so if the game thread starts and is immediately pre-empted by another thread before line 43 executes, the application is left in an inconsistent state.
- Access to the flag variable `running` is not synchronized across the GUI and game loop threads (e.g., by using the Java keyword `volatile` in its declaration).
- It creates a large number of new object instances (Sprites in this case) instead of re-using a pool of them. The amount of heap in use at any time depends completely on how often the garbage collector runs.
- No effort is made to manage the frame rate. The thread simply sleeps for 50 ms each time the game loop executes. Changing the sleep period speeds up or slows down the movement of the ghost sprites.
- There is no way to turn off the tone sequence used for sounds. This is very annoying.
- There is no concept of ‘game over’ – it just runs forever with no challenge.
- It uses magic numbers (line 131) for special key codes instead of defining them as constants. Unfortunately, you also can’t use the `Canvas.getGameAction()` method to handle softkey events because these are not part of the standard MIDP key mappings and are specific to each manufacturer.

8.2.3 Debrief Summary

So far we have discovered two important concepts regarding game development with Java ME:

- It is really easy to write Java games.
- It is really easy to write Java games *badly*.

This is not all that surprising. The vast majority of mobile phones in the mass market at the time MIDP 2.0 was designed had a number of built-in limitations. So MIDlets did not have much memory, little local storage space and were rarely paused. Few mobile-phone operating systems are multitasking and MIDlets that lose focus are often terminated outright. A direct consequence is that there is a large code base of Java ME sample code in existence that takes little or no heed of the pause–resume cycle. It is common to see (as above) a `start()` method on the `GameCanvas`

that is called whenever `startApp()` is called. In almost all cases, this starts a new game thread without checking for an existing game thread or storing a reference to the game thread at class scope so that it can be shut down in a well-defined manner.

TIP: Always keep a class-scoped reference to the game thread to control its behavior.

To help clarify these issues, try the following exercises:

1. Use the Sun WTK (or whatever IDE you prefer) to start the ***GhostPong*** MIDlet and let it run for a few minutes. You can quickly see the degradation in performance as more and more Sprite instances are created. Even though all references to these are removed, there are simply too many allocated too quickly for the garbage collection thread to keep up.
2. Install the MIDlet on a Symbian OS phone (by cable, Bluetooth transfer or PC Suite software). Start the game and then move it into the background. Assuming your phone is not in silent mode, you can hear the game continuing to execute in the background, happily consuming battery power, CPU cycles, RAM and delivering a bizarre user experience. Moving the MIDlet back into the foreground in fact triggers a new game thread instance and very quickly you have a process with a number of unmanaged threads.

TIP: Where possible, try to reuse objects by using a managed pool of instances.

Symbian OS is a fully multitasking operating system which means that Java ME MIDlets need to be designed to acquire and release resources as they are moved in and out of the foreground. For an excellent discussion on this the reader is referred to Section 3.5.1, Section 4.5 and [de Jode 2004b].

8.3 MIDP 2.0 Game API Core Concepts

In Chapter 2, we ran through descriptions of the packages and classes that make up the Game API. The aim of this section is to detail a number of features it provides, to show how it aids game development on mobile phones. The MIDP 2.0 Game API supplies a rich game development framework. A lot of the functionality was incorporated into the MIDP 2.0 specification as a result of real-world experiences writing early Java ME

games. As you read through this section, spare a thought for those early pioneers who had to build all of this from scratch.

8.3.1 The Full-Screen Canvas

The game canvas supplied as part of MIDP 2.0 allows the game to take over the full screen real estate – or at least as much as the implementation allows (UIQ devices do not allow *complete* use but they tend to have larger screens in any case). Prior to this, developers were stuck with even smaller areas than are around today, unless they leveraged custom libraries provided by the phone manufacturers. One example of this was the Nokia UI which provided a full-screen canvas.

8.3.2 Game Key Mappings

These are an abstraction of the basic actions taken by players during a game. In most cases, these actions are moving left or right, up or down and triggering some action such as firing a weapon or jumping. This allows Java ME game code to work over a variety of hardware input methods as each manufacturer can map, for example, the UP action to an input method best suited to the device.

Game actions are defined as fields of the Canvas class. The code snippet below demonstrates how to extract the game action in an event handler:

```
public void keyPressed(int keyCode){
    int gameAction = getGameAction(keyCode);
    switch(gameAction){
        case Canvas.FIRE: ...
        case Canvas.UP: ...
```

8.3.3 Painting and Double Buffering

The GameCavas provides double buffering by default. The painting model is such that all drawing occurs to an offscreen buffer which is then sent to the display using the `flushGraphics()` method. As a developer, your task is to provide an implementation of the `paint()` method. It is called by the framework: you should never call this method yourself. Application-initiated painting is done via the `repaint()` and `serviceRepaints()` methods. Calling `repaint()` only issues a request to update the screen – it completes immediately and the framework calls your `paint()` method asynchronously. In this manner, a number of calls to `repaint()` may actually result in only a single update. The `serviceRepaints()` method is synchronous and forces any pending

repaints to be completed immediately. For a more in-depth discussion of efficient use of graphics on Symbian OS please refer to Section 9.2.

8.3.4 Key-Stroke Detection

Games often need to detect fast key strokes in the game loop. In order to make sure that no key strokes go undetected, the `GameCanvas.getKeyStates()` method is used. This returns an integer representing the states of the keys on the phone, where each bit represents a specific key. The bit is 1 if the key is currently down or has been pressed at least once since the last time the method was called. It is 0 if the key is currently up and has not been pressed at all since the method was last called. The following code snippet demonstrates the use of this:

```
private void processUserInput(){
    if(!gameOver){
        int keys = getKeyStates();
        if((keys & GameCanvas.FIRE_PRESSED) != 0){
            gameEngine.startFiring();
            ...
        }
        if((keys & GameCanvas.UP_PRESSED) != 0){
            gameEngine.upAction();
            ...
        }
    }
}
```

In addition, the constructor of the `GameCanvas` class takes a Boolean value indicating whether to suppress key events. When true, the framework does not call `keyPressed`, `keyRepeated` and `keyReleased` event handlers, which can reduce unnecessary overhead. Two points to remember here is that it only applies to game actions and is appropriate only if you want to use the `getKeyStates()` method for input detection.

8.3.5 Sprites, Layers and Collision Detection

The game API has excellent support for sprites. A `Sprite` can be moved, positioned, flipped, and mirrored around both the horizontal and vertical axes or about an arbitrary point by using the `defineReferencePixel()` method. A sprite can also be constructed from a single image consisting of a set of equally sized frames which can then be animated using the `nextFrame()` and `prevFrame()` methods.

You can also use the `collidesWith()` and `defineCollisionRectangle()` methods for collision detection between sprites and other layers in the game world at either a bounding box or pixel level. Pixel-level collision tests only check for collisions between opaque pixels, which gives the best visual results but requires more processing.

Chapter 2 gives an excellent example of the use of the `TiledLayer` class specifically designed to enable the creation of large scrolling backgrounds from a relatively small set of images known as tiles. This class also supports the concept of the animated tile which is very useful for creating interesting and active game backgrounds quite cheaply.

Finally while not part of the game API, the MIDP 2.0 specification made support for tone generation mandatory as part of the Media API which is a subset of MMAPI. This meant that there was always some way of creating basic sounds in games. When you put all of this together the MIDP 2.0 Game API is without peer in the mobile space.

For a more detailed example of how to build simple MIDP-based games using many of the other techniques outlined above, please refer to [Mason 2008].

8.4 Building an Advanced Java Game on Symbian OS

In Section 8.2.3, we outlined some of the constraints that devices and operating systems in the feature phone market impose on Java ME. Most of the limitations you encounter on other platforms do not apply when building MIDP games on Symbian OS. Within the bounds of available resources, Symbian OS does not place any limits on JAR size, the number of sockets you can open, or the number of threads you can have; the local storage space and the size of the heap can grow and shrink dynamically as required.

This means that you can do some things in a Symbian OS MIDlet that you would not even consider on other platforms. For example, you can load a large number of high-resolution images at once, play advanced sound formats instead of MIDI sequences, and distribute detailed models or level data as JAR file resources. That is not to say that you should be lax with your memory management but it need not be your first concern (see Section 3.4).

More importantly, as we see in this section, you can adopt a more abstract framework for your code base which translates directly into a high degree of re-usability. There is no need to use the extreme Java ME optimization techniques that are needed on low-end platforms (such as, avoiding abstract classes and interfaces to keep JAR sizes down; see Section 9.4.2).

For the rest of this chapter, we work through the main features of a sample game called ***Finding Higgs*** in honor of recent work done (well, planned anyway, before it shut down for repairs in its first week) at the Large Hadron Collider in CERN.⁷ The aim is to review a first-person-shooter game that demonstrates the use of various JSRs commonly used

⁷ 'God Particle Discovery Likely', news.bbc.co.uk/2/hi/uk_news/scotland/edinburgh_and_east/7608342.stm

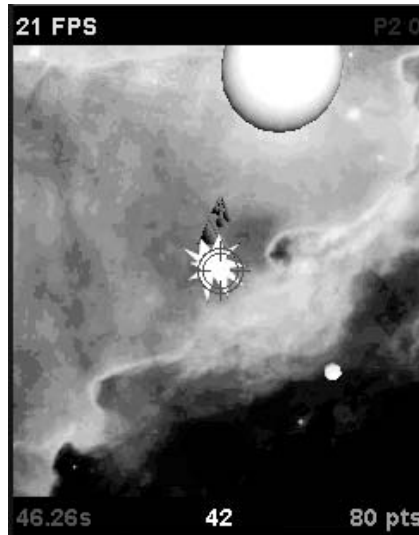


Figure 8.3 Screenshot from *Finding Higgs*

in game development and that are provided with Symbian OS. Figure 8.3 shows a screen shot of the game in action.

8.4.1 Planning

Having clear answers to the following questions before development begins directly reduces risk and development time for the project. It also helps simplify your game design from the outset:

- 1. What are the aims of the player?**

The player has one minute to destroy as many attackers as possible while using the least amount of ammunition. Bonus points are awarded if the player's hit rate exceeds 60%. In single-player mode, the aim is to beat the previous high score. In two-player, head-to-head mode the aim is to beat your opponent.

- 2. What are the 'game over' conditions?**

The game completes after one minute has elapsed.

- 3. What controls are available for player input?**

The player can use the five-way navigation joystick or the keys marked 2, 4, 5, 6 and 8 on the keypad for navigation. In this game, the FIRE action fires the gun and the up, down, left and right keys move the player in a polar orbit around the world origin in 3D space.

4. How are scores and game status displayed?

As you can see in Figure 8.3, the top left corner shows the current frame rate at all times. The top right shows the opponent's score during multi-player mode. This is updated via Bluetooth in real time. The bottom left shows the remaining game time; the bottom centre shows the number of shots fired; and the bottom right shows the player's current score.

5. How is the pause–resume cycle handled?

The MIDlet class monitors its own state, keeping itself in sync with notifications from the AMS at all times. The `Controller` class manages the application lifecycle and the `GameController` manages the game lifecycle. A separate pause menu is used to allow the player to quit the current game or change the settings at any time.

6. How is the game world represented?

The game world is 3D space where a series of attacks are launched along the z-axis towards the player. The Mobile 3D Graphics API (JSR-184) is used to represent this world.

7. How are frame and simulation rates managed?

The frame rate is capped at 25 FPS by granting up to 40 ms for each frame. This works by tracking the amount of time it takes to complete the rendering phase of the game loop. If this is found to be less than 40 ms, the thread sleeps for a short period in order to re-synchronize itself. This approach enables other active threads to use processor cycles for other useful tasks and allows CPU power management which can help increase battery life.

Be careful with this – on Symbian OS, the native timer has a resolution of 1/64 seconds so what you request may not be what you actually get. Using a periodic timer in this manner is susceptible to drift because you actually only receive an event once about every 30 ms – so your actual frame rate is slightly less than expected.

The simulation rate runs at a different rate than the frame rate by tracking the time since the last simulation step and only performing a new one if a sufficient time period has elapsed.

8. Does the game respond to a change in display size?

In this case, the game is not designed to respond to changes in screen size.

9. What multimedia effects are employed?

A WAV file is used for the sound effect of the weapon and a set of MIDI sequences are used to provide background music. At the start of each music sequence, a vibration event is triggered.

10. How is navigation between screens in the application managed?

In this case, the entire application navigation is abstracted over three packages with the main management being the responsibility of the `Navigator` class. The point is to provide both a custom-drawn menu system using only the low-level graphics routines in MIDP 2.0 and a more advanced menu system using the Scalable 2D Vector Graphics API (JSR-226) where supported.

8.4.2 Design

The game design consists of the logical subsystems shown in Figure 8.4.

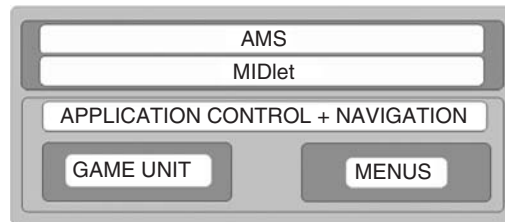


Figure 8.4 High-level game design

The AMS and the MIDlet together represent the interface between the application and the outside world, i.e. the operating system. The application itself is designed on a Model–View–Controller (MVC) pattern, where the `Controller` class manages application view changes and navigation using the active menu system. Its primary role is to respond to lifecycle events from the operating system and to drive the application framework. To a large extent, the controller is not aware that is a ‘game’ so to speak. It owns a `GameController` instance that it notifies when events occur. Other than that, the controller remains independent of the particular application.

Figure 8.5 shows the application’s main classes and their relationships. As you can see from the names, most of the classes are designed for re-use and have little to do with the particular game being built.

8.4.3 Menus and Navigation

This game uses a relatively heavy UI layer in the form of three packages. This is not necessarily recommended practice – it has been done this way to demonstrate how you can support two different menu system implementations and select the one to use at run time based on the target device (see Section 4.4). Both the SVG and LCDUI subpackages

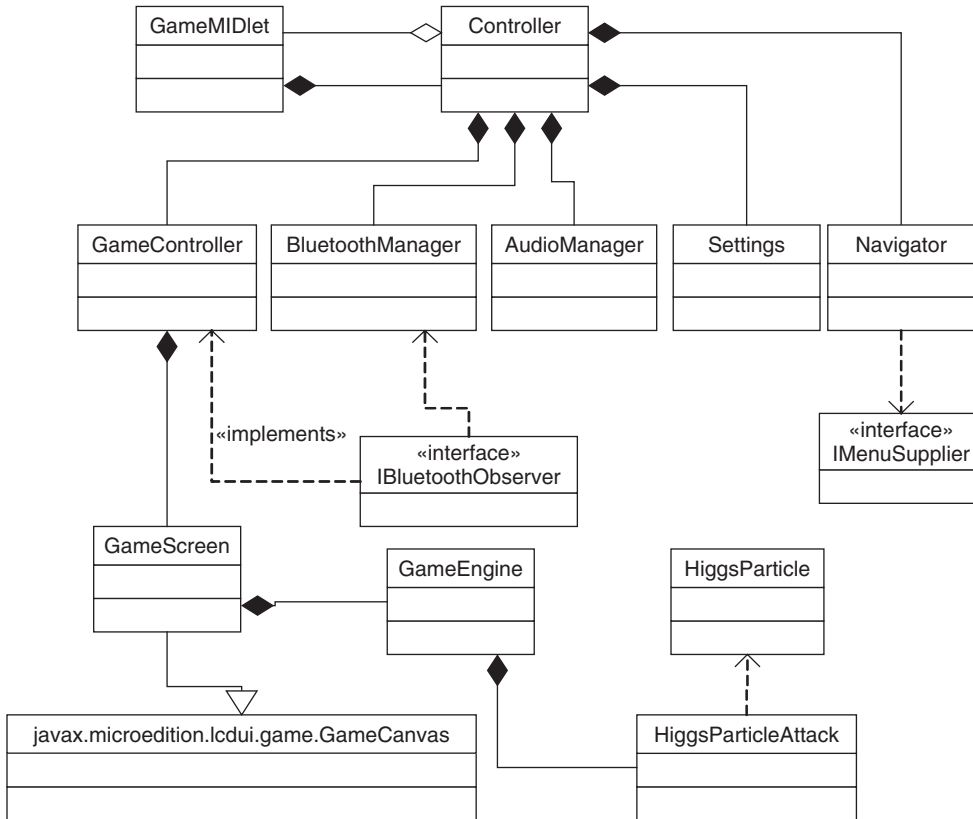


Figure 8.5 Main application classes

in the source code for this chapter provide implementations of the `IMenuSupplier` interface to achieve this:

```

public interface IMenuSupplier {
    public void showMainMenu();
    public void showPauseMenu();
    public void showSettings();
    public void showMultiPlay();
    public IMessageScreen showMessageScreen(String message);
}

```

Application navigation is performed by the `Navigator` class which delegates menu requests to the appropriate implementation. In this way the rest of the application is completely isolated from the menu system in use. Figure 8.6 shows the main game menu with the SVG version on the left and the low-resolution graphics version on the right.

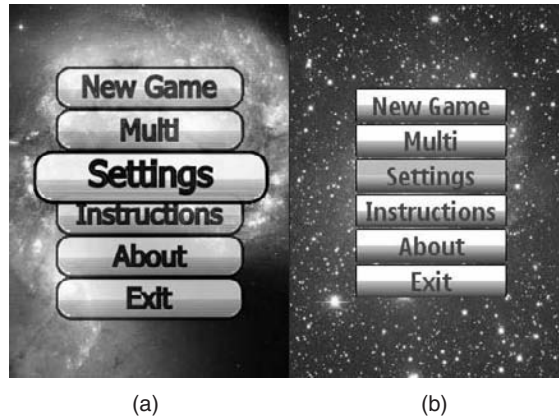


Figure 8.6 Dual menu systems: a) SVG and b) low-resolution graphics

Incidentally, you should be able to see that the menus overlay different background images. Every time a new game is started, the game randomly selects a new background from a set of 10 high-resolution images included in the application JAR file. That becomes the active background for the entire MIDlet until a new game is played.

There is no particular point to this other than that it varies the feel of the game increasing its interest level and the pictures look great (they're all from NASA). This also shows just one of the things you can do on Symbian OS when you do not have to operate under the stringent limitations imposed by other operating systems. You can use your imagination to come up with an almost endless variety of great effects and features.

8.4.4 Using the Mobile Media API (JSR-135)

Chapter 2 discusses the lifecycle of a `Player` and demonstrates how to instantiate, realize, set starting volume and pre-fetch to minimize delay when the `start()` method of the player is called, so we won't repeat that information here. What we are going to show is how to provide background music without it running all the time. Many games do this simply because studies have shown that music helps create a more immersive experience. By interspersing MIDI sequences with periods of silence we can change the 'feel' of the game in a subtle manner which helps maintain player interest.

To do this we use timers, some basic parameters and a random-number generator. We also need to detect when the active sequence has finished by implementing the `PlayerListener` interface. Whenever the current background music stops we wait for a random (bounded) interval of time and then start the next sequence:

```
// from PlayerListener
public void playerUpdate(Player player, String event, Object eventData){
    try{
        if(event.equals(PlayerListener.END_OF_MEDIA)){
            if(player == backgroundPlayer1){
                activeBackgroundPlayer = backgroundPlayer2;
            }
            else if(player == backgroundPlayer2){
                activeBackgroundPlayer = backgroundPlayer3;
            }
            else if(player == backgroundPlayer3){
                activeBackgroundPlayer = backgroundPlayer1;
            }
            startBackgroundFXTimer();
        }
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

private void startBackgroundFXTimer(){
    stopBackgroundFXTimer();
    TimerTask task = new TimerTask(){
        public void run(){
            playBackgroundMusic();
        }
    };
    backgroundTimer = new Timer();
    backgroundTimer.schedule(task, musicDelay);
    musicDelay = Utilities.random(BACKGROUND_WAIT_MIN, BACKGROUND_WAIT_MAX);
}
```

8.4.5 Using the Scalable 2D Vector Graphics API (JSR-226)

As explained earlier, this game uses a menu system based on the Scalable Vector Graphics API, when it is detected on the device. Figure 8.6a shows the results of a very simple linear menu using scaling and transparency. The best way to get started using JSR-226 is to read [Nokia 2006e] and [Powers 2005].

An SVG image is simply an XML document containing a set of drawing instructions which allows an image to be created using high-level instructions. You can say, for example, ‘draw a circle radius 5 at point 2,10’. The main benefits you get by using SVG images instead of traditional raster images is their scalability, ease of manipulation and the fact that they compress extremely well in a JAR since they are just a text document.

The full SVG specification defines support for a wide range of features but not all of these are yet supported on mobile phones. Instead phones support a subset of the specification called SVG-Tiny also referred to as SVG-T. The current version of SVG-T is 1.1 and this is the version currently shipped on Symbian OS 9.x devices. To determine from Java ME

what version of SVG-T your phone supports, you can query the system property as follows:

```
String svgVersion = System.getProperty("microedition.m2g.version");  
boolean useSVG = svgVersion.equals("1.1");
```

In order to read the XML, the SVG-T specification defines a micro-DOM that the implementation of the JSR must provide. This is a small XML parser specifically designed to parse SVG elements. Note that as part of the SVG-T implementation, it should not be considered an XML parser for general use.

There are two ways to create SVG content: SVG images can be created programmatically through a series of API calls but it is far more common to load and manipulate SVG images prepared using a third-party graphics tool such as Adobe Illustrator. If you don't have access to this, a common open-source, free alternative for SVG content creation is Inkscape.⁸ All of the SVG images used for the menu system in this chapter were created using Inkscape.

Before you attempt to load or display an SVG image from a MIDlet, you need to convert it to SVG-T format. The S60 3rd Edition SDKs come with an SVG to SVG-T conversion tool which you can use for this. The installer can be found in the `\S60Tools\` subdirectory of your SDK root. There are some incompatibilities with support for the 'text' element on the Symbian OS implementation at this stage so it is best to convert any text to a path (a series of line segments and arcs) in Illustrator or Inkscape (see Figure 8.7) before converting it to SVG-T. The resulting SVG file will be quite a bit bigger, depending on how much text it contains but for simple menus like ours this is not an issue – remember that we don't need to care too much about JAR size on Symbian OS.

Rendering to screen is done with an instance of the `ScalableGraphics` class. Within our `paint()` method, we simply bind our `Graphics` context to our scalable context, directly render SVG images to our canvas and then release our context. By obtaining a handle to the root element of an `SVGImage` instance we can also directly scale, rotate and translate the image itself as well as change its properties such as its CSS style attribute which defines fills, strokes, color, and so on.

The following code snippets are taken from the `BasicSVGMenuScreen` class which forms the base class for all the SVG menus in the game. It is surprising how easy it is to get good effects with very little code.

⁸ www.inkscape.org

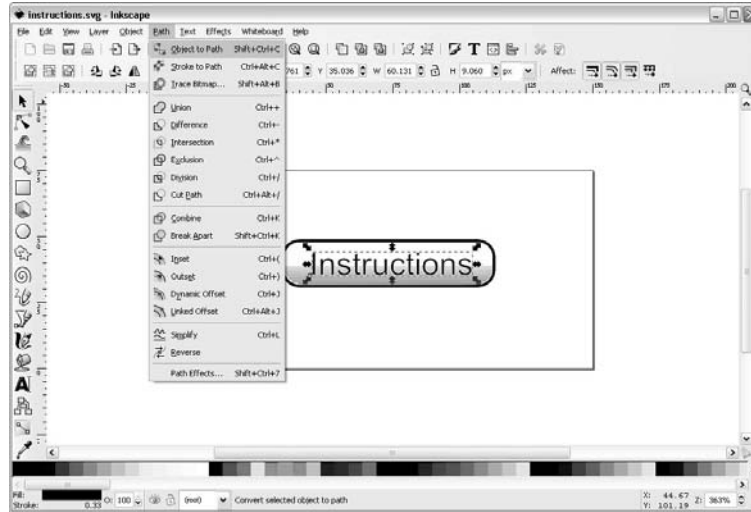


Figure 8.7 Converting text objects to paths with Inkscape

```
public abstract class BasicSVGMenuScreen extends BasicScreen {
    protected ScalableGraphics sg;
    protected static final float BASE_SCALE          = 2.0f;
    protected static final float ACTIVE_SCALE        = 2.5f;

    protected static final float BASE_TRANSPARENCY   = 0.70f;
    protected static final float ACTIVE_TRANSPARENCY = 0.90f;
    ...
    // constructor
    ...
    sg = ScalableGraphics.createInstance();
    sg.setRenderingQuality(ScalableGraphics.RENDERING_QUALITY_HIGH);
    sg.setTransparency(BASE_TRANSPARENCY);
    ...
    public void paint(Graphics g){
        Graphics context = g;
        sg.bindTarget(context);
        sg.setTransparency(BASE_TRANSPARENCY); // more transparent
        ...
        // now render the active item
        sg.setTransparency(ACTIVE_TRANSPARENCY); // more opaque
        sg.render(x, activeY, menuOptions[selectedIndex].getImage());
        sg.releaseTarget();
        ...
    }

    // load SVG content from the JAR file
    protected void setImage(String imageName, boolean active){
        InputStream svgStream = null;
        try{
            svgStream = getResourceAsStream(imageName);
            image = (SVGImage) (SVGImage.createImage(svgStream, null));
        }
    }
}
```

```

        image.setViewportWidth(getWidth());
        image.setViewportHeight(getHeight());
        setActive(active);
    }
    catch(Exception e){
        e.printStackTrace();
    }
    finally{
        if(svgStream != null){
            try{
                svgStream.close();
            }
            catch(IOException ioe){ // ignore
            }
        }
    }
}

public void setActive(boolean active){
    SVGSVGElement svgDoc = (SVGSVGElement)
        image.getDocument().getDocumentElement();

    if(active)
        svgDoc.setCurrentScale(ACTIVE_SCALE); // bigger
    else
        svgDoc.setCurrentScale(BASE_SCALE); // normal
}

```

8.4.6 Using the Mobile 3D Graphics API (JSR-184)

Entire books have been written on 3D graphics and there is very little that can be covered here in a few pages. Before starting, it would be a good idea to run through some basic online tutorials⁹ or even get hold of [Malizia 2006]. We assume that the reader is familiar with 3D concepts such as meshes, transforms, cameras, vector mathematics, and so on, and we do not cover them in detail here.

JSR-184, also referred to as M3G, allows developers to load and animate 3D assets with very few lines of code. Unlike many other 3D graphics libraries (such as OpenGL and DirectX), M3G supports two programming models which can be freely mixed together. In *retained* mode, you usually require fewer lines of code to get your content on-screen, but you do so at the cost of low-level control. Retained mode operates on the concept of a scene graph where the world is represented by a tree of nodes with the `World` node at the root. This includes meshes, lights, cameras, 3D sprites, and so on. You can also group subsets of nodes together and operate on the group as a whole.

Immediate mode is more like OpenGL where you send commands directly to the graphics hardware for execution. You have much more

⁹ Sony Ericsson has a Mobile 3D Tips, Tricks and Code Archive at developer.sonyericsson.com/site/global/techsupport/tipstrickscode/mobilejava3d/p.mobilejava3d_tips.new.jsp. The Sun Developer Network has a 'getting started' article on developers.sun.com/mobility/apis/articles/3dgraphics.

fine-grained control in immediate mode. These modes are logical ones not actual – you don't specify a mode to code in, rather it is implicit from the way you structure your program.

M3G is not a replacement for non-Java mobile graphics technologies such as OpenGL ES and Direct3D Mobile but rather should be regarded as a complementary technology. For example, on Symbian OS, the JSR-184 implementation uses the OpenGL ES libraries directly and therefore M3G automatically benefits from any hardware acceleration chip present on the device.

You can create 3D content programmatically using (a lot) of code but that is only useful for small applications and demos. To use rich content, developers usually import the 3D world from a special M3G file. This is a file format specific to JSR-184 that can hold animations, textures, cameras, lights, materials, meshes, and so on. A Java ME MIDlet can include .m3g files in its JAR or download them from a server.

Commercial modeling tools such as 3D Studio come complete with the option to export a model in M3G format. Unless you are a professional, you probably won't have access to this, however, there is an open-source alternative called **Blender** which you can download for free from www.blender.org (see Figure 8.8).

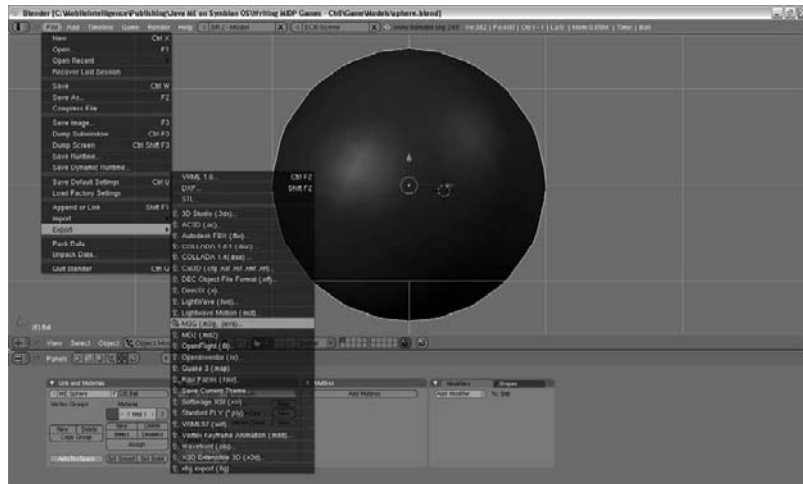


Figure 8.8 Exporting to M3G format with Blender

Blender does not include the option to export models in M3G file format by default, but there is a way around this. **Blender** supports extension via Python scripts and one such well-known script (`m3g_export.py`) can be found online.¹⁰ To get this to work correctly on Windows, you need to

¹⁰ Blender Export for J2ME is available from www.nelson-games.de/bl2m3g/default.html.

install Python (at least v2.5.x) and you need to copy `m3g_export.py` to your blender script directory – normally `C:\Program Files\Blender Foundation\Blender\blender\scripts`.

When that is working you can create entire 3D worlds and easily use them in Java MIDlets by using the static `Loader` class which de-serializes a stream of classes that derive from `Object3D` into a convenient array. In the sample game, we use a sphere mesh to represent distant planets and orbiting objects. Below is the code which shows how the sphere model shown in Figure 8.8 is loaded from the M3G file into the MIDlet proper:

```
Mesh sphere;
private void loadModels(){
    Object3D[] objects = null;
    try{
        objects = Loader.load(... + "sphere.m3g");
        World world = null;
        // find the world node first
        for(int i = 0; i < objects.length; i++){
            if(objects[i] instanceof World){
                world = (World) objects[i];
                break;
            }
        }
        if(world != null){
            for(int i = 0; i < world.getChildCount(); i++){
                Node node = world.getChild(i);
                if(node instanceof Mesh){
                    sphere = (Mesh) node;
                    break;
                }
            }
            if(sphere != null){
                setUpUniverse();
            }
        }
        ...
    }
}
```

Before we can draw anything with M3G we need to set up our graphics environment. Since this game is a first-person-shooter game, we always position the camera at the point of view of the player. In this case, our model only includes the sphere mesh so we have to add our own camera and lights during setup:

```
private void initialiseGraphics(float width, float height){
    // set up a camera
    camera = new Camera();
    // set up a 60 degree FOV
    camera.setPerspective(60.0f, width / height, 0.1f, ...);
    ...
    // set up basic lighting
}
```



```

g3d = Graphics3D.getInstance();
g3d.resetLights();
// add a white ambient light at full intensity
Light light = new Light();
light.setMode(Light.AMBIENT);
light.setColor(0xffffffff);
light.setIntensity(1.0f);
light.setRenderingEnable(true);
g3d.addLight(light, identity);
...

```

In M3G, many of the visual properties that describe a `Mesh` can be specified using the `Appearance` class. This is a container class that holds child objects such as `Material` (how the object responds to different light colors), `PolygonMode` (shading model, winding and culling mode) and `Fog`.

Once you're an experienced 3D modeler you can specify these features during the design phase but for now we define them in code for two reasons: it gives us the flexibility to experiment easily and it allows us to demonstrate a very useful technique. A mesh is expensive to create and takes up a fair bit of memory when you consider all the vertices, texture coordinates, material data, and so on. So in a mobile game we want to re-use the same mesh as much as possible.

In our sample game, the `HiggsParticle` class is a subclass of the `Mesh` class. When a new attack is launched at the player consisting of, say, 10 particles, we do not want 10 copies of the same mesh in memory. What we can do instead is to use a single mesh and change the material properties of its `Appearance` object. If we do this, we can simply render the same mesh multiple times by specifying different transforms and materials.

In Figure 8.3, you can see a large blue-green planet off in the distance and a small red-orange orbiting planetoid at the lower right. The following code snippet shows how, during game setup, we create materials and transforms for each of these while re-using the same mesh:

```

// setup an Appearance object and Materials
private void setUpUniverse(){
    appearance = sphere.getAppearance(0);
    polygonMode = appearance.getPolygonMode();
    polygonMode.setCulling(PolygonMode.CULL_BACK);
    polygonMode.setShading(PolygonMode.SHADE_SMOOTH);
    planetoidMaterial = new Material();
    planetoidMaterial.setColor(Material.AMBIENT, 0x00ff8000);
    planetoidMaterial.setColor(Material.DIFFUSE, 0xFF876A56);
    planetoidMaterial.setColor(Material.SPECULAR, 0x00C0C0C0);
    planetoidMaterial.setShininess(35.0f);
    blueGreenMaterial = new Material();
    blueGreenMaterial.setColor(Material.AMBIENT, 0x00000ff8);
    blueGreenMaterial.setColor(Material.EMISSIVE, 0x0000002a);
}

```

```

blueGreenMaterial.setColor(Material.DIFFUSE, 0xFF876A56);
blueGreenMaterial.setColor(Material.SPECULAR, 0x0000a0a0);
blueGreenMaterial.setShininess(90.0f);

// initialize
trOrbitingPlanetoid = new Transform();
trBigRemoteBlueGreen = new Transform();

trOrbitingPlanetoid.postTranslate(-40.0f, 0.0f, -10.0f);

float bigScale = 100.0f;
trBigRemoteBlueGreen.postTranslate(100.0f, 200.0f, -800.0f);
trBigRemoteBlueGreen.postScale(bigScale, bigScale, bigScale);

```

Once everything is loaded and initialized, it is time to draw. The M3G model is analogous to that of the SVG model. Firstly we bind our standard MIDP Graphics context to the static instance of the Graphics3D class from JSR-184. Since we are in immediate mode we need to set up our camera each time we draw, execute a series of graphics commands and then release our graphics context.

```

private void draw3D(Graphics g) {
    try{
        g3d = Graphics3D.getInstance();
        g3d.bindTarget(g, true, Graphics3D.ANTIALIAS |
                        Graphics3D.TRUE_COLOR |
                        Graphics3D.DITHER);

        g3d.clear(backGround);
        g3d.setCamera(camera, gameEngine.getPlayerLocation());
        drawUniverse();
    }
    catch(Exception e){
        e.printStackTrace();
    }
    finally{
        g3d.releaseTarget();
    }
}

```

The Graphics3D class has a number of render() method overloads, the most useful being the one that takes a Node and a Transform as parameters. This allows us to use the technique outlined above simply by changing the mesh material and supplying a different transform to the render() method.

```

private void drawUniverse(){
    ...
    // move the small orbiting planetoid
    trOrbitingPlanetoid.postTranslate(-1.0f, 0.0f, 0.0f);
    trOrbitingPlanetoid.postRotate(-1.0f, 0.0f, 1.0f, -0.6f);
    // draw the planetoid
}

```

```

sphere.setTransform(trOrbitingPlanetoid);
appearance.setMaterial(planetoidMaterial);
g3d.render(sphere, trOrbitingPlanetoid);
// now draw the distant stationary planet
sphere.setTransform(trBigRemoteBlueGreen);
appearance.setMaterial(blueGreenMaterial);
g3d.render(sphere, trBigRemoteBlueGreen);
}

```

Normally in 3D games you need a fair bit of vector and matrix mathematics to determine if any two objects collide in space. However, in an FPS game, such as ***Finding Higgs***, which has a high rate of fire and runs on a mobile phone, performing these calculations to determine if your shot hit its target can very quickly become a performance bottleneck.

In M3G, collisions can be detected using the `pick()` method of the `Group` class. This method takes two vectors and a `RayIntersection` instance as inputs. The first vector is a position in space and the second is a direction. What happens (see Figure 8.9) is that the `pick()` method finds the first object in the group (in z-order) that was intersected by a ray fired from the position represented by the first vector (the position of the player in space) in the direction of the second (down the axis of the gun towards the attackers).

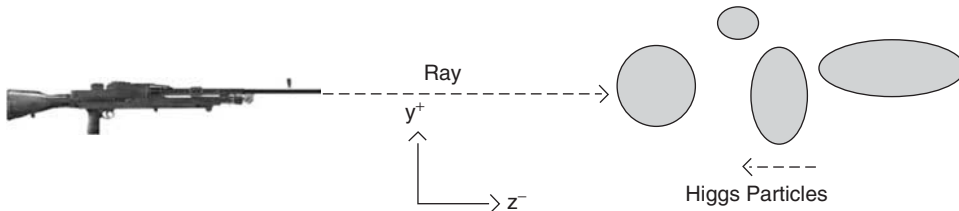


Figure 8.9 Detecting collisions in M3G

It only performs the check against group members for which picking is enabled and on return the `RayIntersection` instance contains information such as the `Node` that was picked, the distance to it, and so on. If no node was picked, the `getIntersected()` method returns null.

It should be obvious that this saves us a lot of work in our sample game. In our context, ‘picking’ means getting shot, so to determine if we hit an attacking Higgs particle, we only need to do the following:

```

private void handleWorldCollisions(){
    if(isFiring()){
        ...
        // extract vectors from current player transform
        float[] plyrPos = new float[16];
    }
}

```

```

trPlayer.get(plyrPos);
RayIntersection rayIntersection = new RayIntersection();
attackGroup.pick(-1, plyrPos[3], plyrPos[7], plyrPos[11], // player loc
    - plyrPos [2], - plyrPos [6], - plyrPos [10], // direction
    rayIntersection);

Node hit = rayIntersection.getIntersected();
if(hit != null){
    numberOfHits++;
    updateScore();
    hit.setRenderingEnable(false); // don't show it again
    hit.setPickingEnable(false);   // don't allow it to be picked again
    ...
}

```

One last point to make with our sample game is that it also demonstrates the large effect that the number of pixels to be updated has on frame rate. As you move the camera around the 3D universe, the large blue-green planet in the distance comes in and out of view. Since this game always displays the current frame rate in the top left corner you can see the frame rate vary sharply as this happens. As you do this take into account that the small orbiting planetoid is the same mesh (therefore has the same polygon count) but occupies far fewer pixels on the screen.

8.4.7 Using the Bluetooth API (JSR-82)

Multiplayer games are one of the fastest growing areas in the games industry and one of the great things about mobile phones is multiplayer games over Bluetooth. Bluetooth is a wireless radio technology that operates in the 2.4 GHz range over distances of up to 10 meters. These days, we use Bluetooth to connect all types of peripheral devices so it is a great technology to know how to use. Before proceeding, we recommend reading an excellent series of articles on multiplayer games available from the Sun Developers' Network,¹¹ which gives a great background into a number of concepts.

Bluetooth can be reasonably difficult to understand at first because, unlike most other areas of networking, the actual topology of the network can be very dynamic as new devices may come in and out of range at any time. Even the process of establishing a connection between two Bluetooth devices is unusual because connections are initiated from the server (or 'master') device to clients that listen on server sockets. Most examples you find online mix UI code directly in with connection management code and events, often making it difficult to focus on the key concepts.

The Bluetooth code included in our sample game has been taken directly from Forum Nokia,¹² specifically Sections 5.1 and 5.2 of 'Games

¹¹ developers.sun.com/mobility/allarticles/#games

¹² Bluetooth API Developer's Guide and Games over Bluetooth.

over Bluetooth'. The main changes we have introduced are the use of an observer class and simplification of the code to support only two players. If you wish to learn about Bluetooth technologies in detail, these articles are required reading as we cannot cover everything here.

Selecting the Multi option from the main game menu (see Figure 8.6) displays the multiplayer menu shown in Figure 8.10. In this game, if you choose Player1 you are the 'Master' (or game server, if you like). If you choose Player2, you are the 'Slave'.



Figure 8.10 Selecting the role

Before connections can be established between Bluetooth devices, the Bluetooth stack must be initialized, devices must be found (known as the inquiry phase) and then the available services on each device must be queried (known as the discovery phase).

In the inquiry phase, the master device detects other devices in the area in order to determine their types and Bluetooth addresses. The type of device is referred to as the class of device (CoD). The CoD is simply a number that indicates whether the remote device is a computer, an audio headset, or a mobile phone. In JSR-82, this is represented by the `DeviceClass` class. You retrieve the CoD by calling the `getMajorDeviceClass()` method, which returns an integer. These integers are well-known, centrally assigned fixed numbers – for example, mobile phones have the value 0x0200.

There are two modes commonly used to find other devices – the General Inquiry Access Code (GIAC) and the Limited Inquiry Access Code (LIAC). The Symbian OS Bluetooth stack does not support the use of the LIAC so we won't discuss it any further here.

The following code snippet shows how to start a search for nearby devices using JSR-82 classes:

```
// this throws a BluetoothStateException if Bluetooth is not enabled,
// which is a good way to check that it is before proceeding
LocalDevice localDevice = LocalDevice.getLocalDevice();
// start an inquiry for other devices in GIAC mode
DiscoveryAgent discoveryAgent = localDevice.getDiscoveryAgent();
discoveryAgent.startInquiry(DiscoveryAgent.GIAC, listener);
```

Notice the `listener` variable passed on the last line above. This is a reference to an object that implements the `DiscoveryListener` interface which defines a handler interface for a series of callbacks that occur during the inquiry and discovery phases. To stop an inquiry in progress, you can also pass this handler to the `cancelInquiry()` method of the `DiscoveryAgent` class. For reference, this interface is shown below:

```
public interface DiscoveryListener
...
// called once for each device found during the inquiry phase
void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod);
void inquiryCompleted(int discType);
// called during service discovery
void servicesDiscovered(int transID, ServiceRecord[] servRecord);
void serviceSearchCompleted(int transID, int respCode);
```

Once a set of devices matching the desired CoD has been located, they need to be queried in order to find out what services (identified using UUIDs) they offer. In the case of a Bluetooth game, we are interested in the game ‘service’ that we have built. Each device is queried in turn and the `servicesDiscovered()` callback method is called on the supplied listener, passing an array of `ServiceRecord` objects. These represent the various attributes of services found on the remote device. You can specify what attributes (such as, service name) that you want returned in this process as part of the discovery setup:

```
// an array of service ids we want to find
UUID[] uuid = new UUID[1];
RemoteDevice remoteDevice;
...
// include service name attribute in returned service records
int attribs[] = new int[]{0x0100};
...
discoveryAgent.searchServices(attribs, uuid, remoteDevice, listener);
```

The discovery process callback methods are also supplied with a transaction identifier that can be used to stop an active service discovery process by passing this to the `cancelServiceSearch()` method of the `DiscoveryAgent` class.

Once devices and services have been located and queried, we are ready to open a connection. As already mentioned, setting up a Bluetooth connection can be a little confusing at first because, unlike in other networking scenarios, connections are initiated by the server rather than the client. To make sure that our design remains clean, we've defined an observer interface that is notified whenever our Bluetooth state changes:

```
public interface IBluetoothObserver {
    public void notifyStateChange(int newState);
    public void notifyMessageReceived(String message);
}
```

In Figure 8.5, note that the `GameController` class implements the `IBluetoothObserver` interface shown above. Using an observer is good practice as it keeps UI-related code separate from Bluetooth management code.

The listing below shows the states that are defined in the `BluetoothManager` class for this game. You can see that each state change notifies the registered observer instance in the `setState()` method:

```
// states
public static final int IDLE = 0;
public static final int DETECTING_BLUETOOTH = 1;
public static final int BLUETOOTH_ENABLED = 2;
public static final int BLUETOOTH_DISABLED = 3;
public static final int FINDING_DEVICES = 4;
public static final int FINDING_SERVICES = 5;
public static final int WAITING = 6;
public static final int CONNECTING = 7;
public static final int CONNECTED = 8;
public static final int CONNECTION_FAILED = 9;
public static final int CANCELLING = 10;
public static final int DISCONNECTED = 11;
// called on state transitions
private void setState(int state){
    this.state = state;
    observer.notifyStateChange(state);
}
```

It is the slave (client) that opens a server socket and waits for a connection request from the master as illustrated in the code snippet below taken from the `BluetoothManager` class in our sample game:

```
String SERVICE_URL = "btspp://localhost:" + SERVICE_UUID;
private void startClientMode(){
    ...
    // advertise that we have the game
    String clientName = localDevice.getFriendlyName();
    String url = SERVICE_URL + ";name=" + clientName;
    ...
    notifier = (StreamConnectionNotifier) Connector.open(url);
    // wait for server-initiated connection
    connection = (StreamConnection) notifier.acceptAndOpen();
    // get I/O streams
    inputStream = connection.openInputStream();
    outputStream = connection.openOutputStream();
    ...
    // notify our Bluetooth observer that we are connected
    setState(CONNECTED);
}
```

Since the master device has already run an inquiry for devices and discovered their available services, all that remains is for the master to initiate a connection to the remote device (the slave). Bluetooth addresses are cumbersome and change frequently so the master uses the `getConnectionUrl()` method of a remote device's `ServiceRecord` to determine how to open a connection to it:

```
private void startServerMode(){
    ...
    ServiceRecord serviceRecord;
    ...
    // specify connection settings
    int security = ServiceRecord.NOAUTHENTICATE_NOENCRYPT;
    boolean mustBeMaster = false;
    // determine url to use given above
    String url = serviceRecord.getConnectionURL(security, mustBeMaster);
    // initiate the connection
    connection = (StreamConnection) Connector.open(url);
    // get I/O streams
    inputStream = connection.openInputStream();
    outputStream = connection.openOutputStream();
    // notify our Bluetooth observer that we are connected
    setState(CONNECTED);
}
```

Once connections have been established you can use normal read and write methods to send data between the connected handsets. In ***Finding Higgs***, we have kept data transfers minimal as the main point of them is simply to keep each player aware of the other player's current score.

Of particular interest, however, is that in this case we want the game on each handset to start at the same time (as closely as possible) since it's only a one-minute game and it's far less fun when your opponent finishes five seconds before you do because they started five seconds earlier. To achieve this, we wait until each device is connected, show the game screen (which may be faster on more powerful hardware) and *then*

inform the other device that it is ready. Only when both devices have received the initialization message do they start the game loop:

```
// called when Bluetooth connection is established
public void startNewGame() throws Exception {
    ...
    controller.getDisplay().setCurrent(gameScreen);
    notifyInitialised();
    ...
}

private void notifyInitialised(){
    try{
        gameStatus = GAME_STATUS_INITIALISED;
        if(isMultiPlayerGame()){
            bluetoothManager.sendMessage(MSG_GAME_INITIALISED);
        }
        else{
            // single player game so just start
            startGame();
        }
    }
    ...
}

// called whenever the Bluetooth connection receives a message
public void notifyMessageReceived(String message){
    if(message.equals(MSG_GAME_INITIALISED)){
        startGame();
    }
    else{ // remote player score updated
        gameScreen.handleMessage(message);
    }
}
}
```

Once everything has been constructed and synchronized, we only exchange messages when either player's score changes. In this case, it is our `GameEngine` instance that handles this:

```
// called from game thread - update local points and notify
// remote player when playing multi-player game
private void updateScore(){
    try{
        pointsP1 += POINTS_PER_ATTACKER;
        if(gameController.isMultiPlayerGame()){
            bluetoothManager.sendMessage(String.valueOf(pointsP1));
        }
    }
    ...
}
```

In a more complex game, it would be advisable to define a message abstraction class instead of just passing strings back and forth. For example, a common approach is to use messages containing pre-defined integer codes (opcodes) and an optional payload. However you do it, in

the end your approach need only match your requirements and you can be as complex or simplistic as you need.

8.5 Summary

We have covered a fair amount of ground in this chapter. We started with what a game actually is and ended with Bluetooth multiplayer functionality. In the process, we learned a lot about mobile game development in general as well as some of the more common pitfalls you can avoid. The rich set of features that the MIDP 2.0 Game API provides developers were discussed and the demo game ***GhostPong*** showed what even an exhausted author managed to do in only an hour with this groundbreaking mobile game framework.

Section 8.4 went quite a bit deeper and looked at how you can leverage the rich set of JSRs that Symbian OS provides the Java ME developer. We discussed the planning and design of a sample game, ***Finding Higgs***, and showed how we can be a little inventive and push the boundaries with Symbian OS because it does not place hard constraints on Java ME, unlike many other mobile operating systems.

We reviewed the basics of how to use the multimedia, scalable 2D vector graphics, 3D graphics and Bluetooth libraries to build an advanced mobile game and we looked at some of the open-source tools available to help you build compelling game content.

We did not touch on the topic of Nokia's Scalable Network Applications Platform (SNAP) initiative. This is a framework for multiplayer games using Java ME and should be considered a key indicator of current market trends. For more information on SNAP, see Appendix B.

We hope you're as excited about Java ME games as we are and that, from this point on, the only thing you'll be thinking about is how many games you can write, sell and play. Remember – with Symbian OS, you're only limited by your imagination so good luck!



9

Java ME Best Practices

Applying best practices is an essential part of the development process. There is so much to be considered here – quality, user experience, testing, future-proofing, optimizing resource usage, and so on. Clearly, there is no single script we can follow to write an application. Nor is there a definite set of rules.

We can, however, propose some common practices and API usage patterns. First, we explore how to improve the user experience. Second, we delve a level deeper and explore some practical Java ME patterns, including for resource usage. Then, we offer some ideas for streamlining the deployment and lifecycle of the ultimate MIDlet. Finally, we look at general guidelines for developing on Symbian OS.

9.1 Investing in the User Experience

The user experience is paramount in a Java ME application, as in any other user-facing environment. You'll have noticed discussions about the user experience throughout the book – and the reason is, clearly, that we all write software for the end user, even though that is often lost in the 'noise' of software development.

Getting the user experience right requires deliberate investment. One approach is defining key use cases in advance and making sure they are easy to access and perform well. Not all applications are use-case driven, however. In this case, we can occasionally put on our 'user' hat – assume the role of the application user – and see how we can improve the application. The two approaches can also be combined and help identify and eliminate usability problems – overly complex menus, unresponsiveness, error handling, and so on.

But what is the user experience? Clearly, there are many factors, but let us name a few:

- interface design – visually pleasing interface
- interaction design – ease of use, intuitiveness of the interface
- responsiveness
- resilience – the way our application behaves in special situations

User interface and interaction design are well-researched topics and are beyond the scope of this book. Interested readers can find a wealth of information in [Ballard 2007].

9.1.1 Supporting Input Methods

There is one aspect of interaction design that is worth discussing here – simply because it is specific to Java ME to an extent and rarely covered in books discussing this topic.

Different Java ME devices allow for different input methods, such as keypad and pointer. A good Java ME application should support both but be resilient to absence of either. For example, an MIDP Sudoku game that only allows keypad controls could be very frustrating for a touchscreen device user.

While `Form`, `List`, and similar screen implementations should offer this functionality out of the box, when using `Canvas` and its subclasses, we are on our own. For more information on supporting input methods, refer to Section 4.6.

9.1.2 Improving Responsiveness

A well-written application must produce a response to user input. If tapping the screen or pushing a button does not produce visible results quickly, the user doesn't know if their command has been received and is executing or was not received at all. Additionally, if you have a long operation running, you should keep the user informed by displaying the progress so that the user can see that something is indeed happening and can estimate when this operation will end.

There is another key point in making sure the application is responsive – never block a system thread. Clearly, this requires some knowledge about Java ME run-time implementations and a bit of proficiency with threads but it can be distilled into a set of simple rules.

Let's first briefly discuss Java ME run-time system or 'infrastructure' threads. First, system threads are not specifically defined in the Java ME or MIDP specifications. However, they are an implementation detail

present in all Java ME implementations we have come across. There are, in essence, three threads we need to know about:

- the lifecycle thread
- the event dispatcher thread
- the painter thread.

The lifecycle thread invokes MIDlet lifecycle methods such as `MIDlet.startApp()`, `MIDlet.pauseApp()` and `MIDlet.destroyApp()`. Our rule in this case translates into avoiding any blocking operations inside these methods, such as opening and reading streams, creating or loading images, and so on. Such tasks should be performed in a separate thread if at all possible. Consider this MIDlet startup example:

```
class MyLoader implements Runnable {
    MyMIDlet midlet = null;
    public MyLoader(MyMIDlet midlet) {
        this.midlet = midlet;
    }

    // implementing abstract method run() from Runnable
    public void run() {
        // various long-running tasks, eg. load pictures
        ...
        midlet.view = new MenuView();
        midlet.notifyDone();
    }
}

class MyMIDlet extends MIDlet {
    SplashScreen splash = new SplashScreen();
    MenuView view = null;

    protected void startApp() throws MIDletStateChangeException {
        // sets the Display
        Display.getDisplay(this).setCurrent(splash);
        // starts a new thread to load other views and data;
        MyThread loader = new MyThread(this);
        loader.start()
    }

    public void notifyDone() {
        // at the end of processing set up display to be MenuView
        Display.getDisplay(midlet).setCurrent(view);
        // make the splash screen eligible for garbage collection
        splash = null;
    }
}
```

This is a typical example of neat application startup handling. The system-called `startApp()` method completes immediately as it just sets the current screen to a lightweight `SplashScreen` and launches a thread

to initialize the heavier-to-load `MenuView`. Once loading is complete, the thread sets the current screen to `MenuView`.

The event dispatcher thread calls our event handlers, such as `commandAction()`, `keyPressed()`, and `pointerDragged()`. Again, the rule is never to perform blocking operations inside event-handling methods. It is a common mistake to try to open a connection or do UI construction work inside a `commandAction`. Consider, for example, that opening an HTTP connection could prevent the user from any interaction with our application until the connection is open. The worst-case scenario is that an opening connection hangs due to a broken DNS record or an incapacitated server. The user is at the mercy of some timeout that we can only hope exists in the HTTP implementation. Notice how simple and quick our event handlers are in the previous example showing how to handle both key and pointer events.

The painter thread is equally sensitive, however it is a very unlikely place to do any blocking operation with a potential to run for a long time. Most of the time, painting is carefully scrutinized for other reasons, so we normally don't expect blocking here.

9.1.3 Increasing Resilience

In a mobile environment, our MIDlet may be interrupted at any point in time. Typical reasons are an incoming call, a low-battery warning or loss of network connectivity. Handling these situations correctly is very important for the user experience. Some level of resilience can be achieved by expecting failures and handling unexpected errors gracefully. A more thorough approach could include carefully managing application state in the case of errors.

9.2 Good Programming Practices

9.2.1 Consider Hardware Limitations

At the time of writing, the Nokia N96 has just been released. It is one of the most powerful smartphones based on Symbian OS: it has a dual CPU ARM 9 processor running at 264 MHz, 128 MB of RAM and 16 GB of user storage memory. Several Symbian OS smartphones have support for VoIP, multiple media formats, 5 megapixel cameras, and GPS technology, among others. This is a remarkable evolution, considering that only four years ago Nokia's flagship 6600 was not capable of even playing MP3 files.

All this is great news for us as Java ME developers, since applications developed in it can be richer in features, cover more use cases and replace native application development in many scenarios. However, you should

keep in mind that, even though devices are not as resource-constrained as they used to be in the past, MIDlets must be developed with consideration for utilization of system resources. This ensures that they perform well and remain responsive, not only in easy scenarios but also when conditions become harsh, with many applications installed and running, fighting for processor time and memory resources.

9.2.2 Handle Strings Efficiently

Java strings offer great convenience in handling text, although it has frequently been argued that the String API encourages inefficient code. Clearly, it is our job to use strings efficiently and Java, including Java ME, comes with some built-in mechanisms that can help.

Use StringBuffer

`StringBuffer` is the ideal tool for any string composition that involves concatenating several strings. For example, the following code that concatenates five strings could be rewritten with `StringBuffer` and offer much better performance.

```
String message = "Error: " + emsg + " [code=" + code + "];
```

The code above results in the creation of several intermediate strings holding intermediate concatenation results. The intermediate strings are discarded immediately. So, not only have we taken time creating several unnecessary strings, the garbage collector also has considerably more work to do to clean up the discarded objects. The right way to do this is as follows:

```
StringBuffer messageBuf = new StringBuffer();
messageBuf.append("Error: ");
messageBuf.append(emsg);
messageBuf.append(" [code=");
messageBuf.append(code);
messageBuf.append("]");
String message = messageBuf.toString();
```

Granted, this looks less concise, but it can profoundly improve application performance.

Working with `String` and `StringBuffer` can result in large amounts of memory being used unexpectedly. The problem is this: when `StringBuffer.toString()` creates a `String`, the newly created `String` uses the `StringBuffer` character array. This means that if a `StringBuffer` with a 1 KB capacity only holds a 10-character string, the new `String` also uses a 1 KB character array to store 10 characters.

If the `StringBuffer` is modified, the `StringBuffer` array is copied in its entirety and then modified. Now both the `String` and `StringBuffer` have separate 1 KB character arrays. We have just lost the best part of 1 KB of memory! Repeating the process continues to use excessive memory, as we generate 10-character strings each of which uses a 1 KB character buffer.

Use `InputStreamReader` to Read Strings

This is a pretty obvious statement, but unfortunately it appears to be necessary to mention because we often see developers casting `byte` to `char`. The problem with casting here is that we are ignoring the encoding that was used to store characters in the first place. Unicode formats, especially UTF-8 which is commonly used in Java programs, can have a variable number of bytes per character hence they cannot be decoded byte-by-byte. `InputStreamReader` efficiently performs the conversion for us so that we don't need to worry about decoding.

Use `String.intern()` (or not)

In essence, `String.intern()` takes a string and returns a JVM-wide unique reference to a string with the same content. This helps preserve memory because it allows us to have only one `String` instance representing a specific string. For example, all string constants in our classes are interned.

Although `String.intern()` is available for use in any Java application, we should only use it with great care. There is no mechanism to un-intern a string, so once you have called `String.intern()`, it remains in memory until JVM shutdown. It has been suggested that if we use `String.intern()`, we can use reference comparison instead of the normal `String.equals()` for comparing strings. For example, consider the following code:

```
// Be careful about using intern - it has hidden costs
string1 = string1.intern();
...
string2 = string2.intern();
...
if(string1 == string2) { /* do something */ }
```

A call to `String.intern()` includes a call to `String.equals()` so using `String.intern()` to 'speed up equals' is usually not a great idea.

A good use of `String.intern()` may be if we have many occurrences of a particular string throughout the application runtime. Using `String.intern()`, we can preserve memory that would normally be

used to keep copies of the string. Always keep in mind, however, that once interned, the string is never garbage collected.

9.2.3 Manage Objects Carefully

Java offers great benefits for developers by providing garbage collection. It is, however, still necessary to take care when managing objects.

First, let us reiterate the rules. An object becomes eligible for garbage collection when there are no references to it in any of the ‘alive’ threads. In essence, this translates either to objects going out of scope (e.g. local variables) or to setting class-scoped variables to `NULL` when we don’t expect them to be used any more. It is important to note that a Java object may be associated with a native peer with high native memory consumption (e.g., graphical and UI elements). The allocated native memory, which is outside the Java object heap, is not freed and made accessible to native applications until the associated Java object is discarded and garbage collected.

For example, consider the MIDlet startup example in Section 9.1.2. First, let us consider the lifecycle of the `MyThread` instance (`loader`) created in `startApp()`. Being declared inside a method, `loader` immediately falls out of scope of the current thread. However, it is a reference to a running thread and is not eligible for garbage collection. Once `loader` is complete however, the `MyThread` object is not reachable by any further running threads and is therefore eligible for garbage collection.

Now consider our use of `SplashScreen`. Since we never need the `SplashScreen` instance after our MIDlet has started (once the `MenuView` has been initialized and set as the current screen), we discard our class-scope reference to make the `SplashScreen` instance eligible for garbage collection. The benefit of doing this may be small for lightweight objects, however our `SplashScreen` is likely to have had references to `Image` objects, e.g. for off-screen drawing, and it could amount to significant memory savings.

One important technique for managing objects is to use object pools. A pool is essentially a collection of reusable objects. Instead of a create–use–discard cycle for an object, we may obtain one from a pool, use it and then release it back to the pool for further usage. In doing so, we have potentially saved some time in object creation and initialization as well as reducing the garbage collection load. Pools are extremely helpful, typically, for managing reusable network connections or objects that are frequently created and destroyed (e.g. buffers). A simple pool implementation is shown in the code sample below:

```
public class BufferPool {
    byte [][] buffers;
    int count = 0;
    int defaultBufferSize;
```

```
public BufferPool(int poolsize, int defaultBufferSize) {
    buffers = new byte[poolsize][];
    this.defaultBufferSize = defaultBufferSize;
}
public synchronized byte[] getBuffer() {
    if ( count == 0 ) {
        return new byte[defaultBufferSize];
    }
    return buffers[--count];
}
public synchronized void releaseBuffer(byte[] buffer) {
    if ( count == buffers.length ) {
        return; // exceeds pool size, discard
    }
    buffers[count++] = buffer;
}
}
```

If the pool is to be used by a single thread, you can remove the `synchronized` keywords for slightly faster execution.

9.2.4 Use Images Efficiently

Images are essentially heavyweight objects that have the potential to use large amounts of memory. There are two main practices for using images in MIDlets efficiently: use the right format and resolution, and dispose of unused images as soon as possible.

In general, most MIDP implementations provide support for the PNG image format. The good thing about PNG format is that it is lossless (i.e. no image information is lost during compression). This is great for many use cases, for example, images with solid and contiguous colors (such as, corporate logos and charts) or when transparency is required. However, using PNG for photographic images or complex artwork is not as efficient – image files tend to be large and take longer to load. For such images, we can recommend JPEG format. They have a relatively smaller heap footprint and file size, and are supported on all Symbian OS devices supporting JTWI (see Chapter 2). PNG files are the minimum supported by MIDP specification but, as most of the devices today are JTWI-compliant, JPEG support is becoming widely available, especially on Symbian OS devices.

Image resolution and color depth should be kept to the minimum which allow the proper display of the image. For example, there's no need for 24-bit color depth images if an 8-bit image provides satisfactory results. Another point favoring the use of JPEG format is that some platforms based on Symbian OS, such as S60, have hardware support for it, making the decompression of JPEG images before viewing very fast compared to PNG. A good place to learn more about JPEG and PNG file formats is [Rutter 2003].

As discussed above, images are 'heavy' objects, which allocate large native buffers containing the image data. It is extremely important to have a well-defined image object lifecycle – essentially you must know when to dispose of images you no longer need so that the native buffers can be freed. Finally, if you need to load many images into memory before starting your application, you should do so in another thread, to prevent the blocking of the system thread.

9.2.5 Handle Exceptions

A fairly common mistake made by novice Java developers is declaring their heavyweight resources (such as, connections or streams) inside a try–catch block. For example, consider the following code sample.

```
// WRONG - never do this!
try{
    HttpURLConnection hc = (HttpURLConnection)Connector.open(url);
    InputStream is = hc.openInputStream();
    ...
    hc.close()
} catch(Exception e) {
    // handle error
}
```

If an error occurs while reading the stream, we immediately leave the scope of our try–catch block and are therefore unable to close the stream and connection. Some implementations e.g., Java ME on Symbian OS, may be able to recover using internal finalizers (if the implementation calls finalizers and the finalizers for affected objects close native resources). There are too many ifs and none of these behaviors are mandated by MIDP or JSR specifications. Finally, although the implementation of Java ME on Symbian OS has this level of resilience, there are no guarantees on when the object is garbage collected. This kind of error counts as relatively nasty resource leakage.

To be sure we clean up correctly, we can use the `finally` block as follows.

```
HttpURLConnection hc = null;
InputStream is = null;
try{
    hc = (HttpURLConnection)Connector.open(url);
    is = conn.openInputStream();
    ...
} catch(Exception e) {
    // handle error
} finally {
    if ( is != null ) { try{is.close();}catch(Exception e){} }
    if ( hc != null ) { try{hc.close();}catch(Exception e){} }
}
```

The key point here is that we have kept the references to potentially heavyweight objects. Since we clean up in the `finally` clause, our cleanup code is guaranteed to be executed regardless of whether the exception is thrown.

9.2.6 Buffer I/O

Most stream implementations in Java ME are not internally buffered. This means that reading or writing, a byte at a time, can result in extremely slow code. Therefore, it is always preferable to read and write using buffers.

Our first example copies from an input stream to an output stream, known as 'piping':

```
public void pipe(InputStream in, OutputStream out) throws IOException {
    byte [] buffer = new byte[1024];
    int read = in.read(buffer);
    while ( read > 0 ) {
        out.write(buffer, 0, read);
        read = in.read(buffer);
    }
}
```

The second example for buffered I/O shows how to fully read a stream into a buffer when we know exactly how much we need to read in advance.

```
public void readFully(InputStream in, byte[] buffer) throws IOException {
    int numBytes = buffer.length;
    int read = 0;
    while (read != numBytes) {
        int rd = in.read(buffer, read, numBytes - read);
        if ( rd == -1 ) {
            throw new IOException("End of stream!");
        }
        read += rd;
    }
}
```

If the length of the stream is unknown, we can use our pipe function with a `ByteArrayOutputStream`:

```
public byte[] readFully(InputStream in) throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    pipe(in, baos);
    return baos.toByteArray();
}
```

9.2.7 Use the Record Management System Correctly

When using the Java ME Record Management System (RMS), it is important to consider using filters and comparators. Filters are used to determine

what records satisfy the criteria for being included in the RecordEnumeration returned by a call to `RecordStore.enumerateRecords()`. Take this code as an example:

```
public class StringFilter implements RecordFilter {
    public boolean matches(byte[] candidate) {
        if(new String(candidate,0,16).equals("Employee")) {
            return true;
        }
        return false;
    }
}

StringFilter filter = new StringFilter();
RecordEnumeration enum =
    someRecordStore.enumerateRecords(filter, null, false);
```

In this code, `enum` now contains only records whose `String` representation of the first 16 bytes is 'Employee'.

Filters are simple to use, but you must remember that they are matched against every single record in a `RecordStore`. If you know the store won't grow much, it's OK to have complex filters that do comparisons based on a big part of the byte array. However, if there's a slight chance of your record store growing to hundreds or thousands of records, you need to optimize your filters. The following code, handling about 2000 records in the `RecordStore`, takes about two minutes to complete on a Nokia E61:

```
public class StringFilter implements RecordFilter {
    ...
    public boolean matches(byte[] candidate) {
        ByteArrayInputStream bais = new ByteArrayInputStream(candidate);
        DataInputStream dais = new DataInputStream(bais);
        int code = dais.readInt();
        int category = dais.readInt();
        if(code == product.getCode() && category == product.getCategory()) {
            return true;
        }
        return false;
    }
}
```

The problem with the above code is not RMS related. For each of the 2000 records present in the store, the code creates a new `ByteArrayInputStream` of a 200-something byte array, then creates a `DataInputStream` on top of it, to read just a couple of integers for comparison.

A much more efficient way to do the filtering is:

```
public class StringFilter implements RecordFilter {
    ...
    public boolean matches(byte[] candidate) {
        int code = arrayToInt(candidate,0);
```

```

    int category = arrayToInt(candidate,4);
    if(code == product.getCode() && category == product.getCategory()) {
        return true;
    }
    return false;
}
private int arrayToInt(byte[] b, int offset) {
    int value = 0;
    for (int i = 0; i < 4; i++) {
        int shift = (4 - 1 - i) * 8;
        value += (b[i + offset] & 0x000000FF) << shift;
    }
    return value;
}
}

```

The filtering logic has been changed to do the same thing without creating any objects, and using only primitive types. That fits perfectly with the primary reason for which the Java language has primitive types (which are not objects) – for efficiency. This simple change speeds up the search several times and the search completes in seconds.

RecordComparator implementations are used to order elements in a record store prior to their return to a RecordEnumeration, upon a call to RecordStore.enumerateRecords(filter, comparator, Boolean). Although filters and comparators serve different purposes, the advice is the same: keep your comparators fast and efficient, as they are executed in many records (in all records if the filter is NULL).

9.2.8 Use the Mobile Media API Correctly

Nearly all use cases requiring MMAPI deal with resources that are constrained in mobile devices: large amounts of memory (playing and recording audio and video files), hardware (camera, memory cards, disk) that is expensive with respect to performance, frequent use of the network and tight integration to native components. All these factors put together can make a multimedia application heavy and slow to use. To avoid this, let us look at some best practices for the main use cases of MMAPI.

The JSR-135 API is very abstract and relies on multiple optional components for proper functioning. Not all implementations of it support all multimedia formats or operations, therefore run-time checks are needed to ensure your MIDlet behaves properly on the current device. Let us consider the following example code:

```

/* Portions of the code omitted for brevity */
private Player player = null;
private VideoControl control = null;
private RecordControl record = null;

```

```

public void startRecording() {
    try {
        player = Manager.createPlayer("capture://video");
        player.addPlayerListener(this);
        player.realize();
        //set up recording
        record = (RecordControl) player.getControl("RecordControl");
        record.setRecordSizeLimit(300000);
        conn = (FileConnection) Connector.open(PATH, Connector.READ_WRITE);
        if (!conn.exists())
            conn.create();
        stream = conn.openOutputStream();
        record.setRecordStream(stream);
        // Grab the video control and set it to the current display.
        control = (VideoControl) player.getControl("VideoControl");
        if (control != null) {
            control.initDisplayMode(VideoControl.USE_DIRECT_VIDEO, this);
            control.setDisplaySize(getWidth(), getHeight());
            control.setVisible(true);
        }
        player.start();
        record.startRecord();
    }
    catch (Exception e) {
        Alert error = new Alert("Error", e.toString(), null, AlertType.ERROR);
        Display.getDisplay(midlet).setCurrent(error);
        e.printStackTrace();
    }
}

```

The code sets up a Player whose locator is `capture://video`, then proceeds to use the `RecordControl`, a `FileConnection` (JSR-75) and an output stream to record video from the camera and save it on the device's permanent storage. This code works well on Symbian OS devices supporting video recording in Mobile Media API. However, if any of the operations attempted here is not supported, for example, video recording on Nokia S60 2nd Edition or Series 40 3rd Edition devices, an error message is displayed which offers little useful information to the user. For example, if fetching the video stream from the camera is not supported, a cryptic message, 'Invalid Locator', is shown. This illustrates the need to perform all the required run-time checks, when using MMAPI, and handle errors gracefully. In some cases, the MIDlet may still work with limited functionality. For a video-recording application, however this is unlikely.

In our example, we must check for the MMAPI system properties in Table 9.1 before attempting to record video. Failing gracefully in this case also means giving the user a readable and self-explanatory error notification. Our corrected code example would then look like:

```

public void startRecording() {
    if (!System.getProperty("supports.video.capture").equals("true")) {

```



```
        //TODO: show clear message saying that video capture is not supported
        return;
    }
    else if(!System.getProperty("supports.recording").equals("true")) {
        //TODO:show clear message saying recording is not supported
        return;
    }
    else if(System.getProperty("video.encodings") == null) {
        //TODO:show clear message saying video recording is not supported
        return;
    }

    //continue with the rest of the code
}
```

Table 9.1 MMAPI System Properties

MMAPI System Property	Value
supports.video.capture	Returns true if fetching the video stream of the camera is supported or false if otherwise
supports.recording	Returns true if RecordControl is supported or false if otherwise
video.encodings	Returns a list of supported video encoding formats or NULL if video recording is not supported

Please note that in this particular code, checking for `video.encodings` would be enough. However, your application may have other options, such as taking pictures or recording sounds, therefore checking for all related properties is recommended. One other important piece of advice is that many developers get confused by the meaning of the `supports.video.capture` property. It does not tell us anything about video recording support. Instead, it says that capturing the video stream from the camera is supported. What is the difference? For instance, if `supports.video.capture` is `TRUE` but `video.encodings` is `NULL`, this means that video can be seen but not recorded. However, the camera stream can be used when taking pictures with the `VideoControl.getSnapshot()` method.

Besides checking for supported controls and operations, it is highly recommended that you check for multimedia formats and protocols supported. For more information on checking protocols and formats, see Section 4.7.

When creating multimedia applications, it is fundamental to remember that you will most likely be working with big files. For most audio and video formats, a file must be loaded entirely on a memory buffer so it can

be decoded and played. This process takes some time to complete when the data is in RMS, a JAR file or a regular file accessed with the File Connection API. It is also possible that the data resides in the network; in this case we are again subject to network conditions that can affect speed and reliability of I/O operations, as discussed in previous sections. The most immediate implication is that, in general, all your operations with `Players`, especially creation, realization and pre-fetching, should be done on a separate thread, to prevent these operations from freezing a system thread.

Keep in mind that this is valid not only for reading, but also for writing data. Let us look at the following excerpt from our previous example:

```
//setup recording
record = (RecordControl) player.getControl("RecordControl");
record.setRecordSizeLimit(300000);
conn = (FileConnection) Connector.open(PATH, Connector.READ_WRITE);
if (!conn.exists())
    conn.create();
stream = conn.openOutputStream();
record.setRecordStream(stream);
```

Here we are recording video from the camera and sending it to a file, up to a limit of 300,000 bytes (roughly 292 KB). A real-world application, however, would not have such a hard-coded limit, as it would severely impact its functionality. A more realistic approach would be to record until the user decides to stop or the disk is full. In either case, there might be several megabytes of video in the memory buffer waiting to be flushed to the disk when `RecordControl.commit()` is called. This operation should also be moved to a separate thread, as writing millions of bytes to the disk will take some time and doing so in the system thread will most certainly hang the application from the user's standpoint.

While in principle those cases should be self-evident, at times they may be hidden in code sections that apparently don't perform any heavy operations. Consider this example for capturing a snapshot with the device's camera:

```
Player p;
VideoControl vc;
// initialize camera
try {
    p = Manager.createPlayer("capture://video");
    p.realize();
    // Grab the video control and set it to the current display.
    vc = (VideoControl)p.getControl("VideoControl");
    if(vc != null) {
        Form form = new Form("video");
        form.append((Item)vc.initDisplayMode(vc.USE_GUI_PRIMITIVE, null));
        Display.getDisplay(midlet).setCurrent(form);
    }
    p.start();
}
```

```
catch (IOException ioe) {  
}  
catch (MediaException me) {  
}  
// now take a picture  
try {  
    byte[] pngImage = vc.getSnapshot(null);  
    // do something with the image ...  
}  
catch (MediaException me) {  
}
```

Apparently this capture can be performed in the same thread, since it's not reading or writing any data from or to the disk or network, therefore no I/O operations are being executed. A closer examination however proves this to be wrong. In older devices, such as the Nokia 6600 (S60 2nd Edition), the supported resolution was only 160×120 pixels, making a photo capture very fast since there was not much data to copy from the camera buffer to a Java byte array. However, newer devices these days support resolutions of up to 1600×1200 pixels, which in JPEG format, with reasonable quality, can result in about 550 KB of data! Copying all these bytes from one buffer to another is an expensive process, which can easily hang the main application thread, preventing the user from interacting with the application. As a result, it should also be performed on a separate thread.

We mentioned at the beginning of this section that for most audio and video formats, data must be loaded entirely into a memory buffer before it can be decoded and played. Notable exceptions are the streaming formats, such as Real-Time Streaming Protocol (RTSP). Therefore, besides expensive I/O operations being performed to load the data, players for those formats keep huge buffers in memory, containing the decoded data ready to be played. This makes the players very memory-intensive and their use must be accompanied by careful memory management.

There are two best practices on memory management for `Player` objects: do not keep too many pre-fetched players in memory and explicitly close any unneeded players.

As we discussed in Chapter 2, players go through some states before they are ready to play media data:

- `UNREALIZED`, where it has just been created
- `REALIZED`, where it acquires the media resources it needs (by communicating with a server, reading a file, etc.)
- `PREFETCHED`, where it acquires scarce or exclusive resources (an audio device, for example), fills buffers with data and does other start-up processing.

The Player's five states and the state transition methods are summarized in Figure 9.1. As you can see, once a player reaches the `PREFETCHED` state, it is using all possible memory and resources to allow the immediate start of media playing. As its resource consumption is at maximum, we must not keep too many prefetched players together in memory; the risk is of slowing down the application or even running out of memory.

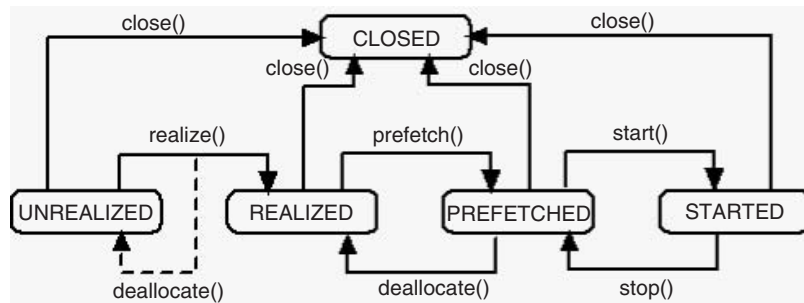


Figure 9.1 Lifecycle of a `Player` object

You must also not reload media from the disk or the network frequently. Let us say you want to play an MP3 file each time your game reaches a certain stage. For this you create a `Player`, call `start()` on it, play the sound, and close it to save memory. The next time the game reaches the same stage, you do this all over again. Essentially, what's being done here is to reload data constantly, from the disk or the network, by calling `close()` on a `Player` effectively renders it unusable, by freeing all resources it is using. This situation seems to conflict with the previous one: if one must not reload data frequently or keep many prefetched players in memory, what is the optimal approach for reusing sounds in an application?

There are many answers to this question. First, if you need to use multiple players, you want to make sure that the data they are using is small and efficient. Second, keep only players that you need immediately in the `PREFETCHED` state; all the others must be moved to the `REALIZED` state, where they are not consuming scarce resources. This can be achieved with a call to `Player.deallocate()`, which moves the player back one state. Last, but not least, if you are using a player only once, as soon as you are done with it, call `close()` so all its scarce resources, media buffers and memory can be disposed of.

As we have already mentioned, MMAPi players use a lot of memory, by holding buffers where media data is stored. Therefore it's necessary that you use them efficiently with regards to memory, by calling `close()` explicitly on an unused player, to free its resources as soon as possible. We

discussed in Section 9.2.3 how the garbage collector cleans up objects running out of scope, for example, a `Player` stored in a local variable inside a method is freed automatically. However, using this approach does not give you any guarantees on when this will happen (since the collector's behavior is non-deterministic); this is a good enough reason for calling `Player.close()` whenever it's needed.

We also talked about how explicitly trying to help the garbage collector usually does not work. Correctly scoping your variables and setting them to `NULL` when needed should ensure your application works smoothly without running out of memory. Calling `System.gc()` explicitly, in general, has no beneficial effect and can hurt performance if done too often. However, when dealing with the Mobile Media API, sometimes you will have many players active, consuming a lot of memory, and want to dispose of them quickly, to load a new batch of players with different sounds or videos. In most cases, just calling `Player.close()` is sufficient, but in some extreme situations (15–20 players) you may want to investigate calling `System.gc()` right after closing your players. This makes absolutely sure your application has enough memory to load more media as needed.

9.3 Streamlining the Deployment and Execution Lifecycle

We now look into best practices in the deployment cycle and the execution of the application as a whole.

9.3.1 Bundle Lightweight Libraries

At the time of writing this book, MIDP 3.0 LIBlets are not a supported standard and Java ME only allows us one JAR file per application. This is unfortunate, because some libraries not included in the Java ME implementation could potentially be shared between several MIDlets.

Mechanisms for provisioning and managing dependencies are typically overly complex for such limited hardware. Hence, Java ME shared libraries are defined in JSRs and bundled with the Java ME implementation. As a result we often need to include an external library with our MIDlet. Our advice is to keep in mind that libraries not specifically written for Java ME may cause performance and footprint problems. For example, we could take an existing regular expressions library that was not originally designed for embedded devices, but we would be much better off taking a lightweight library that was designed with constrained devices in mind.

9.3.2 Use Remote Portals

As powerful as mobile phones are becoming, there is often the case for removing some MIDlet functionality to a server. With the advancement

of cloud computing and the growth of software as a service (SaaS), it is becoming more common that computationally difficult or resource-hungry functions are performed on the server. For example, a mobile route-navigation application is better off keeping the intensive route calculations on the server side, rather than hogging the phone's CPU. Further, many applications use remote portals for storage or authorization, such as software license checking.

Removing functionality to a portal begs the question: What do we do when the network is not available? Applications that require cooperation by a remote portal could handle this situation by providing an offline mode. Clearly, not all functions can be available in offline mode, but the user may still be able to use the program. One way to improve offline mode usability is to keep an on-device cache of the crucial remote data. An excellent example of using a cache to improve offline mode can be found in [Yuan 2004].

9.3.3 Use Obfuscation

Class files carry enough information from the original source file to allow reverse-compilation of it back into a source file. Without obfuscation, your published classes can essentially be converted back to source and modified with little effort. While many developers don't find this worrying, it is not so welcome to commercial application vendors.

Obfuscators are intended to make this reverse compilation process less useful by ensuring that the decompiled code is not as readable as the original source code. Some obfuscators are so good that they are able to confuse decompilers enough to prevent decompilation altogether.

Obfuscators also use a variety of techniques to reduce code size and, to a lesser extent, enhance performance. For example, this can be achieved by removing unused data and symbolic names from compiled Java classes and by replacing long identifiers with shorter ones. The ProGuard obfuscator reduced the size of the GoSIP demo JAR file from 20 KB to 17 KB. Chapter 3 explained that in Symbian OS devices there is no hard limit to the JAR size. Therefore reducing the JAR size is an additional benefit but not the primary reason to use obfuscators when targeting Symbian OS devices.

9.3.4 Sign your Application

MIDlet signing was discussed in detail in Chapter 2. All important concepts were laid out and applied in several code examples, using protected APIs and highlighting the permissions needed to use each of them. However, one aspect of signing was not discussed there: its contribution to application usability.

If an unsigned MIDlet uses protected APIs, a `SecurityException` is thrown if permission is denied by default by the security policy in effect

on the device (see Figure 9.2a). The user may be asked for permission (see Figure 9.2b) and if it is denied, the `SecurityException` is thrown.



Figure 9.2 Unsigned MIDlet: a) Permission denied and b) permission request

For all but the most restrictive environments, where security domains have been customized by manufacturers or operators, the use of unsigned MIDlets is fine. Users grant permission when requested, perhaps once or twice during the MIDlet's lifetime, or set them to 'Ask first time' for a session-based grant. However, when it comes to JSR-75 (the `FileConnection` and `PIM` API), life is not so easy. Most Symbian OS devices come with 'Ask every time' or 'Not allowed' for reading or writing user data, and this makes it impossible for an application dependent on JSR-75 to be of any use. Security prompts are shown too often, when reading a new folder or listing contacts from the phonebook, and this renders all but the most basic applications unusable.

If you want to test this scenario in practice, it is recommended that you download the Image Viewer application that comes with [Nokia 2006c]. It is a basic file browser that can read and display folders and image files. Running on the Nokia N95 yielded no fewer than six security prompts from the moment it was started to the moment it was possible to see the first image. The same happens with another JSR-75 application, [Nokia 2006d]. Every time the user presses Find Contacts to read the phonebook, he is prompted at least four times with the same message.

The reason for such extreme prompting in JSR-75 applications is that it deals with very sensitive data: the user's contacts and files. If all MIDlets were to be granted access to this data, a number of nasty consequences could happen, leading to a breach in privacy from malware spamming the contact list or uploading private photos to some public website.

If your application is dependent on the `File Connection` or `PIM` APIs, you will want to sign your MIDlet so it becomes usable to your audience. As a side effect, your MIDlet also gets more permissions to use other protected APIs. For example, unsigned MIDlets have a maximum setting

of 'Ask first time' for network connections, but signed MIDlets can use 'Always allowed', so that users are never prompted again.

Signing improves overall security and usability of the application, therefore it is a good recommendation to go for it, even it involves some bureaucracy and lengthy processes. For more information please refer to the Java Verified program.

9.3.5 Consider OTA Provisioning

Provisioning applications is an important part of the application lifecycle and equally important for a good user experience. Simple provisioning can be achieved with little effort with some server-side cooperation. For example, over-the-air (OTA) installation with checking for updates can be implemented with relatively little effort.

For more streamlined provisioning, consider using JSR-124 Java EE Client Provisioning Specification which offers a specification for a server-side provisioning service. With built-in OTA support and pluggable application provisioning schemes, support for automatic deployment bundle selection based on device, and so on, it appears to offer a complete provisioning solution.

9.3.6 Encrypt Sensitive Data

All network communications, and particularly wireless communications, may be exposed to public networks and therefore there is the potential for a malicious third party to intercept or record data. In some cases, this is not a major concern – e.g. downloading web pages or images available on a public web address. However, for any sensitive data such as passwords, financial transactions, and so on, we should use secure means for data transfer, such as HTTPS or secure sockets. It is important to note that secure communications have much higher computational requirements than non-encrypted communications. Therefore it is advisable to use secure communications only when necessary.

9.4 General Tips for Symbian OS

So far in this chapter, we have covered various best practices for development in Java ME. Other best practices for Java ME could also apply. However, this book is specifically about Java ME on Symbian OS and therefore we now look at two important guidelines that are specific to Java ME on Symbian OS.

9.4.1 Be a Good Symbian OS Citizen

Symbian OS is an open platform that hosts various run-time environments and can run multiple applications simultaneously. With openness comes

responsibility and that brings us to asking kindly that your Java application is a good Symbian OS citizen. Apart from ensuring the quality of the application itself (e.g., fixing defects and ensuring correct functionality), the synergy of your Java application within the context of the Symbian OS run-time environment should also be tested.

Let us give a few examples of what it means for your Java application to be a good Symbian OS citizen:

- The Java application must not affect the use of the core system features or other applications. Whilst using the Java application, users should still be able to switch away and use the main system applications and features (e.g. phone, calendar/agenda, clock, and contacts).
- During extended usage, the application must handle exceptional, illegal, or erroneous user actions. At no time should the application crash or freeze, and it should exit gracefully from any application-specific exceptions. To ensure this, it is advisable to apply stressful test scenarios (such as, switching rapidly between applications and opening many other applications simultaneously) to the device whilst using the Java application as per its normal operation.
- All interruptions must be handled as a user would expect and the application should continue to operate normally after the interruption. For example, a game should store its state and pause, whereas a more passive or less interactive application should continue even during the incoming interruption.

As you can see, this is very broad and general advice. There are no hard, defined rules and we trust that you, as an application developer, will achieve high standards of quality and become a good Symbian OS citizen with some time and effort spent on thinking, planning and on-device testing.

9.4.2 Do Not Use Extreme Java ME Optimization Techniques

Symbian OS is a very powerful platform that does not impose hard limits on computing resources for Java applications, such as the JAR size, heap size, and so on (see Section 3.4). Having this power means that there is no need to apply the kinds of extreme measurements that might be needed on low-end devices. Applying extreme measurements can take its toll in areas such as the ease of adding a new feature in the future, code maintenance, and so on.

Let's give a few examples of extreme measurements that should be discouraged. It is sometimes stated that in Java ME it is better not to use Java interfaces (because of performance) and abstract classes (because of increased JAR size), and that it is better to collapse inheritance hierarchies

to minimize footprint, even if this means duplicating code. Such practices can indeed be required for low-end devices but they come at the expense of quality in areas such as code readability, reuse, extendibility, and abstraction level.

On high-end platforms, you don't need to fight for every kilobyte in the JAR by limiting your code abstraction or using the most compressed media formats even at the expense of the user experience. You can define interfaces and use abstract classes. You can and should do the right thing in order not to compromise on quality. This means that you can fully use generic software engineering best practices and provide an excellent user experience. If the right thing for your application is to run an additional thread or open an additional connection, you can do that. Having a proper code abstraction ensures the quality and extendibility of your application, without risking not being able to install or run your application on a Symbian smartphone.

Having said that, even a Symbian smartphone is limited in comparison to a desktop PC. Early prototyping of your resource utilization is another good practice that will give you early indication if the application design should be reworked. For example, if at design time it is known that the application requires a large amount of run-time memory or a large number of concurrent threads, then a simple resource utilization prototype that takes five minutes to write will assure you that this resource utilization is realistic.

9.5 Summary

In this chapter, we have covered various Java ME best practices. We explored techniques to ensure a good user experience, and some practical Java ME patterns and resource usage. We discussed deployment and the execution lifecycle and finished with guidelines that are specific to Symbian OS.

There are many more best practices that can be considered and there is no single set of rules that will ensure correctness and quality. We hope we have managed to highlight some of the most important best practices in a balanced way.

Part Four

Under the Hood of Java ME

10

Java ME Subsystem Architecture

The final two chapters of this book give you an insight into how things work under the hood of the Java ME platform. If you ever wondered what makes the virtual machine (VM) tick and click or how the Application Management System (AMS), profile, configuration and VM are built and how they interact, then the next two chapters give you a glimpse into the fascinating implementation of Java technology on Symbian OS.

This chapter gives an overview of the implementation architecture, and the interaction and roles of the various entities that comprise the Java ME platform, as a Symbian OS subsystem.¹ The next chapter deals exclusively with how the Symbian OS Java ME subsystem acts as an abstraction layer for Java applications to make them use the same API calls as native Symbian OS applications. These two chapters are also highly valuable for engineers who work in systems development and integration.

There are a few widespread misconceptions about the Java ME subsystem architecture that will be shattered after reading this chapter. When you have finished, you will easily spot the four mistakes in a sentence such as 'The KVM aborted the MIDlet installation'.

10.1 Java Applications and Symbian OS

From the point of view of the Symbian OS architecture, the Java ME subsystem is a replaceable component that Symbian OS licensees can extend, customize or replace altogether. Additionally, the Java ME subsystem, as a whole, is a native application just like any other standard native application (although it is a very large application) that is shipped with the device, burnt on the ROM.

So the Java ME subsystem is a native application that accesses operating system resources generally through the same publicly available APIs as

¹ The Java ME subsystem sources are under `\src\common\generic\j2me` in the Symbian OS source repository.

any other third-party application. The only difference is that it is a very big application. The phrase 'the tip of the iceberg' describes quite well the Java ME subsystem as seen from the point of view of the user of Java applications. Only a small part of something largely hidden can be seen (see Figure 10.1). Underneath every lightweight Java application, there is a very big and highly complex engine. There is a very good reason for that. The Java ME platform is a huge abstraction layer (composed of many internal abstraction layers) that lets you deploy and run a 'Hello, World' application in less than 10 lines of Java code on a variety of completely different platforms!



Figure 10.1 An iceberg

The Java ME specifications define a few core logical entities (see Figure 10.2), such as the application management system (AMS), the profile (MIDP) and the configuration (VM and base classes). However, the separation between entities is mostly logical, which leaves room for different possible implementations.

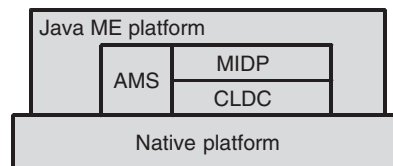


Figure 10.2 Core logical entities of Java ME

Although we discuss only the implementation on Symbian OS, there are other implementation models. For example, the open-source PhoneME project² takes a highly generic and modular approach that is designed

² phoneme.dev.java.net

to be easily portable to many different native platforms (e.g., BREW, Windows Mobile and Linux to name just a few). The Symbian OS Java ME subsystem takes a proprietary integration approach which is tightly coupled with Symbian OS. Let's see why.

10.1.1 How Symbian OS Differs from Other Java Hosting Operating Systems

There are a few key reasons as to why the Java ME subsystem takes a unique and proprietary approach to integration with Symbian OS.

First, Symbian OS is an open platform and has an existing application management model for installation and launching. MIDP application management must be integrated with the native model which makes any MIDP-only application management inappropriate.

Secondly, although versions of Symbian OS from version 9 support POSIX, it is not shipped on all devices and specifically not on devices before Symbian OS v9. The older and deprecated `stdlib` layer is not complete enough to support MIDP 2.0. That would leave most of the JSR functionality to be implemented using Symbian C++ APIs. Hiding the native Symbian C++ code under a generic platform-agnostic C layer is also not a proper solution because of the idiomatic use of Symbian OS services which would not fit well with a generic C layer.

Thirdly, as an application platform, Symbian OS has a rich set of native UI controls that are customized by different licensees to produce a distinct and precise look and feel to which Java applications must adhere.

These are some of the key issues that limit the reuse of code developed for other platforms to only the VM engine and Java packages that are implemented in pure Java. The general shape of the Java ME subsystem architecture is therefore unique and proprietary to Symbian OS. If at some point a legitimate question arises of possibly being able to implement a component in a more generic way or of reusing existing code from other platforms, please assume that it was considered but there was a good reason to opt for a Symbian OS proprietary approach.

10.1.2 Overview of Architecture and Main Processes

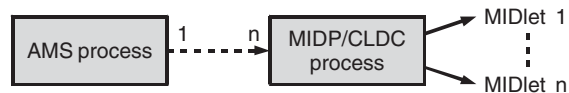
We take at least two points of view in this chapter: that of a hosted Java application (i.e., the point of view of an application developer) and that of the Symbian OS architecture (i.e., the point of view of an operating system integrator). Each has different logical groupings of Java ME components. So we had better start by clarifying some of the terms that are used across this chapter (see Table 10.1). Some groups overlap in the area which they point to but imply that it is being discussed from a different point of view. In cases where a certain component is not included in the

Table 10.1 Definitions of Component Groups

Group	Definition
Virtual machine (VM)	CLDC Configuration VM; the bytecode engine written in native code (Note: Java Configuration classes are not included.)
MIDP layer	MIDP implementation only (Note: the underlying Configuration and VM are not included.)
MIDP/CLDC run-time environment	Combination of the MIDP layer, Configuration classes and VM, as seen from the point of view of the Symbian OS architecture (Note: the AMS is not included.)
Java ME subsystem	Combination of the AMS and the Java ME run-time environment, as seen from the point of view of the Symbian OS architecture
Java ME platform	AMS and the Java ME run-time environment, as seen from the point of view of a hosted Java application
Java ME run-time environment	MIDP/CLDC environment, as seen from the point of view of a hosted Java application (Note: the AMS is not included)

group definition, although it is relevant to it, we explicitly note that it is excluded from that group definition.

Now let's take a first look at Figure 10.3, which depicts the two main processes in the Java ME subsystem implementation on Symbian OS. There are two main process types: the AMS runs in a single native process and takes care of the lifecycle and resource management of MIDlets amongst many other responsibilities; each MIDlet suite executes in the MIDP/CLDC run-time environment which runs in a separate native process.

**Figure 10.3** Interaction between the AMS and the MIDP/CLDC run-time processes

When the user wishes to launch a MIDlet, the AMS spawns a new child process executing the MIDP/CLDC run-time environment, for each application suite. That is, a new run-time process is spawned only for the *first* MIDlet that is launched from the suite. Additional concurrently running MIDlets from the same suite are executed in the same run-time process.

The interaction between the AMS and the MIDP/CLDC run-time process is not one way. There are a few cases which require the AMS to

invoke operations in the run-time process, and other cases which require the run-time process to invoke operations in the AMS. For example, when a Push connection is dynamically registered, the Java connection object will be managed by the MIDlet in the Java ME run-time environment and the native connection will live in the AMS process. The Java ME run-time environment will instruct the AMS to carry out the operation of opening the connection. For that purpose, and others, there are inter-process communications (IPC) mechanisms to allow each process to invoke remote operations on the other process.

The MIDP/CLDC run-time environment in Symbian OS is built from two major components: a replaceable VM that can come from different VM vendors (e.g., Sun and IBM) and a proprietary Symbian-developed Profile (e.g., MIDP 2.0) including the various JSR implementations. This separation is done to get the benefit of existing implementation reuse where possible.

There are two main insulation layers in the MIDP/CLDC run-time environment (see Figure 10.4). The first is the porting layer that every VM would have, which lets it receive basic services from the underlying operating system. The second is a layer which insulates between the MIDP layer and the VM. We will talk more about this separation and insulation layer in Section 10.4.

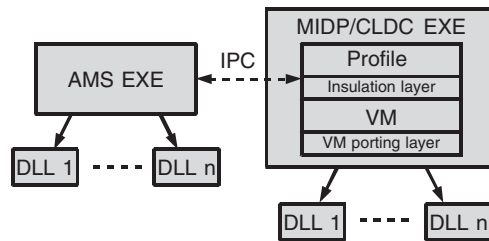


Figure 10.4 IPC mechanisms

Let's discuss how common functionality and customizable functionality are architected in the Java ME subsystem.

Due to the need for different licensees to customize various elements of the AMS and the Java ME run-time environment, both are delivered as a series of executable files. Such a design allows Symbian OS licensees to implement a specific variant of the functionality by localizing changes in a designated plug-in. The AMS and the Java ME run-time process each have one main executable (EXE) that is built from the common functionality code, and many smaller dynamically loaded libraries (DLLs) that are built from the variant functionality code. To give just a few examples: the AMS has DLLs for customization of the security module, MIDP Push modules, and so on; the MIDP run-time process has separate DLLs for customization

of the UI, multimedia support, and so on. This mechanism allows the Java ME subsystem to be easily adapted to different UI platforms (e.g., S60 or UIQ) and to comply with different OEM or operator requirements.

So far, this has been a high-level overview of the general architecture which defines the separation of logical entities into different processes, and how the various components are built and interact within the native environment. The following sections each focus on a specific area and discuss it in more detail.

10.2 Application Management Software

Application Management Software (AMS) is a generic term used by the MIDP specification to describe the software component that manages the downloading and lifecycle of MIDlets and listens for inbound notification requests on connections that are registered in the Push Registry.

The AMS implementation in the Java ME subsystem is called SystemAMS.³ SystemAMS provides the functionality described in the MIDP specification but also acts as a façade layer between Symbian OS and the Java ME subsystem as a whole. Internally, SystemAMS is decoupled into separate SystemAMS components (see Figure 10.5), each responsible for a specific area of functionality. Below is a non-exhaustive list of some of the SystemAMS components:

- Core: responsible for the management and ownership of the other components, including the servers that are used by Symbian OS to access the SystemAMS; it runs in the main SystemAMS thread
- Connection: responsible for all connection functionality, including Push connections
- Installer: responsible for all installation functionality; it runs in its own thread
- Launcher: responsible for the launching of MIDlet suites
- Lifecycle: responsible for managing the lifecycle of MIDlet suites
- Security: responsible for providing security-related functionality

As we discussed previously, the AMS runs in its own separate process, independently from the MIDP run-time process. There are two scenarios in which the AMS can be launched: at Symbian OS boot time or when the user launches or installs an application (and the AMS is not already started). At Symbian OS boot time, the SystemAMS is started in order

³SystemAMS sources are under `\src\common\generic\j2me\systemams` in the Symbian OS source repository.

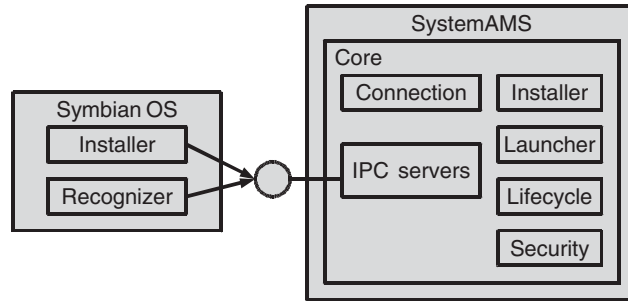


Figure 10.5 SystemAMS

to check if there are any connections registered in the Push Registry. If a preinstalled MIDlet has registered a connection, then the SystemAMS initializes the connection and waits for an inbound notification request. If there are no registered connections, then the AMS has no reason to remain active and can exit gracefully. Later, if the user launches an installed MIDlet or starts a new installation, then the Symbian OS Recognizer or the Installer connects to the SystemAMS IPC server which starts the SystemAMS.

In a similar way, after a running MIDlet has finished its execution, the SystemAMS checks if it is still needed to manage any resource and if not, it can exit gracefully.

By now you've probably noticed that the key areas of installation and launching involve entities from outside the Java ME subsystem. The MIDP installation process is the joint responsibility of the Symbian OS Software Installer within the Symbian OS Security Component and the SystemAMS Installer Component. SystemAMS must be involved in the installation process which implies maintaining an IPC interface to the Symbian OS Software Installer. The same applies to launching MIDlets, which also implies maintaining an IPC interface to the Symbian OS Recognizer.

10.3 MIDP Push

As defined by the MIDP 2.0 specification, there are two types of Push connection: static and dynamic. Static connections are specified by a MIDlet suite upon installation in the JAD file and are created at installation time. During suite installation by the Symbian OS MIDP Installer, SystemAMS attempts to open the specified Push connections. If this fails, the suite installation is rejected. Dynamic connections are set up programmatically when the MIDlet is already running.

From the point that a Push connection is registered, both types of connections are handled identically by SystemAMS. Upon reception of

an incoming connection, the associated MIDlet is started if it is not already running and accessing the Java connection results in IPC calls to the SystemAMS Connection component to perform the actual operation. For each Generic Connection Framework (GCF) protocol that supports Push, SystemAMS manages a connection plug-in that is loaded and initialized at SystemAMS boot time to recreate a static or dynamically defined server connection outside the context of a running MIDlet.

As a consequence of MIDP Push, all connections that *can* be registered to the Push Registry always live in the SystemAMS process. That remains true even if they are not registered as a Push connection, in order to prevent duplicate functionality between the MIDP run-time process and the AMS. Additionally, in the case of listening connections that can spawn a new connection when an incoming request is received, the newly created connection cannot be transferred to any other process. The new connection lives in the same process in which it was created (the SystemAMS process). The MIDP run-time process enables MIDlets to access those connections through IPC.

10.4 Mean and Lean Virtual Machine

When referring to the VM, the referral is always to the headless language and core classes engine only (i.e., no UI). At the time of writing this book, the majority of shipped devices include a port of Sun's CLDC-HI HotSpot VM 1.1 implementation (which implements v1.1 of the CLDC specification). However, the VM is a replaceable module in the Java ME subsystem; for example, in S60 3rd Edition FP2, the included VM is J9 from IBM.

VM technology contains a wealth of theory and practical tricks that make the JVM one of the most optimized pieces of code written for mobile phones. To peek into some of those goodies, we use the open-source CLDC-HI HotSpot VM as a reference (all sources are available in the PhoneME project⁴).

Just one note before we continue. Please do not use the term 'KVM' ... because there is no KVM any more! The KVM was a major milestone in Java ME evolution but it was put behind a glass box in the Java museum a long time ago. Symbian OS v7.0s devices (e.g., the legendary Nokia 6600) were shipped with Monty, which was an early version of CLDC-HI HotSpot. So the term 'KVM' is an ancient word which belongs to the Java hall of fame, but does not exist on today's devices. These come with VMs that are much more advanced than the KVM, which was slow (it was an interpreter only, no JIT) and had high memory usage.

We previously said that one of the few areas where the Java ME subsystem can reuse existing solutions is VM technology. So the VM

⁴ See phoneme.dev.java.net

needs to be ported to Symbian OS – which does not mean that the whole VM needs to be written from scratch! The VM needs to be written in a standard dialect of C or C++ that is compatible with the Symbian OS toolchain. Additionally, the VM must define and use some porting layer through which it receives basic services from the underlying platform. Our reference VM, the CLDC-HI HotSpot, is written in standard C++ and has a minimal porting layer. Interested readers can examine the implementation in the phoneME project repository.⁵ (Just to remind you, we are currently talking about the headless VM, not the full-blown Java ME run-time environment which includes MIDP.)

Below you can find a few examples of some of the CLDC-HI HotSpot porting layer interfaces that need to be implemented using the native operating-system APIs:

```
// All VM output goes through this method
void JVMSPi_PrintRaw(const char* s);
// Returns the time as System.currentTimeMillis()
static jlong java_time_millis();
// Allocate a memory chunk for the Object heap that can expand or shrink
address OsMemory_allocate_chunk(size_t initial_size, size_t max_size,
                                size_t alignment);
// Flush the CPU instruction cache to ensure that the code generated by
// JIT is loaded from main memory when it is executed.
void OsMisc_flush_icache(address start, int size);
```

The first thing that is done when porting a VM is to build it on the platform and run a headless application – a single class with a single static main function that does not do much more than print to the console. After successfully implementing the porting layer and building the VM, we can run our first headless application. Cool. Now what's next? We need to add the Profile on top of the VM. From here things accelerate and it becomes more and more interesting.

As we said, the Java ME subsystem includes a Profile (e.g., MIDP 2.0) implemented by Symbian which is VM-agnostic so that licensees can use VMs from different VM vendors. We also mentioned an insulation layer between the Profile and VM proprietary interfaces. Let's take a look at this insulation layer, in relation to the CLDC-HI HotSpot VM: CLDC-HI defines the KNI interface which is designed to be a logical subset of JNI and provides functions for instance field access, to objects, strings, arrays, classes, interface operations, and so on. One of the key goals of KNI is to isolate the external code (e.g., Profile implementation code) from the internal implementation details of the CLDC-HI VM. However, while succeeding in doing that, the KNI is still VM-specific and is not portable across VMs. For example, KNI uses a register-based approach for retrieving method parameters from the stack frame. Use of

⁵ phoneme.dev.java.net/svn/phoneme/components/cldc/trunk/src/vm/os/

the KNI interfaces by the MIDP layer would create a direct dependency on CLDC-HI HotSpot.

To allow replacement of the VM, Symbian OS defines the KJNI which is a subset of the Java SE JNI. All native code that is required to perform such operations uses only the KJNI. Porting a new VM to Symbian OS requires implementing the KJNI in terms of the VM-specific interface. For example, the implementation of the KJNI for CLDC-HI uses the KNI. As a result, switching between VMs can be done almost seamlessly, from the Profile perspective.

Figure 10.6 depicts both the porting layer and the KJNI layer in the context of Symbian OS and the MIDP layer.

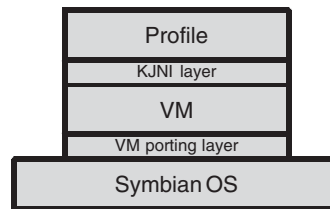


Figure 10.6 Porting and KJNI layers

VM technology requires quite a few tricks to create a hosted run-time environment on top of a native platform. Enumerating the tricks would be long; describing each of the mechanisms would require writing another book. There are VMs for other languages that do similar operations, such as interpreting instructions or managing memory, but Java technology took the theory and optimized it to the maximum. There is an order of a few magnitudes between the complexity of the Java VM and the complexity of VMs for languages such as Python or Ruby. So we now give a few examples of mechanisms in the VM.

10.4.1 VM Threading Model

The Java threading model can vary between different VM implementations. One model maps Java threads to native OS threads; another, the lightweight threading (LWT) model, implements threads entirely inside the VM. Each model has its own pros and cons. A native threading model means that every Java thread is scheduled by the operating system kernel with every running thread in the operating system. An LWT model implies much easier synchronization and is easily portable to many platforms. J9 uses a native threading model and CLDC-HI HotSpot uses an LWT model. As an example, we take a quick look into the LWT model in CLDC-HI.

At VM startup, the system classes are bootstrapped and a main Java thread object is allocated (see `VM::start()` and `Universe::bootstrap()`).⁶ After the main Java thread is initialized, its execution stack is set up with a first stack frame corresponding to the main method (see `Thread::initialize_main()`, `Thread::setup_lightweight_stack()` and `JVM::load_main_class()`).⁶ At that point, the VM is ready to start executing Java bytecode by calling `JVM::run()`. This is how CLDC-HI is initialized.

Let's take a short look at how the LWT emulates multiple Java threads on top of a single native OS thread (see Figure 10.7). The VM runs on a single native OS thread, named the primordial thread. During the execution of the primordial thread, either bytecodes in the context of a Java thread can be executed or no Java bytecodes are executed and the VM runs other internal required functionality.

The scheduling of Java threads is managed by the LWT Scheduler (see `src/vm/share/runtime/Scheduler.cpp`). For example, switching between Java threads is performed in `Scheduler::switch_thread()`. The switching between execution of the current Java thread and the primordial thread is achieved by two assembler code fragments in the interpreter: `current_thread_to_primordial` and `primordial_to_current_thread`. The code for those two assembler fragments is generated in `src/vm/cpu/arm/InterpreterStubs_arm.cpp`.



Figure 10.7 LWT model

10.4.2 Linkage of Native Operations

Although there is no JNI in CLDC, every Profile and its libraries still need to invoke native operations. Java SE (and CDC) has a JNI mechanism in which Java code can instruct the VM to load a native library and define Java methods that are implemented in C. In Java SE, this mechanism is dynamic – the loading of the library and lookup of the corresponding C function is done at run time.

CLDC-HI HotSpot takes a static approach more suitable to constrained mobile environments. The native invocation mechanism is available only to system classes and not to applications. System classes define the methods that are implemented in native code in a similar way but

⁶You will find the `JVM.cpp` and `Thread.cpp` files in the runtime folder at phoneme.dev.java.net/svn/phoneme/components/cldc/trunk/src/vm/share. The `Universe.cpp` file is in the `handles` folder.

are not required to instruct loading of libraries. At build time, tables that map Java native methods to their C counterparts are built and all the linkage is done statically. No dynamic lookup is performed at run time and only pre-built native functions can be invoked by the VM.

10.4.3 Garbage Collection and Internal Finalization

There are various relatively small mechanisms that are built as extensions to the supplied Configuration. Although small in size and complexity, they are nevertheless quite important for internal implementation or proper execution of the run-time process. For example, the Configuration should be extended with a mechanism that ensures that all native resources are freed before the owning Java object is reclaimed by the garbage collector (GC). An internal mechanism of `Object`, finalization, was introduced which ensures that native resources are still released even when application code has not properly released them. This is a simple mechanism in which instances of internal system classes register themselves for internal object finalization. When memory is reclaimed by the GC, every object that is not referenced any more and was registered for finalization can release native resources by executing an internal finalization method. This is a simple mechanism indeed, but highly important to free up acquired resources to allow other multitasking applications or system processes to acquire these resources.

10.5 MIDP Implementation Layer

The MIDP layer is implemented by Symbian OS and is integrated on top of the Configuration and the VM to provide a full run-time environment for Java applications.

Since the VM is a headless engine which can execute Java bytecode but has no notion of applications, the initialization of the Java ME run-time environment is performed at the MIDP layer. Upon launching of the VM by SystemAMS, it is given a class name whose static main function is the first Java code to be executed. The main class loaded into the Java ME subsystem is the system class that initializes the MIDP run-time environment. It instantiates all required system classes and resources, initializes them and, when the environment is finally ready, it triggers the lifecycle sequence of the launched MIDlet.

The MIDP layer comprises various MIDP components; each is typically a JSR implementation. A JSR implementation can be a pure Java implementation e.g., JSR-172 Web Services, or it can be a mix of Java code and native code which uses the native Symbian OS services. In the

latter case, such an MIDP component is usually divided into three main sub-layers:

- Java layer
- JNI layer
- Symbian C++ layer

This three sub-layer structure repeats itself in every JSR implementation that uses the native Symbian OS services. Let's look more closely at each sub-layer.

10.5.1 Java Layer

The Java layer contains the public Java APIs that are accessible to Java applications and a substantial amount of Java implementation code. Some of the operations are handled within the Java layer only, without going into the native code, and return immediately to the user application code. Other operations are required to invoke native Symbian OS services. This usually involves using the concept of native peer objects which invoke required native operations, on behalf of the Java object. The mapping to native operations is never straightforward (as a matter of fact, it is quite complex) and that requires that the Java layer does some processing before and after delegating operations to the native peers.

To remove all doubt: although MIDP applications cannot use native functions, system classes can do that by declaring a native function as in Java SE. For example, the following system class defines a native function called `_readBytes()` and passes three arguments to the native function implementation:

```
package com.symbian.midp;
class MyClass{
...
    private native int _readBytes(int aNativePeerHandle, byte[] aBytes,
                                  int aLength);
...
}
```

Once a native method is invoked from Java, the VM executes a statically linked native function, in the JNI layer. For example, the above method is mapped to a native function with the following signature:

```
JNIEXPORT jint JNICALL Java_com_symbian_midp_MyClass__readBytes
    (JNIEnv* aJni, jobject, jint aHandle, jbyteArray aBytes,
     jint aLength)
```

10.5.2 Crossing the JNI Layer

The JNI layer acts as a trampoline to the Symbian C++ code which performs the actual operations. When the execution reaches the JNI function, as defined in `MyClass.readBytes()`, the first obvious thing to do is to locate the native peer by casting the passed handle:

```
CMyClassPeer* peer = JavaUnhand<CMyClassPeer>(aHandle);
```

Afterwards, there are typically some preparatory operations that require invoking code in the VM itself (e.g., to access the `aBytes` parameter):

```
jbyte* javaBuffer = aJni->GetByteArrayElements(aBytes, NULL);
```

`GetByteArrayElements()` invokes operations directly in the VM itself. However, if you search for such a method in the reference CLDC-HI HotSpot, you will not find it. Let's see why. As we said previously, one of the areas in which the Java ME subsystem can reuse code is the VM implementation. Since there are no standard interfaces to VMs through which the Profile libraries can interact, a few VM-agnostic mechanisms were developed. The KVM Java Native Interface (KJNI) was defined to hide the native interfaces of different VMs. For example, `GetByteArrayElements()` is implemented by using CLDC-HI HotSpot interface functions, such as `KNI_GetArrayLength()`, `KNI_GetRawArrayRegion()`, and so on.

This VM-agnostic KJNI insulates the Profile libraries from differences between underlying VM implementations. The result is that Symbian OS licensees can replace only the VM engine, by providing an implementation of the KJNI.

After the native peer is located, the operation is delegated to the Symbian C++ layer. When the operation returns from the Symbian C++ layer, the KJNI can be used again to assign the return values into Java objects and primitives.

10.5.3 Symbian C++ Layer

The Java and JNI layers both sit on top of a native Symbian C++ layer which performs most of the functionality. The common functionality is built into the MIDP run-time executable and the variant functionality is built into various DLLs, which are loaded from the main executable.

Typically, when the first operation in a JSR that has a customizable DLL is invoked, the common code in the Symbian C++ layer does the following:

```
// load the DLL
User::LeaveIfError(iDLL.Load(KDLLName));
// look up the first ordinal function - the factory function
FactoryFunctionL = (TFactoryFunc)iDLL.Lookup(1);
// invoke the factory function to create a Factory instance
iFactory = (*FactoryFunctionL)();
```

The DLL is loaded and the first exported function of the DLL is invoked. That returns a main `Factory`, which is used to create objects that implement agreed interfaces that are called from the common code. The customized DLLs are provided by Symbian licensees.

One of the major responsibilities of the Symbian C++ layer is to integrate JSRs on top of the native Symbian OS APIs. However, any correlation between a Java ME API and its counterpart Symbian C++ API is only coincidental and minor. The two worlds are completely different in design, idioms and way of thinking. Still, Java ME on Symbian OS is tightly integrated with the native Symbian OS services. That implies that both main layers, the Java layer and Symbian C++ layer, include a large amount of code that decouples, adapts, abstracts and bridges between the two worlds. While the Java ME subsystem architecture can be generalized, the integration of JSRs on top of native Symbian OS is mostly done on a case-by-case basis. Each JSR brings its own challenges. Chapter 11 describes a few cases of integrating JSRs on top of the native Symbian OS services.

In the Symbian C++ layer, there are not only Symbian C++ classes that act as native peers to their Java object counterparts, there is also much code that builds up the internal architecture and frameworks. One of the main frameworks deals with handling asynchronous operations. One possible way to logically break up this layer is to distinguish between synchronous operations and asynchronous operations. The flow of the synchronous operations is that the operation starts in the Java layer, enters the JNI layer, goes down to the Symbian C++ layer and then goes all the way up again. Asynchronous operations do the same thing, with one fundamental exception – when they reach the native Symbian C++ layer, before going up again, they queue the asynchronous operation to be executed on another native thread.

To give an example, getting a file size has to use a synchronous method in the native Symbian C++ File API. The Java layer delegates the operation to the native Symbian C++ layer, via the JNI layer, and

the value is returned synchronously. On the other hand, reading bytes from a TCP connection requires the use of asynchronous methods in the native Symbian ESocket C++ API. Queuing of the asynchronous operation follows a similar route but the actual reading from the TCP connection is performed on a separate native thread.

10.6 Handling Asynchronous Symbian OS Operations

The handling of asynchronous operations is a major mechanism intrinsic to the Java ME subsystem. Just as a KJNI interface was defined to decouple libraries from a VM implementation, a similar argument holds for the handling of asynchronous events and operations, which is clearly VM-dependent, for example, whether the VM supports lightweight threads or not. An abstraction was needed to ensure that all libraries operate independently of the VM implementation.

Asynchronous operations and events in Symbian OS make use of the asynchronous active object idiom, which is inherent to Symbian OS design. The challenge is that handling asynchronous operations is not guaranteed to be identical or portable between different VM implementations. For example, the CLDC-HI HotSpot runs on a single primordial native thread on which all native operations must complete synchronously and never block. So again, a generic VM-agnostic solution was developed – Java Event Server (JES). A JES provides a native thread that can run an active scheduler and offers an inter-thread communication mechanism to simplify interaction between the main thread and the JES thread. There are multiple JES instances running, but the mechanism as a whole operates as if there is a single Java Event Server. That solves the first issue, in which all operations must complete synchronously and never block – the only operation done on the VM thread is to queue a request for the asynchronous operation that is executed later, on the JES thread. Then come two other challenges: the Java operation might be synchronous; and there is a need to report the response back to Java code when the asynchronous operation is complete.

Dealing with the first challenge is fairly straightforward. The request for the native asynchronous operation is queued on the native VM thread and returns immediately. When the call returns to Java, the current Java thread is put to sleep. Who wakes it up and when brings us to the second challenge – reporting the response back to the Java code when the asynchronous operation is complete.

For the second challenge, another VM-agnostic solution was designed. The idea is that the MIDP run-time environment has an internal Java thread which carries notification about the completion of an asynchronous operation. Let's see an example of how this is done in code.

First, queue the asynchronous operation and let the Java thread on which the operation is invoked go to sleep:

```
synchronized (this){
    // queue the async operation
    iError = _callSomeAsyncOperation(iNativePeerHandle);
    if(iError == NO_ERROR){
        try{
            // put the current Java thread to sleep until operation finishes
            iError = PENDING;
            wait(MAX_TIMEOUT);
        }
        catch (InterruptedException ex1) {
        }
    }
}
if (iError == NO_ERROR){
    ... // async operation completed. continue normal flow
}
else{
    ... // async operation failed. handle error
}
```

Secondly, the native asynchronous operation is performed on a native JES thread. (This involves a large amount of code so we do not provide example code.)

Thirdly, when the operation is complete, the Java notification thread invokes a method in the Java object to pass the response data and wakes up the sleeping Java thread:

```
public void notifyBlockingOperationCompleted(int aError){
    synchronized (this){
        iError = aError;
        notify();
    }
}
```

After `notify()` is invoked, the sleeping Java thread wakes up and continues execution from the line next to the call to `wait()`, to handle the operation completion. You probably noticed that there is some magic which triggers `notifyBlockingOperationCompleted()` to be called. How does the Java notification thread know when the operation is complete?

Under the hood, upon completion of the asynchronous operation, the JES puts a Java stack frame on the notification thread stack and wakes up the notification thread (all that is done in native code with direct access to the Java thread stack). When the notification thread wakes up, the first thing it does is the next operation in its stack, which was queued by the JES.

In the layout of the run-time stack of the CLDC-HI HotSpot VM, there are three basic forms of stack frame that are constructed on the Java

stack of a given Java thread in support of application execution: entry activation frames, method activation frames and calls to VM frames. The JES achieves the delayed execution of Java methods by adding a pending entry activation frame to the notification thread stack.

10.7 Java-level Debugging Support

Providing a Java ME SDK is left for Symbian OS licensees who each have their own SDK strategy. It is also the responsibility of the licensees to arrange, for example, the setup and instrumentation of the connection between the host PC and the device.

The responsibility of Symbian OS is to ensure that a complete debugging solution exists for Java applications; it includes debugging applications on the emulator and on target devices by performing debugging operations through a standard Java IDE.

Let's give a brief overview of the core debugging support. In CLDC, the protocol between the VM and the IDE is a subset of Java Debug Wire Protocol (JDWP) called KVM Debug Wire Protocol (KDWP). The KDWP packets go through a proxy on the PC called KDP-Proxy which converts between JDWP and KDWP (see Figure 10.8). All debug operations, such as single stepping, reaching breakpoints or watching variables, are carried on top of this protocol.



Figure 10.8 Debugging functionality

When CLDC-HI HotSpot runs in debug mode, it has an internal state machine that replaces the desktop VM JDI. That state-machine module is responsible for reading KDWP commands, executing them and replying back to the IDE.

To provide a complete debugging solution, another two entities are required: an implementation of the Unified Emulator Interface (UEI) must be integrated with the interactive development environment (IDE) on the PC; and an on-device Debug Agent manages the deployment and lifecycle of applications being debugged.

The deployment and lifecycle operate on a different channel and domain from the raw debugging functionality. The Debug Agent is a native Symbian C++ application that is shipped with device manufacturer SDKs, such as the S60 3rd Edition SDK or Sony Ericsson SJP-3 SDK. The Debug Agent receives requests from the UEI to install, launch and terminate over a bearer, such as Bluetooth, and uses the client APIs of the Installer and

SystemAMS to delegate the operations. For example, when the developer launches a MIDlet from the IDE, the UEI implementation connects to the Debug Agent, which instructs the Installer to install the suite and the SystemAMS to start the MIDlet in debug mode (see Figure 10.9).

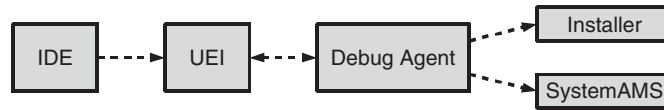


Figure 10.9 Deployment and lifecycle management functionality

Other operations such as Java applications management, `System.out` print redirection, and memory profiling can all be implemented with the same mechanism.

10.8 Performance

The most common criticism leveled at Java ME (or any version of Java for that matter) is that it doesn't execute as fast as native code because it's an interpreted language. But that stopped being a strong argument in the days of Symbian OS v7, which introduced the Monty VM that came with a Just-In-Time (JIT) compiler. The JIT selectively compiles the hot areas of the application into ARM instructions onto a user writable RChunk that is marked by the Kernel as containing code and is private to the VM process only. The JIT boosts the performance significantly and provides an improvement that can be both objectively measured and perceived by the user.

Only the methods that execute most frequently are compiled; other methods are interpreted by the virtual machine. For example, if an application has a method which is called several times, at some point the method is queued for compilation. Afterwards, whenever the Java method is called, the dynamically ARM-compiled code is executed instead. If the method stops being called, or other methods are being called more frequently, then at some point the VM may decide to free up the memory that contains the compiled code in order to use it to compile other Java methods. Then the Java bytecode of the method is again executed whenever the Java method is called.

Not only can application code be compiled, system classes can also be compiled. That can be done at product build time to save the CPU processing time that is required for compilation. For example, in CLDC-HI HotSpot, this process is called 'system classes ROMization', and is done during the product build by using a Win32 build of the VM which runs the JIT module itself.

Additionally, even the interpreter loop which reads and executes Java bytecode and is one of the hot spots of the VM execution can be heavily optimized by implementing the loop in ARM code. For example, the CLDC-HI HotSpot interpreter loop is made of highly optimized ARM code that is auto-generated by the JIT module at product build time.

Naturally, run-time speed optimization comes at the cost of use of memory. The compiled ARM code is bigger in size than the lightweight Java bytecode and occupies more space. However, Symbian OS devices come with sufficient memory to allow this overhead in favor of having an optimized run-time speed.

We previously discussed how IPC and JES are used in the Java ME subsystem. This is an area which involves threads and process-context switching done by the Symbian OS kernel. To minimize the overhead, each operation that involves context switching strives to be coherent but also to pass additional information if that can save another context switch later on. Therefore, OEM engineers who work inside the Java ME subsystem should design those switches to be as efficient as possible.

Other performance considerations are always considered on a case-by-case basis for each newly developed JSR. In the case of the UI, this is a mission-critical operation which is carefully considered. The LDCUI support was handcrafted to ensure maximal performance and other JSRs (e.g., JSR-184 3D and JSR-226 SVG) can also take advantage of any graphics hardware acceleration present on the device.

10.9 Security

Ensuring security on mobiles is one of the highest priority items in every design checklist. In the Java ME subsystem, security is divided into two key areas: the MIDP security model and the Symbian OS security model.

Please note that there is no direct relation between the two. The MIDP security model is completely orthogonal to the Symbian OS Platform Security model. Both the SystemAMS and the Java ME run-time environment are subject to platform security and must be assigned with platform security capabilities in order to allow their internal logic execution or to allow JSR implementations to integrate with sensitive native operations. Allowing Java applications to access restricted Java operations is based on Java security. MIDP Security is an internal module in the Java ME subsystem and handling of all Java restricted operations is done separately, with no relation to platform security.

The MIDP implementation does not implement a default security policy. It provides a customizable security framework which requires a number of modules to be provided by the device manufacturer, which together define the actual security policy. Symbian OS licensees and

operators can customize the policy according to their Java security specification, for example, to define different protection domains.

Another important issue in the Java ME subsystem is the IPC servers' security. As we discussed, the Debug Agent and the Symbian OS Installer and Recognizer access the SystemAMS through client APIs of the internal IPC servers. Similarly, the MIDP run-time environment communicates with the SystemAMS over IPC. Exposing an IPC client API is always a potential security risk which must be mitigated. Therefore, the use of the SystemAMS IPC servers is not open to third-party applications or to system processes other than those defined. Access to the SystemAMS Installer is allowed only from the Software Installer process; access to the SystemAMS Launcher is allowed only from the system process that runs Recognizers and from the Debug Agent. The main EXEs of the SystemAMS and the run-time environment are protected as well, to ensure only authorized parent processes can launch them.

10.10 Summary

In this chapter, we looked at the Java ME run-time environment from the perspective of Symbian OS. We broke the monolithic term 'Java ME platform' into the different processes and binaries which interact in different ways. We then looked at core areas in the SystemAMS, the VM and the MIDP layer. Finally, we discussed support for asynchronous operations, Java debugging, performance and security.

One chapter cannot cover the whole of the Java ME subsystem architecture or VM technology but it should give you a clearer understanding of the main entities and how things work under the hood.

So what are the four mistakes in the sentence "The KVM aborted the MIDlet installation"?

1. The VM is a headless engine which can only run Java bytecode.
2. A VM can certainly not install anything. Only the AMS can do that.
3. In Symbian OS, installation is performed by the system Installer in cooperation with the SystemAMS.
4. There is no KVM on current Symbian OS devices!

The next chapter covers JSRs mapped to native Symbian OS services.

11

Integration of Java ME and Native APIs

In Chapter 10, we discussed the architecture of the Java ME subsystem on Symbian OS – the subsystem binaries, the Java Application Management System (AMS), the CLDC-HI virtual machine (VM), interaction between the VM and AMS and the integration with Symbian OS architecture, platform security and Symbian C++ programming idioms.

Now that the monolithic and so-very-abstract entity of the Java ME subsystem on Symbian OS has been broken into smaller and more concrete entities, and the interaction between them has been explained, a question that should be asked is – when an application invokes a given Java API method, does it go outside the Java ME subsystem into the native Symbian OS services? Or is it handled internally and independently, in the Java ME subsystem only? In more technical terms, how do Java ME APIs map to the native Symbian OS APIs?

For example, given a Java ME API, such as JSR-135 Mobile Media API (MMAPI), is the multimedia content played by the Java ME subsystem itself? Or does it use the native multimedia subsystem and delegate the playing and recording of content to the native Multimedia Framework (MMF)? Another example is the Generic Connection Framework (GCF) – does the Java ME subsystem have its own HTTP implementation? Or does it use the native Symbian OS Application Protocols subsystem, which contains components for using the HTTP protocol, and associated utilities? Assuming that the Java ME subsystem makes use of the native services – how does the Java ME subsystem integrate with the native Symbian OS Multimedia Framework or with the Application Protocols subsystem?

But even more important is the following question – if there is integration, why is it needed and what purpose does it serve? All those challenging questions bring us to the topic of this chapter, which explores the options, considerations and decisions in regards to Java ME JSR integration with native Symbian OS services.

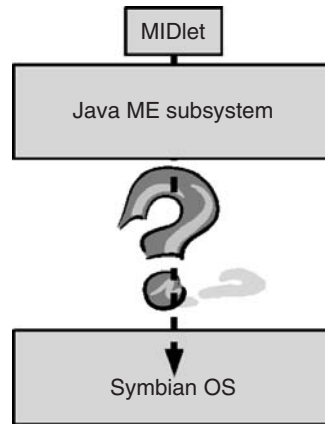


Figure 11.1 Do JSRs integrate with native Symbian OS services?

We start the discussion by providing a definition of different levels of integration, when each type of integration is applied and how it is applied. Second, we enumerate several Java ME JSRs¹ by their type of integration with a relevant native Symbian OS service,² and discuss each JSR and its integration separately. In each example JSR, we provide an overview of the Java ME JSR that sits on one end, then briefly discuss the relevant native Symbian OS service that sits at the other end. The discussion then shifts to what is between those two ends – the binding of the two worlds. We provide an overview of the internal architecture of the JSR implementation focusing specifically on the mechanisms and layers of integration of the Java ME JSR with the underlying native Symbian OS service. We explain the pros and cons of the chosen integration approach and attempt to challenge the approach by suggesting a different type of integration for the JSR and evaluate how this alternative integration approach would impact Java ME applications running on Symbian OS devices.

11.1 Importance of Integration with Symbian OS Services

To understand the importance of integrating Java ME JSRs with the native Symbian OS services, we need to re-examine the uniqueness of Symbian OS compared to other mobile embedded operating systems.

¹ To find the Java ME APIs that are discussed in this chapter, please refer to the JCP at jcp.org.

² To find the native Symbian OS APIs that are discussed in this chapter, please refer to the Symbian Developer Library at developer.symbian.com/main/documentation/sdl.

Symbian OS is an open platform which allows installation of after-market applications. Symbian OS devices can be customized by licensees to be shipped with a large number of pre-integrated applications which must interoperate and provide a consistent user experience. Symbian OS is a multitasking operating system that can run more than one application at the same time.

If the Java ME subsystem were not integrated with the native services, the user experience of after-market applications would be different from one application to another and also from pre-integrated applications. Specifically, we would have noticed the following fragmentation:

- From a user-interface (UI) perspective: Java applications would look different to the native UI platform (e.g., different background and foreground colors, different skins, and different locations of soft buttons)
- From a multimedia perspective: Playing and recording of multimedia formats and content would not have uniform support for Java and native applications
- From a phone-usage perspective: Java applications would not enjoy the rich usability features of native applications (e.g., task manager, touch-screen support, support for multiple languages)
- From a multitasking perspective: Java applications might have problems competing for system resources while running at the same time as native applications
- From a developer perspective: some of the application-level protocols would behave differently in Java applications compared to native applications (and that could possibly impact the features of Java applications using the specific application-level protocol).

More examples could be given and, in many low-end mobile platforms, it is indeed the case that, due to the lack of native services, the Java ME platform comes with an internal implementation for some JSRs (in low-end devices, Java is sometimes the only downloadable applications execution environment and therefore there is no other alternative: you either use Java ME or only pre-installed applications).

However, having an application execution environment showing different behavior to native applications is not acceptable in a high-end mobile platform with high standards of interoperability, such as Symbian OS. The strategic approach, from the early days of Java on Symbian OS and to provide consistent application execution environments and user experience, is to tightly integrate the Java ME APIs with native Symbian OS services, which ensures that high standards of interoperability are maintained.

11.2 Types and Levels of Integration

System integration is the bringing together of different subsystems, which are not necessarily heterogeneous, into one unified system that has a uniform behavior and functionality. In a system that hosts multiple application execution environments and is tightly integrated, the same functionality in two applications running in different execution environments should not be different from each other. Each of the execution environments would internally use the same underlying services, which ensures that the functionality and behavior across the different environments is uniform. Ideally, an end user would not be able to distinguish between applications that run in different execution environments.

Using this definition of system integration, we can deduce that the integration of the Java ME subsystem JSRs onto Symbian OS native services is meant to provide a consistent experience to end users. This goal is achieved by making public Java ME APIs to produce a unified set of functionality and behavior with native applications, by integration with the native Symbian OS services.

At one extreme, a JSR does not need to collaborate with native Symbian OS services and thus avoids the whole integration issue entirely. However, such an integration approach is feasible or acceptable only for a very small number of JSRs. At the other extreme, for each Java method, we call an equivalent Symbian C++ method. Such a one-to-one mapping cannot be considered realistic due to the differences in concepts, architectures, and idioms between Java ME and Symbian OS. Between the extremes of those two integration approaches, there are other approaches for integrating JSRs. Each integration approach addresses some of the JSR and integration requirements in a different way. The more likely approach is that integration is driven by JSR functionality, rather than by mapping Java classes to native classes.

A specific JSR is integrated with the relevant Symbian OS subsystem in a way that ensures that the functionality and use cases are handled using native primitives. In other words, the correct view of an integration approach is not to map the JSR classes to native classes but to map the JSR functionality to native functionality using the native classes.

In the case of a network protocol, if the run-time environment uses the underlying native services that support this network protocol, we can say that the run-time environment's support for the network protocol is *tightly* integrated to the system. If native APIs are not available for a given network protocol, the run-time environment is not necessarily prevented from supporting the network protocol and exposing it to hosted applications. For example, in the case of an application-level protocol,

such as HTTP, the run-time environment can provide its own HTTP stack on top of the native TCP support. If we take a few steps down in the network protocols tree (e.g., to the level of TCP), it is safe to assume that the run-time environment either uses the native support or does not support the network protocol at all since, at this level, access to the native services is mandatory.

Another example of different integration levels is in the UI where high-level functionality (e.g., SVG graphics) may use the underlying SVG graphics support but, if there is no native support, the run-time environment can provide its own SVG graphics implementation on top of low-level rendering (or choose not to support it at all). The run-time environment widgets may map onto the native system widgets or there could be an internal implementation of widgets on top of low-level rendering. If we take a step down the UI levels to low-level rendering, it is clear that, at a minimum, the run-time environment must use some native low-level rendering mechanism which provides access to the screen.

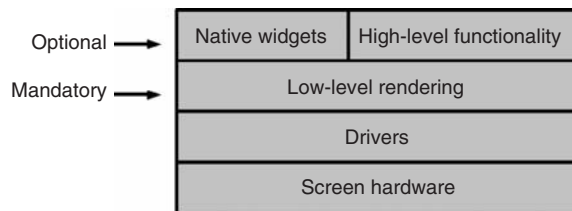


Figure 11.2 Levels of integration in the user interface

11.3 Integration Challenges, Costs and Considerations

Let us now discuss the reality of Java ME subsystem integration – the challenges that JSR integration engineers face and the considerations in preferring one type of integration over another, or providing a non-integrated JSR implementation.

There are numerous challenges in the interoperability of non-heterogeneous software components:

- different languages – Java, C and C++
- different platform idioms – Java and Symbian C++
- different architectures – Java ME architecture and the Java ME subsystem internal architecture compared to the Symbian OS architecture.

There are also challenges specific to each JSR, as we see in the examples given in the rest of the chapter. The technical challenges require a non-trivial engineering effort.

To assess the benefit of providing a tightly integrated Java ME subsystem, we need to take into consideration various factors.

- **Java strategy:** As a guideline, Symbian has chosen to favor the option of providing tightly integrated JSRs in order to ensure unified behavior and functionality across Java and native applications. This is an important guideline (although it is not a rule).
- **Strictly mandatory:** Integration is strictly mandatory if there is no way to provide the functionality without using the native APIs (e.g., the native Symbian OS services are fundamental for providing TCP support).
- **Customizability:** Integration may require extra effort from Symbian OS licensees to customize or extend the JSR implementation, while integrating Symbian OS onto their UI platform.
- **Consistency:** A non-integrated JSR may break the consistent behavior and functionality of the Symbian OS system and have an impact on the end-user experience.
- **Performance:** An integrated JSR implementation may provide better performance. If it does, you must consider the scale of the improvement and whether the improved performance justifies the integration cost.

If there is no equivalent native API for a given JSR or the cost of integration was deemed to exceed the benefit, the JSR can still adhere to considerations of consistency and the general Java strategy: a pure Java JSR implementation could be a fit-for-purpose solution. We discuss one such case in Section 11.5.2.

If consistency and customizability by Symbian OS licensees is critical and performance is of high importance, a tightly integrated JSR may be chosen even if the cost of integration effort is very high. We discuss such a case in Section 11.5.3.

The question then becomes, which integration approach best addresses each of the above criteria?

11.4 Determining the Right Integration Style

What makes for good JSR functionality integration? If the integration needs of Java ME JSR functionality were the same, we could have used a single style of integration for each JSR. But, in reality, JSR functionality integration involves a vast range of considerations and impacts that should be weighed before adding each JSR into the Java ME subsystem.

The trick is not to choose one integration approach to use always, but to choose the best approach for a particular JSR integration after considering its advantages and disadvantages.

In case the idea of some automatic Java-to-native-code generator or any other magic tool that binds the Java world to the native world springs to mind, such a tool does not exist and was never considered as a solution that would be able to handle all the requirements and generate a fit-for-purpose integration of JSRs.

The solutions to the JSR integration challenges were developed by applying generic software-engineering solutions, such as applying relevant structural design patterns (e.g., the Adapter, Bridge, Proxy, and Façade patterns) in the places where they fit and developing idiomatic solutions to idiomatic challenges. For example, the Java Event Server (JES) that was described in Chapter 10 is used heavily in every JSR that integrates with asynchronous Symbian C++ APIs. Most of all, it was neither technology nor patterns that did the trick but the hard work and creativity of a team of expert engineers – the Java group in Symbian Ltd.

11.5 Examples of JSR Integration

Let us now discuss various Java ME JSRs and their integration approach.

We start with two cases of integration at opposite extremes: integration is mandatory for JSR-75 `FileConnection` and there is no integration at all for JSR-172 `Web Services`, which uses a pure Java implementation. With those two JSRs, we learn that opposing integration approaches can coexist in the Java ME subsystem and we also get a first glance into how a Java ME API is integrated onto a native Symbian OS API.

We then consider the integration of the most well-used APIs: JSR-118 `MIDP 2.0 LCDUI API`, JSR-135 `MMAPI` and JSR-118 `MIDP 2.0 Media API`. UI and multimedia are probably the most common features of every mobile Java application. But apart from the importance of these APIs, they are interesting and challenging integration cases.

We then look at the integration of JSR-248 `MSA Components`, JSR-177 `SATSA APDU` and JSR-180 `SIP`. JSR-177 and JSR-180 are newer JSRs that are part of JSR-248 `MSA full`. Their integration provides an interesting insight into how different the Java and native worlds can be, yet integration must still be delivered.

Please note that the focus of this discussion is the integration of the JSRs and not the JSRs themselves.

11.5.1 Mandatory Integration: JSR-75 `FileConnection`

The `FileConnection` API is an optional package of JSR-75³ that facilitates access to the device file system and removable media (e.g., memory

³ JSR-75 `FileConnection` implementation is under `\src\common\generic\j2me\components\file` in the Symbian OS source repository.

cards). It supports the creation and removal of directories and files, IO operations on files, listing directory contents and getting file information. The two most important classes are:

- **FileSystemRegistry**: a central registry for mounted roots and for registering file-system listeners interested in receiving notifications on the addition and removal of file systems at run time
- **FileConnection**: an interface used to access files and directories on the device. It extends the Generic Connection Framework (GCF) **Connection** interface and provides the majority of the optional package functionality. Each **FileConnection** corresponds to a single file or directory at a time. Some methods can be used on both files or directories (e.g., file attributes), while other methods are only usable with either a file or a directory.

Applicable Native Services

The Symbian OS File API has a few key classes:

- The **RFs** class encapsulates a file server session and provides file system manipulation functions, such as creating, deleting, moving and renaming files and directories; manipulating file and directory attributes; and requesting file system change notifications.
- The **RFile** class encapsulates a file that can be opened, created, or replaced.

Other classes, which are less relevant to a discussion of JSR integration, encapsulate drives and volumes, efficient large-scale file management, file searching and directory scanning.

Integration onto Native Services

Before we discuss the integration of the JSR-75 **FileConnection** to the Symbian OS file handling service, we should clarify the reason why it was chosen to be the first example.

The integration of the API was relatively straightforward since the native **RFs** and **RFile** classes provide methods that match the **FileConnection** API sufficiently. This is an example of an integration that is as close as possible to the one-to-one integration type. However, even JSR-75 **FileConnection** is not at the extreme end of the one-to-one integration type and we discuss that as well.

We now present a few examples of how **FileConnection** methods map to native functionality. **FileConnection.create()** creates a new

file corresponding to the file URL provided in `Connector.open()`. There is some level of correlation between the two APIs and so a call to `FileConnection.create()` eventually invokes a simple native call to `RFile::Create(RFs, name, file-mode)`. In a similar way, a call to `FileConnection.mkdir()` eventually invokes a simple native call to `RFs::MkDir(path)`. Deleting a file by calling `FileConnection.delete()` is eventually delegated to a native call to either `RFs.Rmdir(path)` or `RFs.Delete(path)` depending on whether the URI points to a directory or a file, respectively.

The `FileConnection` methods `canRead()`, `isHidden()`, and `isDirectory()`, used to get the readable, hidden and directory attributes, respectively, are performed by first calling `RFs::Entry(name, TEntry)` and then calling `TEntry::IsReadOnly()`, `TEntry::IsHidden()` or `TEntry::IsDir()`. Setting file attributes, when applicable, uses `RFs.SetAtt()` which sets or clears the attributes of a single file or directory.

`FileSystemListener.rootChanged(int state, String rootName)` is invoked when a root on the device has changed state (for example, a file system root has been added to or removed from the device). To receive such events, the JSR-75 `FileConnection` implementation registers for native file-system events using `RFs::NotifyChange(TNotifyType, TRequestStatus, path)` and pushes the knowledge of those events to Java `FileSystemListeners` after translating the native event data structure to Java primitive types.

As can be seen from these examples, Java functionality has to be mapped to native functionality and in JSR-75 `FileConnection`, it is fairly straightforward; deleting files or directories and getting and setting file attributes map closely to the native RFs API.

However, in reality there are more challenges than just mapping functionality. In Chapter 10, we discussed how the Java ME run-time environment is subdivided into several layers (see Sections 10.5 and 10.6). We now use JSR-75 to provide a concrete and detailed example of those layers.

First, the public Java API is implemented by system Java classes that are not accessible to Java applications. Those Java classes process information themselves or delegate it to native code as applicable.

Secondly, jumping from Java code into native code requires a layer of JNI-like C functions that act as a trampoline from Java to Symbian C++ code. That layer uses the VM-agnostic KJNI API (see Section 10.4).

Thirdly, the insulation of the VM threading model and Symbian OS asynchronous operations requires an implementation layer of code that runs in the native Java Event Server (JES) thread and native code that delegates the execution from the VM thread to the JES thread. This layer

also contains support for notification of the completion of asynchronous operations back to the Java code.

So the layer in which the native services client usage resides (e.g., explicit usage of RFs and RFile APIs) is reached only after the code execution crosses the Java, JNI and JES layers. This pattern of layers is a recurring pattern in most JSR implementations (see Figure 11.3). We chose to present it first with JSR-75 FileConnection as it is easier to visualize this layer model with a fairly straightforward JSR.

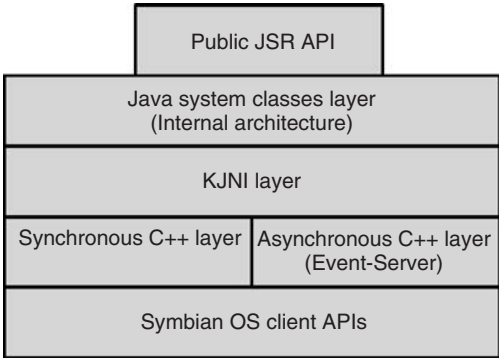


Figure 11.3 Recurring layers in JSR implementations

Conclusion

JSR-75 FileConnection has to be integrated with native Symbian OS services as there is no other way to facilitate access to the file system. Here we have seen a case of a straightforward integration where the Java API maps well to the native Symbian OS service. We have also shown the various layers of integration that are a recurring pattern in the integration of most JSRs.

11.5.2 No Integration: JSR-172 Web Services

Strategically, integration with native Symbian OS services is the first option when designing a new JSR implementation for the Java ME subsystem on Symbian OS. However, there are cases where integration with native services is neither applicable nor serves any requirement of Java applications or the Java ME subsystem. We discuss such a case with reference to JSR-172 Web Services.

A web service is a collection of APIs that can be accessed over the Internet, to be executed on a remote system hosting the requested services. Web services use XML to exchange data over standardized, open protocols to ensure that the web services are interoperable and platform-neutral

to enable servers, desktops and mobile devices to exchange data while removing the complexities that are usually involved when different applications run on different devices or operating systems.

The Web Services specification, JSR-172, provides standard Java APIs that make it easy for a smart Java ME client application to parse XML files (it supports SAX 2.0, XML namespaces, UTF-8 and UTF-16 encoding, etc.), use JAX-RPC client-side APIs to discover and invoke hosted web services, use secure messaging over HTTP or HTTPS, and more.

On Symbian OS devices, Java ME developers can use JSR-172 to discover and exchange data with any number of servers, to leverage existing public web services (e.g., Amazon, eBay, and Google) and easily create new applications that are interoperable with many web services implementations.

Applicable Native Services

Symbian OS provides powerful APIs for all the technologies required to build web services client applications (e.g., HTTP, XML, etc.). However, these native APIs are not grouped together to support web services as a single framework, in the same way that JSR-172 puts XML, HTTP and RPC into the same bucket. An implementation of JSR-172 could provide its own internal web services engine implemented entirely in native code but that would only be one possible combination of various native Symbian OS subsystems, rather than integration with a single Symbian OS component that ensures functional consistency. (In theory, if a UI platform based on Symbian OS introduced its own web services subsystem, the manufacturer could provide its own integrated JSR-172 implementation.)

Integration with Native Services

This is an example of a case where a pure Java implementation was favored. Java packages were implemented in pure Java and integrated with the Java layer of the Java ME subsystem. None of the classes in these Java packages have any direct calls to native functions. Therefore, in this case we do not have any native architecture to discuss nor integration challenges to present.

Pros and Cons of the Integration Approach

The decision to choose a pure Java implementation for JSR-172 came after considering the following factors:

- Java strategy – In this case, it was a legitimate decision to take the pure Java approach.

- Strictly mandatory – JSR-172 provides high-level support which does not mandate integration with any native service.
- Customizability – Customization of web services is not part of the integration by Symbian OS licensees onto their UI platform. (The issue of customization by Symbian OS licensees will become clearer in Section 11.5.3.)
- Consistency – Since the native Symbian OS APIs that support the technologies required for building a web services client application are not grouped together as a single framework, consistency is not broken and functionality is not fragmented between client applications that use native services and Java applications that use the pure Java implementation. In both native and Java web-services applications, the behavior and data handling is always application-specific.
- Performance – Assuming that mobile applications should refrain from parsing large XML files, most of the time spent in web services scenarios is during the transport of data from one end to another, over the air, rather than in processing data on the device. Therefore, an optimized Java implementation is sufficient to provide fit-for-purpose performance.

Now it is time to mention that integration does not come without costs and the glue between the Java ME subsystem and Symbian OS has its own overhead in terms of dynamic and static resources. Therefore, such native integration would add considerable effort and cost but would not give any added value to Java applications.

Having said all that, at the core of web services, there is implicit integration with native Symbian OS services! The JSR-172 implementation is indeed all implemented in Java but it uses other public Java ME APIs, which are tightly integrated with native Symbian OS. For example, JSR-172 uses `javax.microedition.io.HttpConnection`, which uses the native HTTP framework that is used by every native HTTP client application.

Conclusion

JSR-172 is a case where no integration was necessary or required to achieve a fit-for-purpose solution. However, this is the exception and not the common case.

11.5.3 Tight Integration with Customization: LCDUI

LCDUI is logically composed of high-level and low-level APIs. The `javax.microedition.lcdui.Screen` class takes care of all user interaction with high-level user interface components. The various

Screen subclasses handle rendering, interaction, traversal, and scrolling, with only higher-level events being passed on to the application and very little control over look and feel. The actual drawing to the display is performed internally by the Java ME implementation and the MIDP application cannot define the visual appearance. User interaction, such as navigation or scrolling, and other operations are also handled internally by the Java ME implementation. When using the high-level API, it is assumed that the underlying implementation does the necessary adaptation to the device's hardware and native UI style.

The low-level API classes (`Canvas` and `Graphics`) provide very little abstraction but allow the application to control precise placement of graphic elements, as well as access to low-level input events. Applications that program to the low-level API are not guaranteed to be portable, since the precise control means accessing details that are specific to a particular device.

Applicable Native Services

The Symbian OS architecture provides highly powerful UI and graphics capabilities which are encapsulated in two major Symbian OS subsystems: the Graphics subsystem and the Application Framework (APPARC) subsystem.

The Graphics subsystem governs the functionality for drawing to the screen, embedding picture objects, and font and bitmap handling. The subsystem also contains two other sets of APIs: the Window Server Client-Side API, which provides functionality to draw to windows and receive window events, and the animation APIs, which allow animations to be run in a high-priority system thread.

The APPARC subsystem includes, as the name implies, the frameworks that define the application structure and their basic user interface handling. 'Application' here has a slightly more focused meaning than just a software program: it implies a program with a user interface, rather than a UI-less program (e.g., a background process doing a system task). APPARC also includes reusable frameworks for handling UI controls, text layout and non-keyboard input mechanisms.

Integration with Native Services

It was clear from the beginning of Java ME on Symbian OS that LCDUI widgets must be mapped to native controls and rendering must use the Graphics subsystem frameworks. In addition, the tight integration with the Graphics subsystem and APPARC had to take into consideration the mandatory requirement for customization by Symbian licensees.

The LCDUI integration with Graphics and APPARC can basically be characterized into two functionality areas: common functionality inside a

‘core framework’ and the ‘customizable implementation’ which contains all dependencies on the native UI toolkit of the Symbian UI platform.

The Java ME subsystem’s LCDUI implementation can be decomposed into three major parts: the LCDUI core framework,⁴ a UI widgets module,⁵ and a graphics-rendering module⁶ (see Figure 11.4).

The LCDUI core framework contains Java and native code and is responsible for loading and invoking methods in the two extension modules using a set of adaptor interfaces. The widgets module and the graphics module provide a concrete implementation for the set of adaptor interfaces. All code that requires customization by Symbian OS licensees is located in the extension modules, which allows licensees to make highly localized changes. The LCDUI core framework refers to the concrete components solely through the adaptor interfaces and knows nothing about their implementation.

The UI widgets module contains native code only and comes as a DLL providing a heavyweight widgets implementation over the native UI toolkit, which is well-integrated with the native Application Framework. Symbian OS licensees must provide a concrete implementation for several interfaces each of which acts as an adaptor for a given LCDUI widget.

The licensee’s DLL exports a single widgets factory class (called `MMIDComponentFactory`), which creates and returns concrete implementation instances for all the adaptor interfaces. Typically, these concrete implementation objects delegate the operations to the UI platform’s APPARC extension classes, which makes the LCDUI widgets look and behave exactly the same as the native UI widgets.

Each Java LCDUI object generally has an equivalent peer native object, to which it holds a reference as a handle. The handle is created in native code, by the Factory exported by the widgets DLL, and returned to the Java

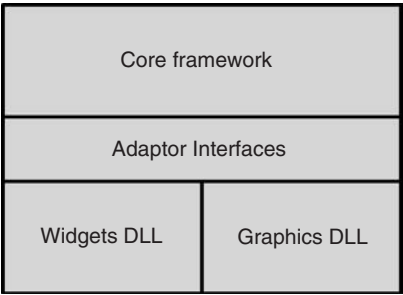


Figure 11.4 LCDUI implementation of the Java ME subsystem

⁴ LCDUI core framework is under `\src\common\generic\j2me\components\lcduib` in the Symbian OS source repository.

⁵ The widgets plugin is under `\src\common\generic\j2me\plugins\reflclui` in the Symbian OS source repository.

⁶ The graphics-rendering plugin is under `\src\common\generic\j2me\plugins\lcdgr` in the Symbian OS source repository.

object. The base class of all LCDUI widget peers is an interface called `MMIDComponent`, which provides some generic methods common to all native LCDUI peers. Most of the other adaptor interfaces build on `MMIDComponent` and typically map to the LCDUI classes.

For example, when a MIDlet creates a UI component, such as an `Alert`, a native method in the core framework calls `MMIDComponentFactory::CreateAlertL()`, which creates a concrete peer from the widgets DLL. This returns an `MMIDAlert` pointer which is stored in the Java object. When the user invokes a method, such as `Alert.setString()`, the handle can then be passed into native method calls, where it is converted back into a pointer to the native `MMIDAlert` and the required native operation is performed in the widgets DLL. The same pattern is repeated for other LCDUI widgets (i.e. the peers for `Command` and `List` in LCDUI are the native `MMIDCommand` and `MMIDList`, respectively). The implementation class must satisfy the adaptor interface contract but internally may do so using a single `CCoeControl` or a set of `CCoeControls` as required and as applicable in each UI platform.

The graphics-rendering module can be replaced or modified if desired by the licensee, but typically is left unchanged since graphics-rendering operations do not impose any dependency on the UI platform and use common and generic Symbian OS rendering APIs. For example, when `Canvas.paint()` is called, the user code draws on the passed `Graphics` object by invoking the rendering methods (e.g., `Graphics.drawRect(int x, int y, int width, int height)`). All the information required to perform the rendering is serialized and packed as buffered commands into an efficient command buffer which is then flushed to a native buffer to do native processing. In the native layer, the execution moves from the core framework to the graphics DLL, where the commands processor of the `MMIDComponent` executes all the buffered operations using the native Symbian OS rendering primitives (e.g., `Graphic subsystem Device, Context, Bitmaps`, etc.).

The Game API package provides a series of classes that enable the development of rich gaming content for wireless devices and minimizes the amount of work done in Java applications. The API uses the standard low-level graphics classes from MIDP (`Graphics`, `Image`, etc.) so that the high-level Game API classes can be used in conjunction with graphics primitives.

In Symbian OS v9.1, the Java layer in the core framework and the graphics DLL have been substantially redesigned to support the high performance required by visually rich Java applications such as games. The native support for the gaming API in versions of Symbian OS before v9.1 has been deprecated and the `javax.microedition.lcdui.game` package was reimplemented in pure Java, with much improved performance and additional collision detection support.

Pros and Cons of the Integration Approach

As we now see, the many advantages of integrating LCDUI with native peers and native rendering primitives gives a strong case for this approach.

The LCDUI implementation approach of having a core framework for common functionality and two additional DLLs that contain all code that requires customization means that the Symbian OS licensees can make highly localized changes and benefit from the already implemented core functionality. At the same time, the usage of native peers by the LCDUI widgets DLL greatly reduces the porting effort required for licensee platforms.

The widgets DLL makes Java applications look, react, and accept input in the same way as their native counterparts, which enhances the user experience and makes Java applications indistinguishable from native C++ applications. All input methods are inherited from the native widgets; everything is allowed, from T9, cycling FEPs, and handwriting recognition to support for the input of non-keyboard characters (Chinese, for example), support for pointer and keyboard input, and more. All text-based widgets inherit the support for bi-directional text, word wrapping for locales, capitalization, mixed direction mode, and so on.

Because of the integration with APPARC, Java applications can run as native applications in their own window group, can be presented as individual applications in the task list, can be brought to the foreground if already running, can change screen orientation, and more.

Challenging the Integration Approach

The alternative approach we present is that of the Java ME subsystem providing its own set of predefined graphical elements (for example, a softkey, a system menu, a ticker, or a progress bar) that can be customized to create a distinct look and feel which closely matches the native widgets on a device. Symbian OS licensees integrating Symbian OS onto their phone could use the default supplied skin or they could customize it for their own look and feel. For example, a set of images and graphical properties are defined to create the look and feel of a TextField or a softkey. Expert visual designers customize the images and properties to match the native look and feel.

This approach is definitely a good option and a fit-for-purpose solution in many other mobile platforms. However, what fits one deployment model does not fit another. Symbian OS is designed from the ground up to be modular and customizable. Symbian OS devices get their look and feel from the UI platform, extending the underlying Symbian OS frameworks. In this deployment model, UI customization is a mandatory step in the Symbian OS integration; duplicating the effort for the Java ME subsystem would incur great expense and require significant effort to implement what is already implemented elsewhere. Such an approach,

although it is feasible and can provide excellent visual results, would not be acceptable in the Symbian ecosystem.

Conclusion

LCDUI is a mission-critical API which every Java application must use and has a direct effect on the end-user experience. It must operate in the way which best fits the Symbian OS model. To enforce consistent behavior and functionality, with minimal customization for Symbian OS licensees, LCDUI is tightly integrated with the native widgets and native rendering primitives.

11.5.4 Integration: JSR-135 MMAPI and MIDP 2.0 Media API

JSR-135 MMAPI is a direct superset of the JSR-118 MIDP 2.0 Media API, defining the support for tone generation, media playback, MIDI, video, camera, and more. The two main high-level classes used in this API are `DataSource` for protocol handling and `Player` for content handling. Together, they encapsulate the two parts of multimedia processing: handling the data delivery and handling the data content.

Applications use the MMAPI Manager to request `Players`, by giving the locator string (e.g., `capture://video`), and to query properties, supported content types and supported protocols. Fine-grained control is an important feature of JSR-135 MMAPI. Therefore, each `Player` provides type-specific controls to expose features that are unique to a particular media type. Support for some features of some media types is mandatory while other features are recommended or entirely optional.

Not all MMAPI implementations support all multimedia types and input protocols. Some implementations may support only a few selected types and protocols. For example, some may support only playback of local audio files. The API is designed in a way that allows optional or new features to be added without breaking the existing functionality. It is designed so that by staying with a high level of abstraction, JSR-135 implementations can extend the API to support newly introduced features.

Applicable Native Services

The multimedia subsystem provides the multimedia capabilities of Symbian OS. These include audio recording and playback, video recording and playback, still image conversion and camera control. Not all of these capabilities are necessarily present but frameworks exist in each case to support them. Ultimately, the functionality provided is at the discretion of the licensee.

The Multimedia subsystem is split into three components:

- The Multimedia Framework (MMF) is a lightweight, multithreaded plug-in framework for handling multimedia data. The MMF offers

client utilities for common high-level tasks such as audio playback, recording and conversion; tone playback; video playback and recording; MIDI playback; and automatic speech recognition.

- The Image Conversion Library (ICL) provides a lightweight library, based on active objects, which supports the decoding and encoding of image files. A number of standard decoders and encoders are supplied by Symbian OS and licensees typically provide additional decoders and encoders as required by their platform.
- The Onboard Camera API (ECam) allows an application to access and control any camera hardware which is attached to the device. The API provides functions to query the status of the camera, adjust camera settings, and capture images and video.

For further information about multimedia on Symbian OS, please consult [Rome and Wilcox 2008].

Integration with Native Services

The integration of the Java ME multimedia framework with the underlying Symbian OS multimedia subsystem is better discussed by multimedia functionality areas. The integration of a JSR's functionality with the underlying subsystem is always influenced by the native subsystem architecture. In the case of JSR-135, the generic and compact APIs use locators to distinguish between devices and functionalities, and run-time casting of controls to perform the actual multimedia operation. On Symbian OS, the design of the native multimedia subsystem separates devices and functionalities into different components of the multimedia subsystem (e.g., CCamera is found in ECam and utility classes are found in MMF).

As a first step, the integration between the two ends requires inspecting the given locator and instantiating the relevant multimedia component objects and MMF utility classes. Each of the JSR-135 `Player` methods or `Control` methods should be mapped to an equivalent native use case, which might require triggering operations in a collection of the multimedia subsystem's primitives. In some use cases, the Java ME subsystem is required to perform some preparatory tasks before delegating the control to the native multimedia subsystem.

As it turns out, there is still a certain degree of correlation between the two ends. This is mainly because of the utility classes in the MMF, which provide very high-level support for the common multimedia use cases and classes with utility methods that trigger the execution of a whole use case with a single method call; for example, you open an audio clip using `CMdaAudioPlayerUtility::OpenFileL()`. (The opposite of such correlation can be seen in the integration of JSR-180 SIP, discussed in Section 11.5.6, which increases the integration complexity because it requires additional adaptation layers to integrate the Java API onto the native API.)

Let us now give a concrete example of the integration of Camera functionality. The following code snippet demonstrates how to take a picture using the phone camera:

```
Player p;
VideoControl vc;
// initialize camera
try {
    p = Manager.createPlayer("capture://video");
    p.realize();
    // Grab the video control and set it to the current display.
    vc = (VideoControl)p.getControl("VideoControl");
    if (vc != null) {
        Form form = new Form("video");
        form.append((Item)vc.initDisplayMode(vc.USE_GUI_PRIMITIVE, null));
        Display.getDisplay(midlet).setCurrent(form);
    }
    p.start();
    byte[] image = vc.getSnapshot(null);
    // do something with the image ...
} catch (IOException ioe) {
} catch (MediaException me) { }
```

An application that uses a camera could be expected to use VideoControl to manage the display (from the JSR-135 point of view, a camera attached to a device does not differ from any other video content); it could also be expected to use RecordControl and other supported and relevant Controls (e.g., RateControl). The multimedia subsystem support for the functionality of this specific use-case is shared between ECam, which defines the CCamera class, and MMF, which defines CVideoRecorderUtility. Therefore the bottom layer in JSR-135 instantiates the two objects and uses their high-level API to delegate the functionality operations. For example, calling CCamera::CaptureImage() when VideoControl.getSnapshot() is invoked or CVideoRecorderUtility::Record() in response to invoking RecordControl methods.

Although we did not go into the fine-grained details of the integration, this single example demonstrates well how using the MMF, which is also designed as a high-level API, to delegate functionality to the native subsystem is done using only a limited number of native operations. In the same way, MIDI and audio are played using CMidiClientUtility and CMdaAudioPlayerUtility respectively; and conversion between different image formats is performed using the multimedia ICL subsystem.

However, there are still cases where the MMAPi implementation has to perform tasks which are not taken care of by the MMF. One example is the controlled fetching of audio playback data when a locator input string is provided. The preferred way is to use the MMF API to get the content from a URL (e.g., CMdaAudioPlayerUtility::OpenUrlL()). However, if this feature is not supported at run time, the MMAPi implementation

handles fetching data itself by downloading it to a temporary file and playing the audio when the download is complete.

To allow easy customization by licensees, the Java ME multimedia system was designed on similar principles to LCDUI: the implementation splits into the multimedia core framework,⁷ which contains the Java code, and generic native code that is responsible for loading and invoking methods in a customizable DLL.⁸ The customizable DLL contains all dependencies on the native multimedia subsystem. For example, the controls that should be added to each player depend on the MMF API used; it is a decision that is taken at compile time and can be customized by licensees. A set of C++ interfaces and factory classes isolate the MMAPI core framework from the customizable DLL.

An issue that has not yet been discussed is the integration of JSR-135 with LCDUI, for the purpose of playing video content, for example. In the above code snippet, the MIDlet displays the view of the camera on the MIDlet display area. As discussed in Section 11.5.3, the LCDUI widgets and low-level rendering are performed in two customizable DLLs. To enable content rendering on the screen, such as a video clip or camera view finder, the native implementation of `Canvas` or `CustomItem` in the two DLLs must satisfy an interface contract which is used by the MMAPI native implementation to pass the content to be rendered in the MIDlet display area.

Conclusion

The native multimedia subsystem was designed to be extendable and customizable in order to handle the many multimedia types and content formats that exist, to enable adding new types and formats that are constantly being introduced and the many diverse methods to store and deliver these various media types.

This section demonstrates that the JSR-135 MMAPI and JSR-118 MIDP 2.0 Media API integration with the underlying Symbian OS multimedia subsystem is designed to meet the requirements necessitated by demands of media-centric and high-performance Java applications. In that sense, the integration is not only mandatory but also ensures consistent behavior and the required customizability.

Access to hardware is usually restricted to supervisor-mode code. A Symbian OS user-side application that needs access to peripheral resources can do so only through the native Symbian OS subsystem that governs the supported functionality, using its exposed frameworks and APIs. Specifically, the Java ME subsystem, which is a user-side

⁷The MMAPI framework is under `\src\common\generic\j2me\components\multimediammf\` in the Symbian OS source repository.

⁸The MMAPI plugin is under `\src\common\generic\j2me\plugins\refmultimedia\` in the Symbian OS source repository.

application, cannot access the multimedia peripherals without going through the multimedia subsystem and using its exposed frameworks and APIs. The integration of MMAPi with the multimedia subsystem is a strict requirement. All functionality of audio and video playback, recording or processing of concurrent multiple multimedia data streams, is delegated to the native multimedia subsystem.

A clear benefit of this mandatory approach is that Java ME applications are consistent with native applications in regards to supported content, playing and recording behavior. UI platforms can further ensure this consistency by making localized changes in the customizable DLL, which acts as an isolation layer between the MMAPi core framework and the native multimedia subsystem.

11.5.5 Integration with Non-Operating-System Services: JSR-177 SATSA APDU

Some JSRs impose a strict requirement for integration with native services. For example, TCP support must use the native networking stack, which, amongst other responsibilities, acts as a high-abstraction layer on top of the baseband hardware that is used for sending radio signals over the air. In the case of TCP, Symbian OS has a well-defined native architecture, ESock, and public APIs (e.g., RSocket) that are used by the Java ME GCF framework.

But what can be done in the case of an optional JSR for which Symbian OS does not have native support? There are technologies where a mandatory native service for the JSR implementation is privately owned by the Symbian OS licensee or a network operator. We now discuss such a case, the JSR-177 Security and Trust Services API (SATSA) Application Protocol Data Unit (APDU) package.

SIM cards store network-specific information (e.g., IMSI, Local Area Identity) used to authenticate and identify subscribers on the network, other carrier-specific data (e.g., Service Provider Name) and types of applications. The use of a SIM card is mandatory in GSM; Universal Subscriber Identity Module (USIM) is the UMTS equivalent; in CDMA-based devices, the Removable User Identity Module (RUIM) is popular.

The Application Protocol Data Unit (APDU) is the communication protocol between a reader and a card. There are two types of APDU packet: a command APDU is sent to the card for processing and a response APDU is sent back from the card to the reader; both are defined by the ISO 7816 standard.

JSR-177 SATSA defines the APDU protocol handler and provides further support for logical channels, slot discovery, (U)SIM Application Toolkit ((U)SAT), PIN operations, Answer To Reset (ATR), and so on.

The main class is `javax.microedition.apdu.APDUConnection`, which acts as the protocol handler for ISO 7816-4 communication

to a smart-card device. Java ME applications can use this interface to communicate with applications on a smart card using the request–response APDU application-level protocol.

Applicable Native Services

The Symbian OS Telephony subsystem has an abstraction layer for only a subset of the APDU package’s required functionality. Symbian’s (U)SAT ETel API (e.g., `RSat`, which inherits from `RTelSubSessionBase` and is defined in `etel.h`) offers access to the (U)SIM Application Toolkit on GSM networks and to the CDMA Card Application Toolkit (CCAT) on CDMA networks. The (U)SAT and CCAT allow the card to be more than just a storage device; they define a protocol that allows the card to ask the phone to perform tasks such as displaying a message on the screen, adding items to the phone’s menus, dialing a number, or browsing to a URL.

However, the (U)SAT ETel API does not satisfy JSR-177 APDU requirements. (U)SAT is only a partial and optional requirement of the JSR and that leaves a big functionality gap between the native support and the required Java ME JSR functionality. It is clear that the available Symbian OS services cannot satisfy the JSR’s mandatory integration requirements and another native library is required. We discuss how the Java ME subsystem solves the integration challenge with another library to which Symbian OS licensees have access.

Integration with Native Services

The Java ME subsystem uses a variant of the Bridge design pattern (see Figure 11.5 and [Gamma *et al.* 1994]). A limited set of pure abstract C++ classes (M classes in the Symbian C++ idiom) has been defined and has to be implemented by Symbian OS licensees or network operators in a separate DLL that is loaded the first time a SIM application is selected.

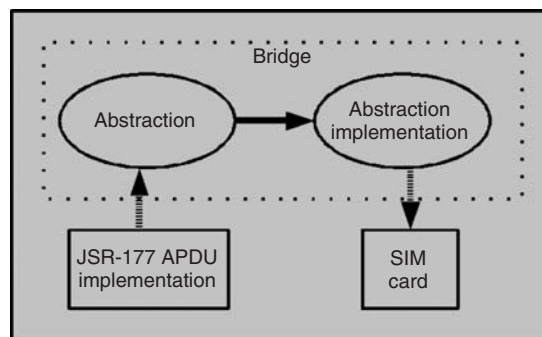


Figure 11.5 Applying the Bridge design pattern to the JSR-177 SATSA APDU

This set of interfaces goes down as far as possible to the underlying low-level functionality of the JSR, which leaves all of the higher-level functional responsibilities of the JSR to be handled by the Java ME subsystem itself (either in the Java system classes layer or in one of the native layers) as shown in Figure 11.6. The licensee's implementation of the APDU Bridge classes does not need to be aware of either the Java application triggering the operations on the SIM card or the Java ME subsystem.

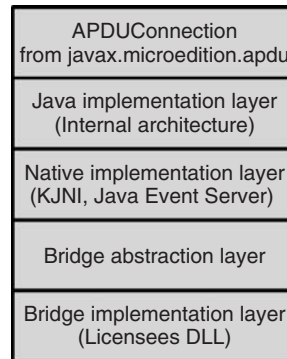


Figure 11.6 Implementation architecture layers of SATSA JSR-177 APDU

The upper layer in Figure 11.6 is the Java code layer, internal to the Java ME subsystem, that breaks down the functionality of the `javax.microedition.apdu.APDUConnection` into a few Java classes. For simplicity, the JSR-177 SATSA APDU package defines a single interface which abstracts functionality which is non-trivial and even quite complex. This requires the APDU package implementation to decouple the support into several Java classes, each of which has a single responsibility (e.g., smart card and slots, SIM-related security, APDU commands and responses, APDU validation). As expected, the internal Java package that implements the functionality support of the `javax.microedition.apdu.APDUConnection` is integrated into the Java ME subsystem's GCF internal framework.

Between the Java code layer and the licensees' interfaces implementation DLL, there is a layer of native C and C++ code whose role is not to do any processing specific to the JSR functionality, but to provide the required layer of the Java ME subsystem's solution to the Symbian OS asynchronous programming idiom that was discussed in Chapter 10 (e.g., handling Symbian OS asynchronous operations using the Java Event Server).

All of the code mentioned so far is built into the VM executable while the implementation of the native interfaces is loaded from the DLL which is provided by the licensees.

Most operations (e.g., opening a connection to a SIM application, exchanging APDU) result in initial processing of the internal implementation Java classes, jumping into the native code, invoking the licensee interfaces, and returning the result.

Data that is static or does not change during the application lifecycle (e.g., SIM security information, ATR) can be fetched from the SIM card once, by calling the set of interfaces that are implemented by licensees. The data is stored in the internal implementation Java classes and whenever it is needed by the Java application, the cached information is fetched directly without going again into native code.

Challenging the Integration Approach

It is clear that integration with native services is mandatory in this case. Let us try to challenge this specific implementation design approach by suggesting an alternative – a native integration which is very similar to the JSR APIs. Let's take it to the extreme and suggest that for each Java method in the APDU package there would be an equivalent native function. In other words, every Symbian OS licensee would be required to provide their own implementation, entirely in native code.

This solution satisfies all of the factors we mentioned – strict requirement, performance, customizability, consistency and strategy. But it has a significant disadvantage in that it multiplies the JSR implementation effort by the number of licensees or by the number of APDU-package-enabled devices.

Such an approach would fragment the Java ME platform on Symbian OS and eventually that might have an impact on the portability of Java applications.

Conclusion

Using JSR-177 SATSA APDU package, we discussed a case where integration was mandatory yet the native APIs are owned exclusively by Symbian OS licensees or network operators. We have also seen again the principle of Java code internal to the Java ME subsystem at one end, native Symbian OS services at the other end, and between them layers of native C and C++ code that is owned by the Java ME subsystem.

11.5.6 Tight Integration: JSR-180 SIP

We now discuss an additional JSR integration type in which all the required functionality is fully available as a native Symbian OS service and

the decision was taken to tightly integrate the JSR with this native service. Integration with native services is the obvious solution to implement. However, as we see, there are many challenges and considerations to be taken into account. We discuss such a case with reference to JSR-180 SIP.

Session Initiation Protocol (SIP) is an application-layer control (signaling) protocol for creating, modifying, and terminating two-party or multiparty sessions that include Internet telephone calls, multimedia distribution, multimedia conferences, instance messaging, and more. At the core of the large number of SIP RFCs that define the SIP protocol extensions and behavior for SIP user-agents is RFC 3261 from the IETF SIP Working Group.

SIP is a highly dynamic protocol, similar to HTTP, that enables intelligent user agents to send text-based messages over an independent bearer protocol (e.g., TCP or UDP) to one another by using other SIP end-points (e.g., SIP servers, proxies, and registrars). Although SIP requests and responses bear a striking resemblance to HTTP messages, the protocol itself and its usage are fundamentally different and SIP is much more evolved for modern multimedia networks.

SIP is used in conjunction with other IETF protocols in order to build a complete multimedia architecture, which provides complete services to the users. SIP sessions usually involve protocols such as Real-Time Transport Protocol (RTP) for transporting real-time data, Real-Time Streaming Protocol (RTSP) for controlling delivery of streaming media, and Session Description Protocol (SDP) for describing multimedia sessions. All the voice, video and other payload communications are done over these other protocols (e.g., RTP, RTSP) but the basic functionality and operation of SIP does not depend on any of these protocols.

JSR-180 SIP defines a compact and generic optional Java ME GCF package that provides MIDP applications with SIP functionality at transaction level, enabling them to send and receive SIP messages. In addition, JSR-180 SIP defines support for request refresh, transport protocols, SIP identity, and authentication.

JSR-180 mandates the implementation to support at least RFC 2976, RFC 3261, RFC 3262, RFC 3265, RFC 3311, RFC 3428, RFC 3515, and RFC 3903. The JSR implementation must implement those requirements of the RFCs that are relevant to the stack, but implementing the application-level requirements is the responsibility of applications using JSR-180 SIP.

The core SIP functionality is facilitated by the main Java interfaces in `javax.microedition.sip`: `SipClientConnection`, `SipServerConnection` and `SipConnectionNotifier`; and other helper classes (e.g., `SipDialog`, `RefreshHelper`, etc.). A typical SIP application uses both client and server connections to send and receive

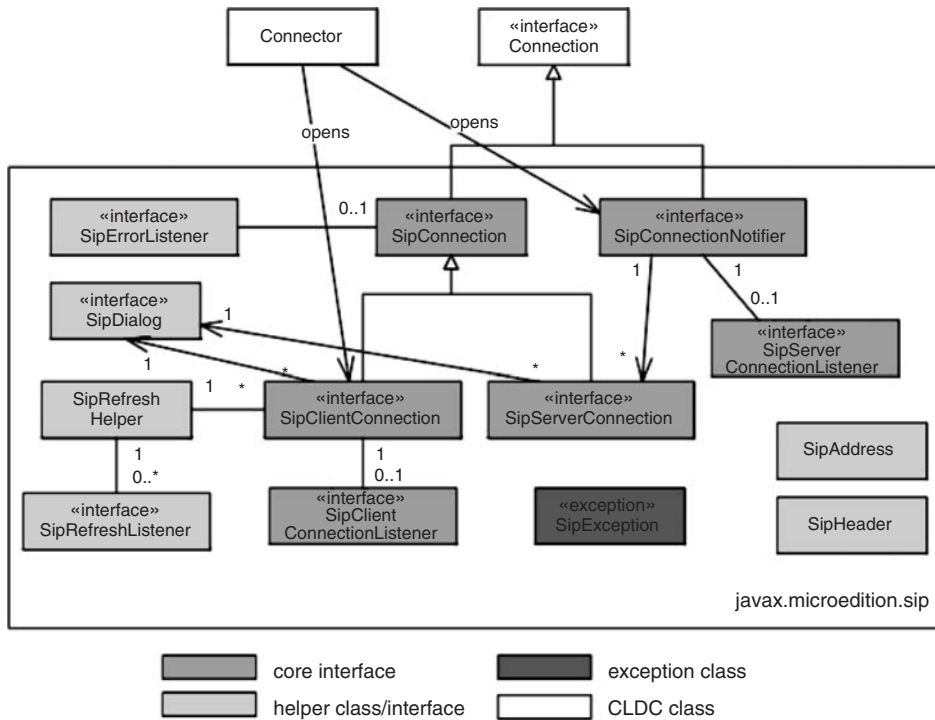


Figure 11.7 JSR-180 SIP classes

SIP messages in and out of SIP dialogs. Figure 11.7 illustrates the relations between JSR-180 API classes and interfaces.

Integration with Native Services

The native Symbian OS SIP architecture is too elaborate to fully explain in the context and within the scope of this chapter. The native SIP service contains a very large number of classes, with an architecture which is considerably more complex than the generic and compact Java ME architecture; it is decomposed into more than one set of APIs. This is only one of the differences that make the integration of JSR-180 SIP very challenging.

The native and Java APIs are very different from one another so the two are difficult to compare. We consider a number of the differences between the Java and native architectures that indicate how different they are and see how the two APIs can be bound.

JSR-180 SIP was designed to be a compact and generic API, in regard to the SIP methods and headers, and the designer chose to use

String-based parameterized operations. The native API takes a different design approach; it is based on a class hierarchy in which there is a class-based abstraction for every SIP method or header.

`SipConnection`, the base interface for SIP connections, defines the String-based `addHeader()` and `setHeader()` methods to manage SIP message headers; `SipClientConnection` defines the String-based `initRequest()` method to initialize the SIP request. In the native SIP API, the application typically needs to instantiate a `CSIPMessageElements` object when it is going to send a SIP request. The headers that are put into the `CSIPMessageElements` object are not Strings but are encapsulated in subclasses of the `CSIPHeaderBase` class (e.g., `CSIPContactHeader`). Another class, `CSIPRequestElements`, contains the `CSIPMessageElements` and the mandatory parts of any SIP request such as the To header, From header, Remote URI, and request method.

In the native SIP API, each supported SIP method is encapsulated by a subclass of the `CSIPDialogAssocBase` class (e.g., `CSIPInviteDialogAssocBase`).

These different design approaches enforce different API usage. For example, using JSR-180 SIP, the application creates a `SipClientConnection` using the GCF factory method, invokes `initRequest()` with an INVITE String, and adds or sets the headers using `addHeader()` or `setHeader()`. The following snippet of code demonstrates sending an INVITE:

```
scc = (SipClientConnection) Connector.open(address.getString());
// initialize INVITE request
scc.initRequest("INVITE", null);
scc.setHeader("From", "sip:user@host.com");
scc.setHeader("Accept-Contact", "*;type=app/chess");
scc.setHeader("Contact", "sip:user@" + localAddress);
scc.setHeader("Content-Length", "" + sdp.length());
scc.setHeader("Content-Type", "application/sdp");
```

Using the native API, the application creates the `CSIPInviteDialogAssoc` and `CSIPMessageElements` objects. The application fills the relevant SIP headers into `CSIPMessageElements` before passing it to `CSIPInviteDialogAssoc` using the `SendInviteL()` function.

In short, the two APIs are fundamentally different. The solution to those different design approaches was to have native peers for Java objects, which would include an internal layer whose functionality is to translate between Strings and classes.

Since there is no exact equivalent in native code to the Java classes, a peer object is actually a container of native SIP classes executing upon them according to changing states, triggered by Java method invocations.

Upon receiving a SIP method or header in the form of a `String`, the translation layer matches it to a single concrete native SIP stack class or, more likely, to a number of interoperable native classes with an equivalent target functionality. The native peers instantiate and use the native SIP stack classes depending on the state and context of the peered Java object. To optimize the performance, the `String` parameters are serialized before they are passed from Java to the native layer. Incoming data is deserialized before it is passed from native to Java code.

There are also considerable differences in the abstraction details and implementation idioms. JSR-180 SIP was designed to be generic and portable over many phones. The abstraction level is very high and, as a consequence, it does not provide support for various configuration details of components specific to Symbian OS. For example, SIP user profiles (which are stored in the phone's memory and can be configured manually by the user or over the air by the network operator) do not have an equivalent abstraction in JSR-180 SIP. The native SIP stack explicitly uses an IAP entry in the communications database that is a logical summary of how to get connectivity access using a special protocol or bearer type; again, that usage does not have any equivalent in Java ME.

To fill any gaps, the JSR-180 implementation contains internal logic which handles and resolves such configuration issues when using the underlying native SIP API; for example, deciding which SIP profile to use according to the mandatory SIP headers set by the application, and using an IAP assigned to the MIDlet suite by the user in the phone settings.

In addition, there is another example of a big difference between the Java API and the native SIP stack, in accepting SIP requests outside a SIP dialog. In JSR-180 SIP, applications can use the `SipConnectionNotifier`, which queues received messages. In order to receive incoming requests, the application calls the `acceptAndOpen()` method, which returns a `SipServerConnection` instance when a SIP message is received.

The following snippet of code illustrates the usage of `SipConnectionNotifier`:

```
private void startListener() {
    try {
        if (scn != null) {
            scn.close();
        }
        // listen to requests on port 5060
        scn = (SipConnectionNotifier) Connector.open("sip:5060");
        scn.setListener(this);
    }
    catch (IOException ex) {
        // handle IOException
    }
}
```

In the native SIP stack, the SIP Client Resolver framework defines the architecture for a solution that makes the resolution of target SIP clients possible upon receiving SIP requests from the default port. However, it works in a very different way; the major challenge is that it uses another Symbian idiomatic mechanism, the ECom plug-in framework, which is a Symbian OS generic architecture for providing and using functionality through plug-in modules.

Native SIP clients must implement an ECom interface called SIP Client Resolver API that is not running in the same process as the SIP client application. Upon receiving a SIP request, outside a SIP dialog, the loaded ECom plug-in is asked by the framework if it can handle the incoming SIP request, based on its fields. When the plug-in responds that it can do so, it does not handle the request itself but returns the UID of a matching SIP client application, which uses the native SIP Client API, in the application process itself.

The Java ME subsystem has to provide its own ECom plug-in which provides an implementation of the relevant SIP Client Resolver classes. Since there is only one ECom plug-in for the whole Java ME subsystem and every VM can execute multiple Java applications, the ECom plug-in is dynamically updated with the SIP fields that should be matched against incoming SIP requests that should be handled by all currently running Java applications. Figure 11.8 illustrates the internal architecture of the JSR-180 SIP implementation.

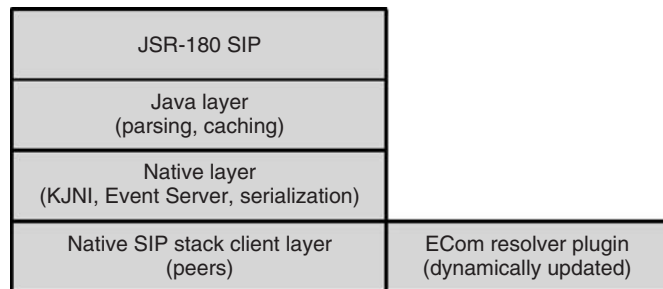


Figure 11.8 JSR-180 SIP implementation architecture

The upper layer in Figure 11.8 is a Java code layer, internal to the Java ME subsystem, that parses and caches information before it is sent into native code. The upper native layer is an intermediate layer which does not do any processing specific to the JSR functionality but provides the required layer to jump from Java into native code and the Java ME subsystem's idiomatic solution to the Symbian OS asynchronous programming idiom discussed in Chapter 10 (handling Symbian OS asynchronous operations using the Java Event Server).

The two bottom components in Figure 11.8 are the clients of the relevant native SIP stack. The ECom resolver plug-in resolves incoming SIP requests to SIP client MIDlets. The native SIP stack client layer uses the native SIP API on behalf of all running SIP client MIDlets and updates the ECom resolver plug-in with information required by listening `SipConnectionNotifier` objects.

Pros and Cons of the Integration Approach

Let us look at each of the integration criteria and evaluate how the integration approach performs:

- **Java strategy** – The option of providing a tightly integrated JSR-180 SIP is in accordance with the general Java strategy.
- **Strict requirement** – SIP is not necessarily an obvious case for mandatory integration. A different type of integration could have been applied.
- **Customizability** – Symbian OS licensees are not required to customize or extend either the JSR implementation or the native SIP API while integrating Symbian OS onto their UI platform.
- **Consistency** – The tight integration approach ensures system-wide unified behavior and functionality across Java and native applications.
- **Performance** – The parsing of `String` objects, due to the nature of the SIP protocol and the `String`-based design of JSR-180, was optimized as much as possible. The rest of the functionality uses the optimized native API.

Challenging the Integration Approach

As we have stated previously, SIP is an application layer protocol; it does not have to be integrated with native Symbian OS services. Therefore, a legitimate and interesting question is, why not implement JSR-180 SIP purely in Java? We now consider the option of providing a pure Java implementation for JSR-180 SIP.

A pure Java implementation would mean that parsing of `Strings`, validation of parameters, handling of SIP state machines as defined in the RFCs, routing of messages inside or outside SIP dialogs, bearer protocol support, and more would be implemented in pure Java.

A pure Java implementation would have had no impact on customizability. Performance would have remained fit for purpose by sticking to the principles of writing efficient and optimized Java code, and, of course, not forgetting the pre-compilation of Java bytecode into ARM instructions during the CLDC-HI VM build process (see Section 10.8 for more information).

So far it looks a good and valid option. However, there is a hidden risk. A pure Java implementation would have a fatal limitation with regard to interoperability with other native applications. They could not work at the same time. The reason is subtle yet quite simple: competition for shared resources – network ports. To accept incoming SIP requests over the default SIP port (or another shared port) both the pure Java implementation and the native SIP stack would try to acquire the port number (e.g., 5060, the default SIP port). The first application execution environment to take ownership of the port would be the one whose hosted SIP applications would receive incoming SIP requests. Such behavior that violates interoperability is not acceptable in Symbian OS, which is designed to run multiple applications simultaneously.

Let us consider a scenario in which the Java ME execution environment is competing with native Symbian applications. A user might have a native VoIP client that uses the native SIP API and which requires the loading of the native SIP Client Resolver framework. At the same time, a Java application, which was added to the MIDP Push Registry during installation, is waiting for its activation by an incoming SIP INVITE request, signaling the Java application to join a multimedia streaming session. The first application to be activated causes its application execution environment (Java or native) SIP stack to load, preventing the other application execution environment and all of its applications from receiving SIP messages.

A pure Java implementation for JSR-180 SIP is indeed fit for purpose on mobile platforms which have no native SIP stack or can run only Java applications. But in Symbian OS, which is designed to run multiple applications simultaneously, such a solution would not be acceptable.

Conclusion

Using JSR-180 SIP we discussed a case where tight integration was applied to enforce consistent behavior and functionality. Later, when suggesting an alternative implementation, we discovered that in this case, surprisingly, the integration was essential for interoperability with native applications.

Another facet of integration demonstrated in this case is an integration which is final and does not require Symbian OS licensees to provide extensions or customization; JSR-180 and the native SIP stack are both shipped ready to go.

One last issue that was obvious to identify is the possible complexity of providing a tightly integrated JSR. In the example of JSR-75 FileConnection, the Java APIs mapped well to the native APIs. A straightforward implementation was sufficient to provide bindings for all functionality. In the integration of JSR-180 SIP, the size of the native architecture and the different designs have made the integration more challenging by an order

of magnitude. And yet with all the intrinsic technical challenges, Java applications can benefit from a tight integration to the native Symbian OS SIP service.

11.6 Summary

In this chapter, we discussed the integration of Java ME APIs with native Symbian OS services in order to ensure consistent user experience between Java applications and native applications. We have seen that there are a few types and levels of integration and that the likely approach to integration is to map Java functionality to native functionality. The two worlds are fundamentally different, therefore the integration has to handle numerous challenges, as we saw in the examples. Some integration cases can be mandatory but straightforward, for example, file access, or mandatory but less straightforward, for example, MMAPi; other integration cases can be highly desirable to ensure consistency, for example, LCDUI; and other cases have their own uniqueness which demonstrates the vast scope of integration challenges, for example, SIM access and SIP.

The result of the tight integration of Java ME with native services is that on Symbian phones the Java ME platform exposes the strength and power of Symbian OS in a consistent manner, which is also interoperable with other run-time environments.

Appendix A

WidSets

Web widgets are extremely popular nowadays. These little applications, composed basically of HTML, cascading style sheets (CSS), JavaScript code and, sometimes, Adobe Flash technology, allow live content from a web service to be displayed in other places, such as other HTML pages, desktop dashboards and, more recently, in mobile devices. Some of the most common web widget platforms today are:

- Google Gadgets (www.google.com/webmasters/gadgets), which provides widgets that can be used on your desktop computer, on iGoogle (www.igoogle.com) and on your own web page
- Yahoo! Widgets (widgets.yahoo.com), which provides widgets that can be used on your desktop computer
- Windows Live Gallery (gallery.live.com), which provides widgets that can be used on your desktop computer, web pages, and a browser toolbar
- Apple Dashboard Widgets(www.apple.com/downloads/dashboard), which provides widgets that can be used on your Mac OS X dashboard.

Widgets are responsible for a great boost in web application usage, since they allow users to use the web quickly, without having to open a browser, type in a URL and wait for the page to load. These mini-applications bring many web services directly to the desktop or to a single web page, making it fast and easy to look at the weather forecast, play a crossword puzzle or read the latest news from the major sites.

Greater availability of smartphones, such as Symbian OS devices, combined with the fast evolution in networks has caused Internet usage from mobile devices to grow steadily in the past few years. Reports show that mobile Internet penetration has reached critical mass and mobile users must be considered an important audience when designing web services and applications.

Keeping in mind that mobile devices are harder to use than desktop computers, due to factors such as the lack of a mouse, small screen size, and slower network speeds, one can see that widgets gain vital importance in making it easy for mobile users to access web services. Mobile users no longer need to struggle with pages not designed for small screens and containing technologies not supported by mobile device browsers. With such a strong need to mobilize the web, a number of widget platforms targeted at mobile devices has surfaced in the past few years:

- Yahoo Go! (***mobile.yahoo.com/go***) brings Yahoo! Mobile widgets to 270+ mobile device models.
- S60 WRT Widgets (***www.forum.nokia.com/main/resources/technologies/browsing/widgets.html***) uses the browser based on S60 WebKit as a run-time environment for mobile widgets on the S60 platform.
- WidSets (***www.widsets.com***) provides mobile widgets to any device based on Java ME MIDP 2.0.

Of these, Yahoo Go! and WidSets are the most popular, with the latter being entirely Java ME-based, therefore supporting a bigger range of devices and having the potential to reach the largest mobile run-time platform, Java ME. In this appendix, we focus on WidSets and how a developer with fair experience in Java and web technologies can use it to bring the power of Internet services to devices enabled with Java ME, ranging from mid- to low-end Nokia S40 models to high-end Symbian OS smartphones.

A.1 Why Are WidSets Relevant to Java?

The reason why the WidSets platform is covered in this book is that developing widgets for it is very simple for Java ME developers. The WidSets Scripting Language (WSL) has the same look-and-feel as the Java programming language and should be very familiar to developers used to Java or any C++-based languages. It was created to allow Java and web developers familiar with JavaScript instant productivity in developing widgets for WidSets.

One may ask why Java is not used for programming, since the WidSets Mobile Dashboard itself is written in this language. The main reason is that MIDP devices do not allow loading of new class files after an application is installed, therefore making it impossible to add new functionality to the application by writing widgets in Java. Using a scripting alternative, such as WSL, overcomes this challenge and allows developers to dynamically

extend the WidSets application by creating widgets that can be used in the dashboard without having to re-install it.

Even though WSL is a different language, it shares many points in common with Java, making it easy for Java programmers to learn and adapt to it. For example, the following code is the same in Java and in WSL:

```
void test(){
    int j; /* error, missing assignment */
    {
        int c = 0;
    }
    {
        int d = c; /* error, 'c' is not visible here */
    }
}
```

Similarities and differences are explained in detail on the WidSets Wiki at wiki.forum.nokia.com/index.php/WidSets_Scripting_Language. Check it out and you will see that becoming productive in this technology is going to be a matter of hours.

It's important for Java ME developers to learn that WidSets is developed in Java. It shows that MIDP is a strong platform for developing any kind of application, ranging from games and media players to a rich widget engine with a powerful user interface that goes beyond the standard LCDUI package.

You should also keep in mind that WidSets development is not meant to replace Java ME. It is instead an easy way of developing mobile Internet services or adapting them to mobile devices. If you are developing a rich application that doesn't fit this description, such as a 3D game, an enterprise application, or a video recorder, you are better off using Java ME and its powerful APIs. Those applications are much more complex and have little to do with Internet services which are the focus of WidSets.

A.2 WidSets Architecture and Features

WidSets is an end-to-end widgets platform, composed of the following pieces (see Figure A.1):

- WidSets.com is where you manage your account and widgets, search the library for new widgets, and configure the dashboard with selected widgets, which are pushed to the WidSets Mobile Dashboard.
- WidSets Mobile Dashboard is a Java ME application that must be installed on the phone (see Figure A.2a). It is the screen where widgets are displayed to the user, who interacts with them to use the web services they represent. The WidSets Mobile Dashboard also contains the

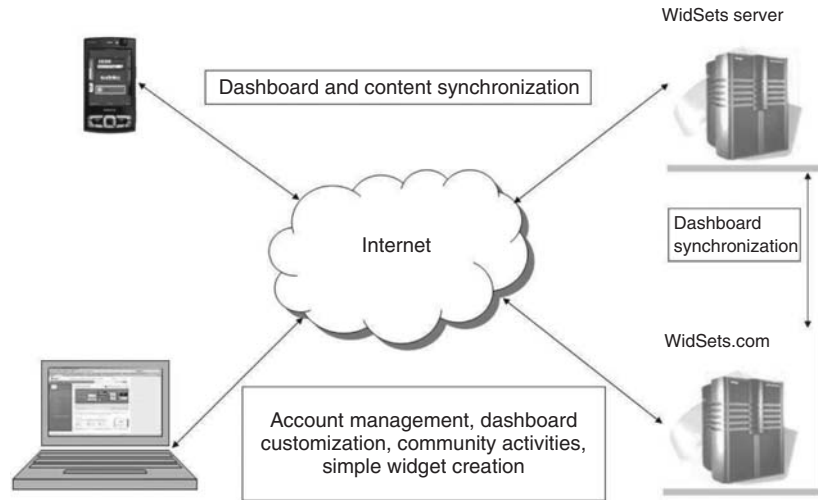


Figure A.1 Architecture of the WidSets service

engine that is used to interpret the code in which widgets are written, the WidSets Scripting Language, which we discuss in Section A.3.

- WidSets server is a back-end application responsible for synchronizing the dashboard contents between the web application and the WidSets Mobile Dashboard.
- Widgets are the mini-applications that display relevant data from selected web services to the end user (see Figure A.2b). Widgets can be created on the website or by using the WidSets SDK and are placed on the mobile dashboard, allowing users to access web content with a single click.

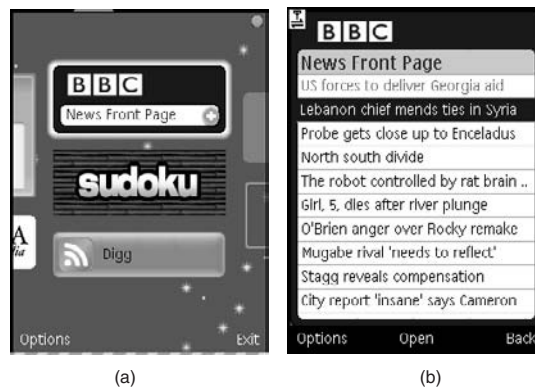


Figure A.2 WidSets on a mobile device: a) WidSets dashboard and b) BBC News widget

Widgets in the WidSets platform are mainly used to read RSS feeds from data services. In fact, this is such a common task that the creation of a feed reader is very straightforward and can be performed with the help of a wizard on the WidSets site or in the mobile dashboard. However, widgets created for WidSets are not limited to being news readers; in fact, by using the WidSets SDK (Software Development Kit) and the WSL (WidSets Scripting Language), it is possible to create rich applications using advanced features such as:

- Taking pictures with the device's camera
- Playing sounds
- Rendering HTML content
- Using WSL's rich library of UI components, such as lists, dialogs, and surfaces
- Accessing content-fetching services provided by the WidSets server.

The content-fetching services allow easy retrieval and parsing of RSS and ATOM feeds, downloading of HTML content, fetching and scaling images for mobile devices, and more. Here is a summary of the main classes and APIs available in the WSL:

- GUI components: Animation, Banner, Camera, Canvas, Choice, Component, Flow, Graphics, Image, Label, Menu, Picture, Popup, Prompt, Progress, Scrollable, Shell, Text, Ticker, View
- Persistent data storage: Store
- System: Timer, System, Config, Widget, I/O streams, data structures
- Date and time
- Sound: Classes for playing sound files
- Utility functions, including those which call services provided by the WidSets server.

Widgets are not limited to being data consumers, but can also be used to develop games, productivity applications, customized web services clients, multimedia applications, and so on. In the following sections, we see how to use WidSets by picking widgets created by other users or by creating your own.

A.2.1 Using WidSets

In order to get started, you need to register with the WidSets website (www.widsets.com) and install the WidSets Mobile Dashboard in your mobile device. The registration process is straightforward and the client installation on your device is guided by the website, which means that after you finish it you should be able to access the service both in the website and in the Mobile Dashboard. For more details on the registration, please check the Get Started pages at www.widsets.com/getstarted.html.

After having registered, you are logged in to the website, from where you can manage account settings and your widgets. Click the Dashboard Manager menu item, to see your dashboard (Figure A.3), populated with some default widgets.



Figure A.3 Dashboard manager

When you start the WidSets Mobile Dashboard on your phone, the same widgets that are presented to you on the web are also on your mobile client's dashboard (compare Figures A.2 and A.3). This is how the service works: widgets are always kept in sync between the two; whenever a new widget is added to the dashboard on the web, it shows up on the device and vice versa.

Let's now populate our dashboard with useful widgets created by the WidSets community. On the website, click on the Library menu to see a list of thousands of widgets ready to be used. Once you have chosen the widgets that interest you the most, click on the Pick icon to add them to your dashboard. Go to the mobile application and click Options, Reload. Your dashboard is immediately refreshed with the newly-added



Figure A.4 *Digg.com* widget added to the dashboard

widgets. In Figure A.4, you can see the Digg widget has been added to my mobile dashboard. Clicking on it gives me the latest stories appearing on **Digg.com**.

At the time of writing, the library contained over 6400 widgets, segmented into categories such as: Fun and Games, Blogs and Forums, Images, Mail and Messaging, and many others. Take some time to get acquainted with the library and add useful widgets to your dashboard; they will help you save time and money (in the form of reduced data traffic) while accessing your favorite websites and allow you to play your favorite games or check your email. Having learned the basics of how to use the WidSets service, let's see how we can create widgets for it.

A.2.2 Creating a Widget using the Website

Creating the most common type of widget, a feed reader, is a very simple task which can be performed directly on the website or on the mobile dashboard. There's no need to download the WidSets SDK or learn the WidSets Scripting Language. In fact, all that is needed is the website or feed address that you wish to display on your dashboard and the service creates all the code automatically.

Log in to the website and click on Create a new widget/Upload on the lefthand menu. A wizard is started and two text fields are displayed: Name, where you must type in the name of the widget, and Feed or website url, where you must type in the RSS/ATOM feed address (e.g. **feeds.feedburner.com/rawsocketDotOrg**) or the regular website address (e.g. **www.rawsocket.org**). If you give the website address, the wizard looks for suitable feeds on the website: if one is found then it is used; otherwise, an error message is displayed.

After the feed is located and parsed, you can choose an icon from the standard library or upload your own. Click on Next to customize some UI elements of the widget, such as color, effects, and the icon. Click on Finish under the Complete the process box. Figure A.5 shows a portion of my dashboard, with a newly-created widget that points to my personal website.

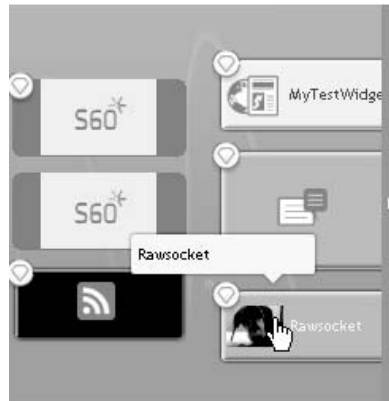


Figure A.5 *Rawsocket.org* widget added to the dashboard

Refreshing the WidSets Dashboard shows my weblog widget with the latest posts right on my mobile device: no need to open the browser, type in a URL and wait for it to parse the whole HTML page; it's all easy and fast.

A.2.3 Creating a Widget using the WidSets Mobile Dashboard

Using the mobile dashboard to create feed-reader widgets is just as easy. Open the WidSets application, click on the Library widget, choose Create new widget, and type in the feed or website URL (see Figure A.6a). You are shown a list of feeds found in that location; choose the one you want and click Select and OK. The widget is created and added to the dashboard; it can be further customized on the website. Figure A.6b shows the newly created widget in action.

A.2.4 Publishing a Widget

Once you have created a widget, you are free to use it in your own dashboard, as it is private to you, which means no other users can see or add it to their dashboard. However, as WidSets is a community-based service, you are free to publish your widgets to the master library so everybody can use them.

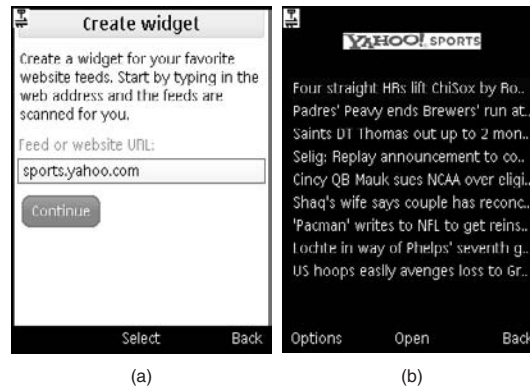


Figure A.6 Using the mobile dashboard: a) creating a widget and b) displaying the new widget

To do this, log in to the website, and click Publish to library on the lefthand menu. You are shown a list of widgets you have created. Choose the widget you wish to publish and click the Publish button. Fill in the form with the name, description and category and click Finish. Your widget is now published and can be used by the whole WidSets community.

A.3 Developing Rich Widgets

A.3.1 Tools

To get started, you need a standard text editor, such as Notepad, UltraEdit (www.ultraedit.com) or EditPlus (www.editplus.com), and the WidSets software development kit (SDK), which can be downloaded from Forum Nokia: www.forum.nokia.com/main/resources/tools_and_sdks/widsets. We recommend that you have access to the documentation, which can be found at Forum Nokia (wiki.forum.nokia.com/index.php/Category:WidSets), and to the example applications, which are in the examples folder of the SDK.

Once you have installed all the tools, open a command prompt, change to `<installation folder>\devkit\bin` and type `devkit`. If you get the options list for this command, it means the toolkit is installed and working successfully. You then create widgets in a text editor and use the command-line tool of the WidSets SDK to compile, test, and upload them. We now create a simple widget application to get acquainted with the tools, the development process and the WSL.

Some text editors can be configured for WSL syntax highlighting. Details on how to perform the configuration can be found at: wiki.forum.nokia.com/index.php/Configuring_an_editor_for_syntax_highlighting.

A.3.2 Creating a Widget Descriptor

To get started, let's create a folder named `hello_world` in the `devkit` directory. Within this folder, create `widget.xml` containing the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<widget spec_version="2.1">
  <info>
    <name>Hello World</name>
    <version>1.0</version>
    <author>render</author>
    <clientversion>1.0</clientversion>
    <shortdescription>Very simple widget</shortdescription>
    <longdescription>Simplest possible widget saying hello to the world.
    </longdescription>
    <tags>test example hello world</tags>
  </info>

  <parameters>
    <parameter type="string"
      name="widgetname"
      description="Name of widget"
      editable="no">Hello World</parameter>
  </parameters>

  <resources>
    <code src="helloworld.he"/>

    <stylesheet>
      bg {
        color-1: white;
        background: solid black;
        align: hcenter vcenter;
        border: 1 1 1 1;
        border-type: rectangle white;
      }
      text {
        color-1: white;
        padding: 2 2 2 2;
      }
    </stylesheet>
  </resources>

  <layout minimizedheight="65sp">
    <view id="viewMini" class="bg">
      <label class="text">${widgetname}</label>
    </view>
    <view id="viewMaxi" class="bg">
      <script id="hello" class="text"/>
    </view>
    <webview>
      <weblabel class="top: 0px; left: 10px;"
        style="color: black;">${widgetname}</weblabel>
    </webview>
  </layout>
</widget>
```

The contents of the widget descriptor gives the WidSets run-time environment all the information needed to create and manage the application within the defined lifecycle for widgets. The root element (`<widget>`) defines the version of the specification being used, as well as grouping all the information for that particular widget. This information is divided into several sections, of which the following are mandatory:

- The Info section describes the widget's metadata, such as the name, version, author's name, and a short description.
- Parameters are a set of dynamic containers for settings-related data. They can be used in `widget.xml`, to configure properties for the widget, or in code. For example, if we configure two parameters, `MyServiceURL = www.myfeed.com` and `color = black`, they can be used in `widget.xml` and in code to determine the location of the feed you want to read and the color of labels. You can specify that certain parameters controlling widget properties are editable. This means that they can be changed by the end user to suit the widget to their own needs, by clicking Options, Widget, Settings. Settings defined in parameters are persistent; their values are saved to the server when closing the WidSets client, and then retrieved when reloading the widget, so no changes are lost.
- The Resources block holds values for images, stylesheets and WSL source code used by the widget. In this example, only the source code and stylesheet elements are used.
- The Layout section defines the views that display data retrieved from the web services to the user in a format defined by the developer. There are two possible values for the elements enclosed by `<layout>`: `view` defines the characteristics (background image, positioning, labels, etc.) of the widget's views on the mobile client; and `webview` is used for positioning and images when the widget is seen in its minimized view on the web dashboard manager. Our example defines three views: `viewMini`, the minimized widget view on the mobile dashboard; `viewMaxi`, used when the widget is opened in its maximized view, and the `webview`, used to display a label with the widget name on the web dashboard manager.

A.3.3 Creating Widget Source Code

Having created the application descriptor, let us now proceed to creating the source code which handles the interaction between users and the widget itself. Using your favorite text editor, create the `helloworld.he` file containing the following code:

```

class helloworld{
    // It's nice to store command ids as static constants
    const int CMD_BACK = 1;

    // MenuItems are displayed over the phone's soft buttons,
    // usually Back, OK, Options, etc.
    MenuItem BACK = new MenuItem(CMD_BACK, "Back");
    // The WidSets framework will call createElement() for each script
    // element it finds from views being created by createView()

    Component createElement(String viewName,
                            String elementName,
                            Style style,
                            Object context){
        // return a simple label with the style defined in widget.xml,
        // in this case, "text"
        if (elementName.equals("hello")) {
            // in order to get the label to align in the middle of the view,
            // you'd need to contain it inside a Flow which you'd return here
            return new Label(style, &Hello World");
        }
        return null;
    }
    void startWidget(){
        // instantiate minimized view in startup
        setMinimizedView(createView("viewMini", getStyle("bg")));
    }
    Shell openWidget(){
        // instantiate maximized view when user opens this widget
        Flow view = createView("viewMaxi", getStyle("bg"));
        return new Shell(view);
    }

    MenuItem getSoftKey(Shell shell, Component focused, int key){
        // return the key we want to display at position=SOFTKEY_BACK
        // this is usually the right soft button (RSB); for other key
        // positions return null, as we don't want other keys
        if (key == SOFTKEY_BACK) {
            return BACK;
        }
        return null;
    }

    void actionPerformed(Shell shell, Component source, int action){
        // when the CMD_BACK event comes in,
        // pop the current shell (this widget)
        if (action == CMD_BACK) {
            popShell(shell);
        }
    }
}

```

This is the programming logic for the widget. First, we create a new class, `helloworld`, using similar syntax to the one used for creating a Java class. All the variables and functions must belong to this class, and at its beginning we define a few instance fields that can be used by all methods, as they have class scope:

```
const int CMD_BACK = 1;
MenuItem BACK = new MenuItem(CMD_BACK, "Back");
```

We will discuss them shortly. First, let's take a look at the most important lifecycle methods for a widget class:

```
void startWidget(){
    setMinimizedView(createView("viewMini", getStyle("bg")));
}
```

`startWidget()` is called after the widget is created. It's a signal that our widget is ready to be displayed in the dashboard and we set the view for its minimized (not open) state. Here we provide the `viewMini` view defined in `widget.xml`, which displays a label with the widget's name. The other important method called by the framework is `openWidget()`:

```
Shell openWidget(){
    Flow view = createView("viewMaxi", getStyle("bg"));
    return new Shell(view);
}
```

Here we provide a view for the widget when it's opened by the user. Our implementation creates `viewMaxi`, as defined in `widget.xml`, using the `bg` stylesheet class defined in the `<stylesheet>` element in the `<resources>` section. Upon `viewMaxi` creation, the framework checks the contents of the XML file, looking for the elements below 'viewMaxi'. In our example, it finds the `<script>` element, with an ID of `hello`. For each `<script>` below a given view, the framework calls the `createElement()` method, which we implement to create the UI elements for the maximized view of the widget:

```
Component createElement(String viewName,
                        String elementName,
                        Style style,
                        Object context){
    if (elementName.equals("hello")) {
```



```

        return new Label(style, &Hello World");
    }
    return null;
}

```

We simply return a `Label` here, which could be done using a static `<label>` element in the XML file. However, by handing over the creation of the UI to the script, we can understand how the framework functions and also learn how to create a dynamic user interface. For example, we could create a different message depending on whether it's morning, afternoon or evening.

Let's now get back to our class variables, `CMD_BACK` and `BACK`. These are used to create a `MenuItem` with the label `Back`, which is displayed on the right softkey of a device, indicating that the user may return from the widget to the dashboard. The actual work of placing the command on the softkey is performed by the `getSoftKey()` method, part of a widget's lifecycle; this is called by the framework so we can return whatever commands we'd like to be placed on the softkeys:

```

MenuItem getSoftKey(Shell shell, Component focused, int key){
    // let's change only the right softkey, whose id is SOFTKEY_BACK
    if (key == SOFTKEY_BACK) {
        return BACK;
    }
    return null;
}

```

Last but not least, we implement the `actionPerformed()` method, called by the framework when the softkeys are pressed:

```

void actionPerformed(Shell shell, Component source, int action){
    if (action == CMD_BACK) {
        popShell(shell);
    }
}

```

If the event passed to our implementation is `CMD_BACK`, that means the right softkey has been pressed and the user wishes to exit the widget. We then call the `popShell()` method, which sends the widget back to the minimized view and frees resources.

A.3.4 Compiling and Running a Widget

After saving the file, let's open a command prompt and compile it:

```
devkit compile helloworld\helloworld.he
```

If we didn't make any mistakes during the programming phase, we should see:

```
Required client version: 0.0.0  
Compile OK
```

If you get any error messages, go back to the source code and correct it, then try compiling it again until you are successful.

Once the source code compiles correctly, we are ready to run our example widget in the emulator. For that, we need to log in to the WidSets server: the widget is uploaded and run in the server context, with the results being displayed in your emulator's mobile dashboard. Type in the following command:

```
devkit login USERNAME PASSWORD
```

The username and password must be correct for your WidSets account. You receive a success message, after which you can run the application:

```
devkit run helloworld
```

The SDK compiles, packages and uploads the widget to the WidSets server. Once the widget is installed correctly on your dashboard, the emulator is opened so you can see the result. Navigate on the emulator's dashboard and look for the Hello World widget.

By now you should feel comfortable enough with the development process, so you can explore more deeply the possibilities of widget development on the WidSets platform. Check the documentation at **wiki.forum.nokia.com/index.php/Category:WidSets** for more details on the WSL, devkit, available APIs and services, example widgets, and many more. Reading this documentation will give you the background to develop complex applications like the one we show in the next section.

A.4 Creating a Flickr Viewer

In this section, we put our newly-acquired WSL and WidSets skills to use in creating a more complex application, Flickr Viewer. Our widget allows users to search the Flickr website for pictures in categories which interest them. Results are displayed in a list, with the pictures resized to fit the small screen of a mobile device. Users are also able to save a search and use it later, to speed up the searching process and save typing.

Flickr Viewer has two views: a form (see Figure A.7a) into which users can type their search queries and the picture view (see Figure A.7b), where images retrieved from the search are displayed.

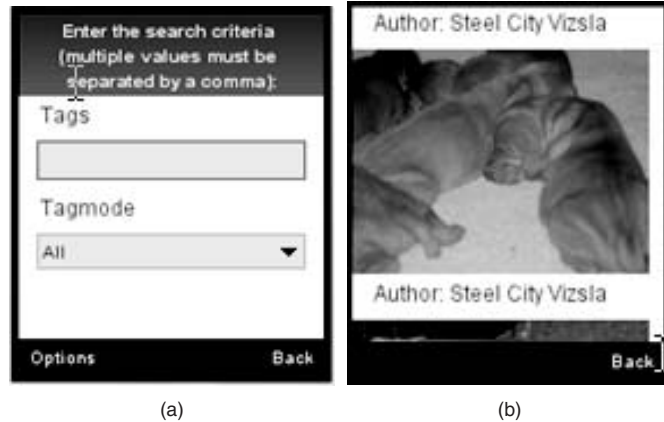


Figure A.7 Flickr viewer widget: a) form view and b) picture view

As the code and resource files for Flickr Viewer are bigger than in the Hello World example, we do not discuss the entire code here. Rather, we focus on the new features used to build this widget. The complete source code for this project can be downloaded from this book's website.

A.4.1 Descriptor File

```
<?xml version="1.0" encoding="utf-8"?>

<widget spec_version="2.0">
  <!-- info section omitted for brevity-->

  <parameters>
    <parameter name="widgetname">Flickr Viewer</parameter>
    <parameter type="url" name="url" editable="false">
      <value>http://api.flickr.com/services/feeds/photos_public.gne
    </value>
    </parameter>
  </parameters>
  <services>
    <service type="syndication" id="syndService">
      <reference from="url" to="feedurl"/>
    </service>
  </services>
</widget>
```

```

<resources>
  <code src="flickr_viewer.he"/>
  
  
  
  
  
  

  <stylesheet>
    <!--some stylesheet contents omitted for brevity-->

    <!-- Form example -->
    form.base {
      margin: 0 10 5 10;
      background: solid #e3e3e3;
      border: 1 1 1 1;
      border-type: rectangle #a4a4a4;
      padding: 2 4 2 2;
      font: proportional small plain;
      color: #333333;
      focused {
        border-type: rectangle red;
      }
    }
  </stylesheet>
</resources>

  <!--layout section goes here -->
</widget>

```

We add the `url` parameter, which points to a feed URL that is used to fetch items from Flickr. You can think of it as a file-wide variable that is used in other sections of `widget.xml`.

Another important addition is the `<services>` element. A widget can call a server-side service (from WidSets server) using its ID and some parameters. For example, there are services for consuming and parsing RSS feeds, syndicating feeds, downloading and scaling images, and so on. As you can see, services provide functionality that would be too heavy or impractical to implement in widgets, therefore most applications use many services. Our example uses one service, Syndication, which allows a widget to receive a feed in an easy-to-use format, regardless of whether it is an RSS or ATOM feed. The server also polls for updates in the feed and notifies the widget about new content.

The `<resources>` section contains a pointer to the source code as usual and also has several `` resources. These images are loaded by the WidSets engine and are made available to your widget through `getImage(String name)`. The developer does not need to worry about classpath, file path or load failures. If the image fails to load, an informative image is loaded.

All UI components used in a widget are laid out according to a stylesheet. The `<stylesheet>` element contains more and more complex

stylesheets, as it is necessary to create many CSS classes for the various components we are using: form, input, text, picture viewer, etc.

Now that we have our `widget.xml` file in place, let's analyze the source code for the Flickr Viewer widget, discussing the most important points for creating the application's UI and functionality.

A.4.2 Source Code

The first thing we notice is that this widget has much more complex views than our simple Hello World example. Let us begin with the creation of the Form view, used to input the search criteria:

```
Shell createForm() {
    Flow content = new Flow(getStyle("default"));
    content.setPreferredSize(-100, -100);

    content.add(createHeaderText("Enter the search criteria (multiple
                                values must be separated by a comma:"));

    // create some text components and choices
    addText(content, "Tags");
    addInput(content, "Tags");
    addText(content, "Tagmode");
    addChoice(content, ["All", "Any"]);
    final Shell shell = new Shell(content);
    shell.setStyle(getStyle("maxi"));
    return shell;
}
```

First, we create a `Flow` object (named `content`), which is the main container for our view. We can think of a `Flow` object as the equivalent of LCDUI's `Form` class, with the difference being that `Flow` uses HTML-style positioning. An instance of the `Style` class is passed to `content` so all components added to it use the same style definitions. We then proceed to add several UI components: a header text, serving as this view's title, then a text item, an input field, another text item and a `Choice` object.

We used several helper methods to create and style these components, and their source code is quite similar to each other: instantiate a component, passing a `Style` defined in `widget.xml`, define a label for it, and add it to the `content` container. For example, the `addInput()` method is written like this:

```
void addInput(Flow content, String st) {
    // create text-input that can contain any characters
    // input is the equivalent of MIDP's TextField
    input = new Input(getStyle("form.input"), st, &, ANY);
    input.setPreferredWidth(-100);
}
```

```

input.setFlags(VISIBLE|LINEFEED|FOCUSABLE);
content.add(input);
}

```

After creation and placement of UI components in the `createForm()` method, we create a root component, an instance of the `Shell` class. It holds all other components and can be thought of as being like a `Screen` (from MIDP) that is displayed to the user. The framework manipulates those shells, using a stack, to determine which one should be displayed at any given time. Returning this shell object in `openWidget()` causes it to be displayed as the first 'window' of our application.

Besides having softkey commands, our Form view has also a Menu, displayed when the left softkey is pressed (see Figure A.8). To create this menu we use the code below:

```

void startWidget() {
    setMinimizedView(createMinimizedView("viewMini", getStyle("default")));
    if(menu == null) {
        menu = new Menu();
        menu.add(CMD_SEARCH, "Search");
        menu.add(CMD_SAVE, "Save search");
        menu.add(CMD_LOAD, "Load search");
    }
}

```



Figure A.8 Form view with a Menu displayed

You only need to create a `Menu` object and add items to it; they are composed of an ID (defined as a constant at the beginning of the class) and a label which is displayed.

The menu can be displayed in two ways: using the `getMenu()` callback, which returns an instance of a `Menu` class like the one

created in `startWidget()`; or manually using the `menu.open()` method. As we have only two views, there would be no advantage in implementing the callback to return a single `Menu`. Therefore this code uses the second approach, implemented in the `actionPerformed()` method of your widget class, avoiding the overhead of another function call:

```
void actionPerformed(final Shell shell, final Component source,
                    final int action) {
    // omitted code
    else if(action == CMD_OPEN) {
        menu.open(shell);
    }
    // omitted code
}
```

`CMD_OPEN` is an action, represented by an `int` value, defined at the beginning of the class and passed to the OK `MenuItem` when it is constructed.

Having finished creating the Form view, let's look at the functionality it provides: an input field used to type in the search criteria (tags), a choice list for setting the search mode (All tags or Any tag) and a menu. The menu has three options: Search, Save Search and Load Search. The first option proceeds to pull data from Flickr's public photo feed, using the parameters defined by the user and the other two options allow the user to save and load a favorite search. Let's see how this functionality is implemented.

When we created the menu, we passed a few IDs to its `add()` method. These IDs are passed back in the `actionPerformed()` method, where we can take action according to the menu item that was pressed:

```
void actionPerformed(final Shell shell, final Component source,
                    final int action) {
    // omitted code
    else if(action == CMD_SAVE) {
        saveSearch();
    }
    else if(action == CMD_LOAD) {
        loadSearch();
    }
}
```

Depending on the item, we choose to save or load a search query. Here are the implementations for `saveSearch()` and `loadSearch()`:

```
void saveSearch() {
    Store store = getStore();
```

```

        store.put(STORE_TAGS, input.getText());
        store.put(STORE_TAGMODE, choice.getSelected());
    }
    void loadSearch() {
        Store store = getStore();
        String tags = store.has(STORE_TAGS)
            ? String(store.getValue(STORE_TAGS)) : "";
        String tagmode = store.has(STORE_TAGMODE)
            ? String(store.getValue(STORE_TAGMODE)) : "0";
        int i = int(tagmode);
        input.setText(tags);
        choice.setSelected(i);
    }
}

```

A `Store` is a hashmap-like structure, in the sense that it holds key–value pairs, where the key is an `Object` and the value is a `Value` object (see API documentation for more details on these classes). The main difference between a `Store` and a normal Java `Hashtable` is that a `Store` is persistent, that is, data contained on it can be used across several Flickr Viewer runs. It can be thought of as WidSets version of MIDP’s record management system (RMS) persistence scheme. The `saveSearch()` method gets a reference to the default store and adds a couple of name–value pairs, where the value is retrieved from the form fields. `loadSearch()` behaves similarly, loading values from the default store and filling in the form fields with information saved previously.

Next, let’s look at the core functionality of your application: downloading a photo feed from Flickr, filtered by the search criteria, and displaying the scaled pictures to the user. Going back once again to `actionPerformed()`, let’s see how this operation is performed:

```

// Action handler is called for widget whenever an event is fired,
// such as menu-item-select, a softkey is clicked, an item is focused,
// or item contents are changed
void actionPerformed(final Shell shell, final Component source,
                    final int action) {

    // omitted code
    else if(action == CMD_SEARCH) {
        prompt.push();
        fetch();
    }
    // omitted code
}

```

When users click on the Search menu item, we display a `Prompt` to inform them that the search has begun and is going to take some time. A `Prompt` is similar to the LCDUI’s `Alert` class: it’s a quick way of displaying short information without having to construct an entire new view. After displaying the prompt, we call the `fetch()` method, which performs the actual job:


```

void fetch() {
    String tag = input.getText();
    int index = choice.getSelected();
    String choiceString = (index == 1) ? "any" : "all";
    Buffer query = new Buffer();
    query.append("api.flickr.com/services/feeds/photos_public.gne");
    if(tag != null && tag.length() > 0) {
        query.append("?tagmode=" + choiceString);
        query.append("&tags=" + tag);
    }
    Value arg = [
        "ts" => 0,
        "refresh" => true,
        "max" => 10
    ];
    Config config = getConfig();
    config.setValue(1, query.toString());
    config.commit();
    printf(query.toString());
    call(null, "syndService", "getItems", arg, ok, nok);
}

```

The first few lines are used to get data typed in by the user and construct a proper query string to pull data from the Flickr feed service. There are two parameters: `tags`, used to filter feed contents to user's interests, and `tagmode`, specifying whether the feed items should have all tags or just one of them, in order to be included in the resulting feed.

Next we create a `Value` object, a general composite type that can hold many different objects, primitive types and other `Value` objects. It is used to communicate with the server in a generic manner much as is done in Java with methods taking `Objects` as parameters, and then casting them to something more meaningful, such as a `String`. The `Value` object takes service parameters: `ts` means 'timestamp' and is used to narrow the retrieval to a certain date; `refresh` means we want fresh data from Flickr, not the data cached in the WidSets server; and `max` sets the maximum number of items retrieved in this feed.

We then call the feed syndication service by its ID, `syndService`, defined in the `<services>` item of `widget.xml`. We specify we want to perform the `getItems` operation, passing in the `arg` parameter value and two function pointers: `ok`, for successful operation, and `nok`, called when any error occurs. Before jumping into the code for these two functions, however, we use the `Config` object to set the URL of the feed being downloaded. Why are we doing that?

Data feeds usually have static URLs, since they return the same data day in and day out. The syndication service reflects this characteristic by allowing a static URL to be configured in the `<services>` tag. However, Flickr's public photo URL is not static: it allows a query string to be passed as an argument to change the items presented in the feed. Therefore we need to dynamically configure this service parameter to allow for more

flexibility as required by Flickr's feeds. With that said, we can now safely take a look at the `ok()` function, called by the syndication service upon successful completion of the download:

```
void ok(Object state, Value ret) {
    int x, int y = getScreenSize();
    Flow content = new Flow(getStyle("default"));
    content.setPreferredSize(-100, -100);
    content.add(createHeaderText("Search results:"));
    foreach (Value item : ret) {
        Picture pict = getPicture(item.content[3][1][1].url,
                                x - 10, y - (y/3), "png8");
        pict.setFlags(LINEFEED|VISIBLE);
        content.add(pict);
        addText(content, "Author: " + item.author);
    }

    prompt.pop();

    Shell shell = new Shell(content);
    shell.setStyle(getStyle("maxi"));
    pushShell(shell);
    current = null;
    shell.updateMenu();
}
```

As the download finishes, the service calls `ok()` passing in a `Value` object that contains the entire feed, already parsed and transformed into a nice in-memory object tree, which you can navigate using the standard `object.property` notation. Here is how each item looks in the memory:

```
iid=51fcc129779ccb934a63c910fd529e6d,
title=l_c27c2a5980eb5b47e3762c1473af9788,
link=http://www.flickr.com/photos/27243990@N04/2785553064/,
created=1219359794000,
modified=1219359779013,
author=erika.chavez14,
content=(
  type=page,
  pg=1,
  left=0,
  val=(
    erika.chavez14 posted a photo:,
    (type=image,
      url=farm4.static.flickr.com/3093/2785553064_e9ea28b5fa_m.jpg,
      width=240,
      height=180),
    (type=image,
      url=farm4.static.flickr.com/3093/2785553064_d0494e50db_o.jpg)
  )
)
```

All properties in this `Value` can be used either by key (for example, `item.created` or `item.title`) or by index (`item.content[0][1][0]`), in the same way as a multidimensional array in Java. In our example, we are interested in object `item.content[3][1][1]`, which has the `url` attribute pointing to the actual picture address in Flickr's services. Once we understand exactly what we are fetching, the rest of the work is easy: create a new `Flow` object, add header text, loop through the `Value` structure, getting each item's photo URL, and call the `getPicture()` method, from the `API` class. This method asynchronously downloads each image, scales it to the screen width and 1/3 of the screen height and places it on the `Flow` object. The `getPicture()` method calls the server-side Image Service, which downloads images from the web and scales them to fit the screen of a mobile device.

Once we are done downloading the pictures, we use `API's push-Shell()` method to display the list of images to the users, filtered by the search criteria and nicely scaled to fit the mobile phone screen. Figure A.9 shows the output of a search for 'puppies'.

Users can perform a new search by pressing `Back`, which is handled by `actionPerformed()` and pops this view from the shell stack, returning to the previous, `Form`, view.

Now that you have finished developing Flickr Viewer, you can publish the widget so that all the WidSets community can use it. There are two ways to do that: zip the contents of the `flickr_viewer` folder and upload it to the website by clicking the `Create a new widget/upload` menu item, or go directly to `Publish to library`, select `Flickr Viewer` from the list and click on `Publish`. Keep in mind that this only works if you have been using your own account for development.



Figure A.9 A list of puppy images

A.5 Summary

WidSets is a great way for Java developers to use their skills to create powerful mobile applications that make use of Web 2.0 services provided by most of the top Internet players these days. The learning curve is smooth, the platform is not limited to standard JavaScript functionality and the fact that WidSets uses Java ME means that a huge audience of Java-enabled phone users can now enjoy Internet services provided by widgets.

The information presented in this appendix is accurate and verified at the time of writing and has the goal of helping developers understand, mainly from a technical point of view, different models of mobile application development. Please note that the Internet services landscape is in development and so are mobile widgets technologies. Many changes may occur in the business models or services available.

To keep yourself up to date with the latest in the WidSets world, refer to the WidSets website (www.widsets.com) and to the Forum Nokia WidSets Wiki, available at wiki.forum.nokia.com/index.php/Category:WidSets, where you can find documentation, source code for examples and many complete applications that will help you mobilize Internet services of interest to your audience.

Appendix B

SNAP Mobile

In this appendix, we introduce the development of connected mobile games, with features such as a friends list, presence, chat or instant messaging, rankings, head-to-head connected games, versatile world games, matchmaking, lobbies, and game rooms. Developing and hosting such an infrastructure is a costly and technological challenge, something that very few companies can afford in this highly competitive market. With that in mind, we focus our attention on Nokia's SNAP Mobile solution (www.forum.nokia.com/snapmobile) for developing connected Java ME games. It shields games companies from having to spend money and resources building their own multiplayer game infrastructure and provides operators, service providers, publishers and developers with strong business opportunities and new revenue streams for mobile Java ME games.

We begin with an overview of a SNAP Mobile game's business model, followed by a description of the game development process. We then review the tools used for developing connected games and finally take a look at some code examples of games using the connected gaming features provided by the SNAP Mobile platform.

B.1 Business Model

SNAP Mobile is Nokia's end-to-end solution for connected mobile games and game-playing communities. It offers operators and wireless service providers a complete turnkey approach and enables them to provide connected mobile games and create online communities quickly and with very little effort. SNAP Mobile uses the same technology that is used by Nokia's N-Gage Arena.¹ Hence, the platform offers the same high

¹ See www.n-gage.com/ngi/ngage/web/uk/en/community.html for more information about N-Gage Arena.

performance and reliability – and an infrastructure that features low risk and minimal or no upfront costs.

Mobile operators and service providers can create and deploy online gaming communities by using SNAP Mobile’s managed solutions, without any changes to the current infrastructure or additional expenditure on hardware. Figure B.1 shows how the SNAP Mobile services plug into the current operator or service provider environment.

Publishers and developers can benefit by not having to write all the middleware infrastructure for connected games. They can leverage what they already know, game development in Java ME using standard tools and IDEs, and just integrate the lightweight SNAP Mobile libraries and tools into their development process to quickly add connected functionality to standalone games or to create new, connected, online multiplayer games from scratch. Note that games developed with SNAP Mobile libraries can be deployed to a number of Java ME devices, from numerous vendors: the basic requirement is that the device supports MIDP 2.0 and CLDC 1.1. In order to find out whether a certain device supports the SNAP Mobile libraries, you can run the SNAP Mobile Device and Network Test application (see Section B.5.1).

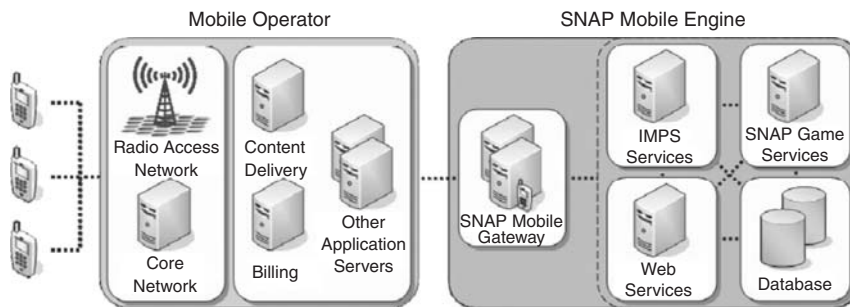


Figure B.1 SNAP Mobile hosted solutions in the operator environment

B.2 Game Development and the Publishing Process

To start explaining how game developers and publishers are supported in creating SNAP Mobile games, one first needs to have an overall picture of how the SNAP Mobile game development and publishing process works.

B.2.1 Planning

The SNAP Mobile game development process starts just like any other game: the publisher comes up with a game concept and a business case for the game. The developer creates a game design matching the concept and, while creating the design, uses the SNAP Mobile documentation

to plan the connected features for the game and to make sure that the game design meets the SNAP Mobile standard game requirements. The publisher contacts a SNAP Mobile Business Development manager to obtain the SNAP Mobile publisher license and services agreement (a contract).²

B.2.2 Development

Moving into development, you should download the SNAP Mobile tools and SDK and use the SNAP Mobile community service emulator. It is strongly recommended that you conduct mobile device compatibility testing for all targeted devices following the SNAP Mobile API Compatibility Test.

Forum Nokia, **www.forum.nokia.com**, provides server access for developers willing to develop and test their games in a live development environment.

After progressing through the Alpha, Beta, and release candidates, with related internal QA and approvals, you finally reach the stage of SNAP Mobile pre-certification testing using the SNAP Mobile Certification Emulator. Once you have ensured the correct functioning of the online features with the emulator and performed the relevant quality assurance tests, gold master for the reference device builds is reached. This means the application has passed the compliance tests and that you can start porting the reference builds to the devices tested for SNAP Mobile compliancy earlier in the process.

B.2.3 Compliance Testing

Following the development process, the completed game is tested against SNAP Mobile Standard Game Requirements (see the document *SNAP Mobile: Standard Game Requirements* available at **www.forum.nokia.com/info/sw.nokia.com/id/9e21160a-6522-4e54-a970-dc42ccd01ad1/SNAP_Mobile_Standard_Game_Requirements.html**), which set the stage for game quality requirements. The Compliance Test ensures that these requirements are met. This way, all SNAP Mobile games meet certain quality and feature criteria, which act as a stamp of approval for operators.

B.2.4 Deployment

Once the compliance test has been passed, the developer or publisher packages the game for distribution and adds the final game files to the catalog. The game is then set up in the SNAP Mobile production environment and the publisher has a one-week, pre-launch check period before

² You can contact the Business Development team by sending an email to **fn.snapmobile@nokia.com**.

the game is ready to be sold to users. Finally, the publisher submits the game for distribution certification to operators, if needed, and uploads the final game assets to the operator's distribution system as any other game.

Once the game is deployed, SNAP Mobile's connected games hosting service is responsible for ensuring the game runs 24/7. Nokia hosts and administers the SNAP Mobile platform as well as the SNAP Mobile community and provides useful game metrics to the publisher about the game's uptake.

B.3 Community Features

As this section describes, the SNAP Mobile solution offers a fully featured combination of connected and community features in one package:

- in-game community features that game developers can build within the game application
- in-game connected gameplay features that enable multiple players to play a game
- out-of-game community features that users can access from a web browser.

B.3.1 In-game Community Features

- A unique user identity allows players to recognize one another while retaining their anonymity. Players retain the same identity whether playing online, accessing the game-playing community through the game, or accessing the game-playing community through a web browser.
- The friends list lets players establish a contact list for messaging and gameplay challenges. The friends list is the user's personal directory to the connected game-playing community experience. Connected game-playing communities provide users with a way to spend time with people, share experiences, and make new friends. The friends list helps players keep track of their friends, find out which friends are online, and chat with friends or challenge friends to a game.
- Presence information encourages online gameplay and messaging between players. It lets them know when their friends are online. Presence indicators typically identify whether friends are online, offline, playing the same game, or not to be disturbed.
- Chat or instant messaging allows users to send messages to friends and other users whether they are offline or online in a lobby or in a gameroom.

- Rankings information encourages online gameplay and competition among players. Games report scores that are used to establish the top-ranked players in a given game, a player's statistics (such as scores and standings in a game), and proximity rankings (a player's ranking in relation to other players).

B.3.2 In-game Connected Gameplay Features

SNAP Mobile facilitates multiplayer gaming through:

- Head-to-head connected gameplay. SNAP Mobile provides peer-to-router-to-peer game sessions in which users can play against one another from beginning to end. These games are well-suited for challenges and matchmaking.
- Versatile matchmaking, which offers ways for users to find and compete against one another. In Challenge mode, one player requests a specific opponent; in Random mode, users automatically join any available gameroom to play against other users in that room; in Join mode, a player joins a specific gameroom and lobby; in Sort Start mode, a configurable load balancer places users in gamerooms.
- Lobbies and gamerooms, which offer places for users to meet, chat, and engage in gameplay. Lobbies contain gamerooms where game playing takes place.

Cross-operator connected gameplay increases the user base, providing customers with more opportunities for online gameplay and messaging. Cross-operator gameplay may be restricted by some operators.

B.3.3 Out-of-game Community Features

SNAP Mobile facilitates user interactivity through community features that are accessible to users from outside:

- A single unique user identity is used for logging in to the web community and the games.
- News and events inform community members about new activities and upcoming community events.
- Moderated message boards let users write messages on game and community-related topics to foster community interaction. The message board content is screened by a community moderator to

eliminate unwanted behavior (such as bad language, harassment or flaming).

- Featured game pages are a marketing tool for selected games. Each featured game has a viewable page where users can read about available features and other game info, and view screen shots.
- Rankings information is essentially the same as in game. Since the in-game user interface real estate is so small, not all rankings statistics collected within the game are usually shown. Most game developers select a subset of the statistics to display in game. However, all game statistics are made available for display on the website.
- A support section includes frequently asked questions (FAQs), self-help information, device settings, troubleshooting, and other information to help users.
- Player profiles give users the ability to learn about the people they play games with and make new friends. Users can modify their profiles to reveal just the information they want. Typical user profile information includes name, age, sex, city of residence, favorite games, mottos or favorite sayings, and the device model they are using.

B.4 Technical Architecture

SNAP Mobile provides mobile devices with in-game access to web services, instant messaging and presence (IMPS), and SNAP Mobile game services. In addition, the communities are accessible on the web, where the community sites provide content (for example, game rankings and user profiles) from the same databases as the SNAP Mobile games. The in-game technology is most relevant to game developers and publishers, whereas operators and service providers find the description of the out-of-game community technologies most relevant. Figure B.2 shows an architectural overview of the SNAP Mobile technology and systems that support it.

SNAP Mobile includes the SNAP Mobile Client API, a lightweight, client-side Java ME library that uses a standard protocol (HTTP or TCP) to make remote method calls to the SNAP Mobile gateway. The gateway offloads heavy processing to a community infrastructure, which includes services for account and profile management, authentication, instant messaging and presence (IMPS), and online multiplayer.

The SNAP Mobile Client API runs on Java-enabled devices that support CLDC 1.0 and MIDP 2.0. It provides game developers with libraries for

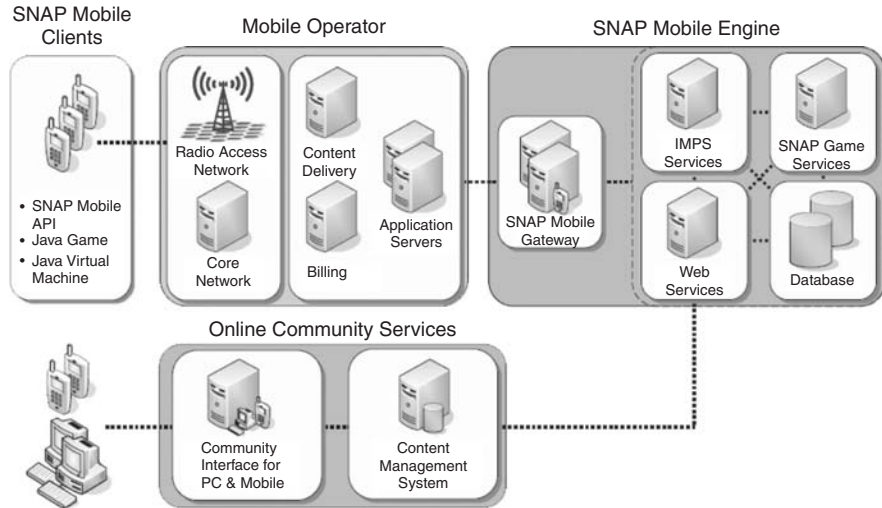


Figure B.2 SNAP Mobile architecture

including the in-game community and connected game-playing features and a graphical user interface in mobile Java ME games.

The SNAP Mobile architecture involves many components whose description is out of the scope of this book. For more detailed information on the technology, visit the SNAP Mobile page on Forum Nokia (www.forum.nokia.com/snapmobile).

B.5 Getting Started with SNAP Mobile Development

Let us now take a look at what is needed to actually start game development. Whether you are a professional developer working for a company that is publishing games or a hobbyist wanting to learn more about the platform, the steps needed on the technical side are the same.

First, you need to download the SNAP Mobile SDK, which is available from the Forum Nokia website (www.forum.nokia.com). It contains:

- SNAP Mobile API: client libraries for integrating connected features to Java ME games that use MIDP 2.0 and CLDC 1.1
- SNAP Mobile Emulation Environment: an environment that allows development and testing of games without the need for access to the SNAP Mobile Live Development Server
- SNAP Mobile Device and Network Test: an application that should be run on target devices and networks to ensure they are compatible with SNAP Mobile service
- Tools, documentation, and sample code.

B.5.1 Device and Network Test

After installing the Java run-time environment, an emulator and the SNAP Mobile SDK, you should run the Device and Network Test application on a real device. Nokia recommends that you run the test application on the devices and networks to which you are planning to deploy your game, to ensure they are compatible with the service. On the device side, being compatible means that it is supported by the SNAP Mobile service; on the network side, being compatible means packets directed to the SNAP Mobile server are not blocked by the operator. Notice that, although this step is only absolutely required for commercial developers writing real games, hobbyists and prospective developers should also run it. Doing this gives you knowledge about compatibility before you think of doing real business in this platform.

Running the Device and Network Test tool is very simple. Browse to the installation folder of SNAP Mobile SDK and look for the Tools/Device Network Test folder. You will see the JAD and JAR files for this application. Install both files to your device, using the proper tool for your phone model and vendor (e.g., use Nokia PC Suite to install on a Nokia device). Please note that you need to install using the JAD file, as it contains required application properties.

Before running the test application, it is important to set the default Access Point Name (APN) access settings correctly so the test application does not return false results. If these are not set correctly, the application may prompt the user every time a connection is established, adding the time the user takes to respond to the prompt to the Round Trip Time in the test results. If you see these prompts during the test, cancel the test application so that it does not upload invalid results to the server.

There are two types of settings that should be checked in the test application:

- The desired access point for connecting to the Internet must be set up as the default in the test application.
- The permission level for accessing the Internet (or using packet data) must be set to 'ask only once' or 'not at all', by default in the test application.

Instructions on setting these parameters vary greatly from device to device. For more information on how to set the Internet Access Point (IAP) for Java applications, please refer to Section 3.7.2.

Once everything is set, you can run the tests; the application displays the compatibility report and uploads it to the SNAP Mobile server. Just open the application and choose Start tests from the Options menu. Figure B.3 shows the test application running on a Nokia N95.



Figure B.3 Device and Network Test in action

To read more information about the configuration options for the test tool and how to interpret its results, check the *SNAP Mobile: Device and Network Test Instructions* document, available from the Forum Nokia website.

B.5.2 SNAP Mobile SDK Emulation Environment

The Emulation Environment is a local development environment and testing tool. In addition to simulating a connection to a remote SNAP Mobile server, it includes debugging features and supports precompliance testing of the game. This version provides a graphical user interface and is highly configurable. You can set up users and game class IDs, and match lobby, presence, and other attributes to your game.

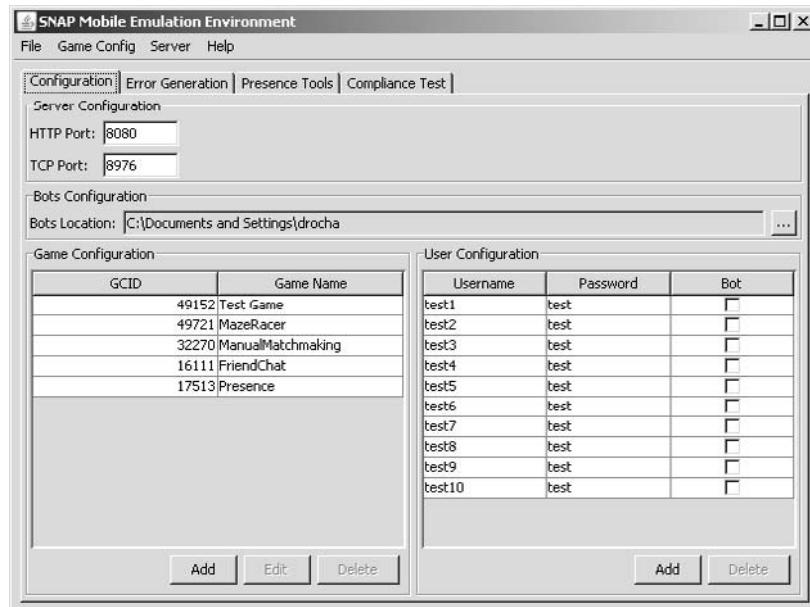
You can use the live development server (to which access is provided by Forum Nokia) during the development phase in addition to working with the Emulation Environment. Table B.1 lists the main feature differences between the remote live development server and the Emulation Environment.

Figure B.4 shows the Emulation Environment Configuration tab with a default HTTP port (8080) and TCP port (8976) for SNAP Mobile games. You can change the port settings if you are using other port numbers for your game or if you need to change them to match open ports in your firewall. The Configuration tab includes the game class ID (GCID) settings for SNAP Mobile sample games (for example, a GCID of 49721 for the **Maze Racer** sample). Each game configured on the SNAP Mobile server or the Emulation Environment has a unique GCID number, which is used for identification.

The Configuration tab also includes several user account configurations (test1/test through test10/test). You can use the text fields in this tab to modify, save, and delete game settings and to add new ones. For example, you can add and configure a new GCID and game, and

Table B.1 Comparison of the Live Development Server with the Emulation Environment

Feature	Development Server	Emulation Environment
API compatibility	Supports all SNAP Mobile API methods, including asset management methods Supports only current API version	Supports most SNAP Mobile API Methods Supports all API versions
Friends list	Saves friends to the Friends list until the user deletes them	Available in the current session only
User registration	Saves user accounts	Available in the current session only
User session	Supports thousands of concurrent user sessions	Supports up to 10 concurrent user sessions.
Game development	Supports fee-based access to the services for six-month periods Can be ordered from the Forum Nokia website through an eStore account	Supports game evaluation, planning, error handling, latency testing, and precompliance game testing

**Figure B.4** Configuration screen with SNAP Mobile sample application settings

new user configurations (user name–password values) to use when you log in with the game to the Emulation Environment.

The Emulation Environment supports multiple game configurations. Its login functionality simulates user access to the remote SNAP Mobile Community server. Like the remote development server, the locally hosted Emulation Environment allows more than one game to share the same user account configuration (user name and password) for login purposes. For detailed instructions on the configuration settings, check the *SNAP Mobile Game Compliance Testing Guide* at the Forum Nokia website.

Let us now test the connected game features of a sample game using the local Emulation Environment:

1. From the Server menu, choose Start emulator. You should see several log messages in the command prompt window showing that the startup was successful. You can also check the status of the startup by opening the Log Viewer from the File menu (or by using the Ctrl+V command).
2. Launch two emulator instances (e.g., WTK or S60 emulator) and run the sample application in both to simulate two players playing against each other.
3. Log in to the game (using the Go online option) in both emulator instances using one of the preconfigured test user name and password combinations provided in the Emulation Environment.
4. In the first emulator instance, choose the Quick Match option. The game starts and waits for a new player from the network to join the game (see Figure B.5a).
5. In the second emulator instance, also choose the Quick Match option. The server notices that someone is waiting for a challenger and connects the two players so that they are able to play a session against each other, as shown in Figure B.5b, where two **Maze Racer** players are racing at the top left corner of the screen.

The Emulation Environment is a development server emulator that can be configured in a number of ways. It provides tools and behaviors which will help you develop and test your game up to a level where it behaves well and is prepared to handle any condition inherent to connected games, such as network latency, losing contact during a match, and so on. By handling all these cases, you ensure the game is prepared to go to the Compliance Test phase of the publishing process and pass through all required tests.

For a detailed guide on the configuration of the Emulation Environment, please refer to the *SNAP Mobile: Device and Network Test Instructions* document at the Forum Nokia website.

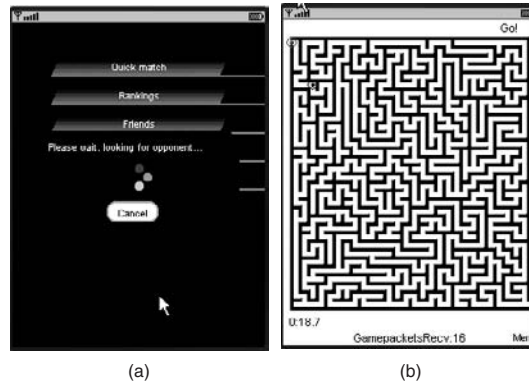


Figure B.5 *Maze Racer* game: a) waiting for an opponent to join and b) multiplayer session

B.5.3 SNAP Mobile Client API

The SNAP Mobile Client API is a lightweight library for adding connected features to Java games. It encapsulates all needed information and procedures from the server components into an easy-to-use Java API, so that you can focus on building the game logic instead of network details, matchmaking mechanisms, and so on.

Although easy to use, the API is extensive and in order to understand it in full you need to study many concepts inherent to connected games. In this appendix, we use the sample applications from SNAP Mobile SDK to understand how to perform basic connected operations, such as logging in and out, matchmaking, instant messaging, using presence information, and getting rankings and player statistics. With this baseline information in mind, you will be able to dive deep into the API documentation and build complex games utilizing the full power of SNAP Mobile. For a full description of the API, study of example games, and Frequently Asked Questions (FAQ), please check the SNAP Mobile section of the Forum Nokia Java ME site: www.forum.nokia.com/main/market_segments/games/snap_mobile_documents.html.

In this example, we see how to log in to SNAP Mobile and send, receive, and echo messages from the game server. This is the code for the HelloWorld MIDlet in a connected environment. Standard programming tasks related to MIDP have been omitted from this code; the full source code for all the examples can be found in the `SampleApps` folder of the SNAP Mobile SDK.

```
import com.nokia.sm.client.* ;
import com.nokia.sm.params.* ;
// Basic SNAP Mobile login and messaging functionality.

public class HelloWorld extends MIDlet implements CommandListener {
```

```

private Console console;
private int gameClassID = 49152;
private Session smSession;
private String username = "test1";
private String password = "test";
public void startApp() {
    console = new Console();
    Display.getDisplay(this).setCurrent(console);
    // Make the JAD file parameters accessible
    Parameters.setInstance(new MIDletParameterAdapter(this));
    console.println("Sending login.", Console.GREEN);
    SMClient smClient = null;
    try {
        smClient = new SMClient(new EventListener());
        console.println("login info:" + username + ":" + password,
                        Console.GREEN);
        smClient.loginSingleSignon(username, password, null, null,
                                   gameClassID, null, 0, null, 0, false, null);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
// omitted code for brevity
class EventListener extends SnapEventListenerAdapter {
    public void loginSingleSignonSuccess(Session session,
                                         Friend[] friendslist, Message[] motd) {
        console.println("Login successful.", Console.GREEN);
        smSession = session;
        console.println("Sending echo...", Console.GREEN);
        // send an echo
        smSession.echo(10, "HelloWorld!");
    }
    public void echoReceived(String message) {
        console.println("EchoReceived:" + message, Console.WHITE);
        console.println("HelloWorld client done!", Console.GREEN);
    }
    public void errorReceived(int code, boolean mustReturnToMainMenu) {
        console.println("Error received with code" + code, Console.RED);
        if (mustReturnToMainMenu) {
            console.println("Please return to main menu.", Console.RED);
        }
    }
}
} // HelloWorld

```

Some of the class members are used in most of the examples, as they provide standard functionality needed to perform SNAP Mobile operations and display results:

- `Console console`: a utility console to display text information being sent to or received from the server
- `int gameClassID`: the unique game ID in the game server that identifies requests and responses as belonging to this game, so the server knows where to send packets requested by the game; a game

class ID must be defined for each SNAP Mobile game; you can configure it manually for the emulated environment in the Configuration tab

- **Session** `smSession`: a session to the SNAP Mobile server through which all requests requiring login are sent and received.

In the `startApp()` method, we create a new instance of `Console` and set it as the main display. We then make the JAD file parameters available to the SNAP Mobile API using this call:

```
// Make the JAD file parameters accessible
Parameters.setInstance(new MIDletParameterAdapter(this));
```

This ensures the SNAP Mobile API can read important parameters from the descriptor file such as `SNAP-Mobile-URLs`, which defines the server to connect to (in this case, our local emulation environment).

After setting the display and required parameters, we are ready to log in to the server. This is done by creating an instance of the `SMClient` class, passing it an instance of a class implementing the `SnapEventListener` interface. Most of the SNAP Mobile programming is event-driven: you send a request using the `Session` object and receive an asynchronous response in the form of a call to one of `SnapEventListener`'s methods. In this example, we pass an instance of the `SnapEventListener-Adapter` class.

Having created the `SMClient` instance, we log in to the server, by calling:

```
smClient.loginSingleSignon(username, password, null, null, gameClassID,
                           null, 0, null, 0, false, null);
```

The main operation here is to pass the username, password and game class ID to this method, so the server can validate our game and establish a session with it. All the other parameters are given default values, as they are not important to this example. For more details on their meaning, please check the javadoc documentation in the `docs` folder of the SNAP Mobile SDK installation.

Let us now see how to handle the events that are sent back to the game in response to our requests. They are implemented in the `EventListener` inner class. In the `errorReceived()` method we can inform the user of an error condition and how to resolve it:

```
public void errorReceived(int code, boolean mustReturnToMainMenu) {
    console.println("Error received with code" + code, Console.RED);
}
```

```
if (mustReturnToMainMenu) {  
    console.println("Please return to main menu.", Console.RED);  
}  
}
```

When receiving an error, we display its code in red, so the user pays attention to what happened and acts accordingly. If the error is severe and requires the game to go back to its main menu, the `mustReturnToMainMenu` flag is set to `true`, and the game returns to the main screen. Our implementation of the error handler is not very sophisticated, but that's because little can go wrong when we are just trying to say hello to the world.

After the login is successfully performed, our event listener is notified with a call to `Session`:

```
public void loginSingleSignonSuccess(Session session,  
    Friend[] friendslist, Message[] motd) {  
    console.println("Login successful.", Console.GREEN);  
    smSession = session;  
    console.println("Sending echo...", Console.GREEN);  
    // send an echo  
    smSession.echo(10, "HelloWorld!");  
}
```

In this method, we print out a success message and keep a reference to the `session` instance, which is used to perform most of our requests. We then send an echo request to the server, containing the 'HelloWorld!' string, which is sent back to us by the SNAP Mobile server. This is useful for testing that the connection remains alive. Figure B.6 shows our application running on a mobile device.

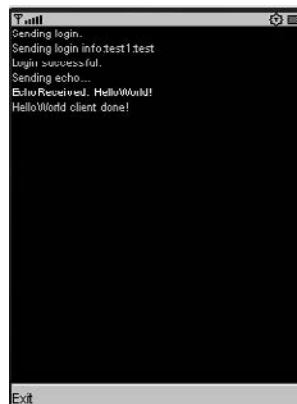


Figure B.6 HelloWorld SNAP Mobile client running

The same login, event-handling and session-establishing operations performed here are used in all other examples.

B.5.4 Presence

Now that we know how to do the basics, let us look at how to use the Presence feature on our connected games. It allows us to notify our friends whether we are available, busy, away, and so on. And, of course, it's possible also to receive events about the presence status of our friends.

In order to test this application, we start two emulator instances. In the first one, we run the Presence example configured with only a username and password. These are used to log in to the server and wait for an event to happen. In the second emulator, we run the Presence example configured with the name of a friend of the first user. Upon checking that a friend name exists in the JAD file, the application sends a friend request to the first application instance, which accepts it automatically:

```
import com.nokia.sm.client.* ;
import com.nokia.sm.params.*;
public class Presence extends MIDlet implements CommandListener {
    private int gameClassID = 17513;
    private String friendname;
    ...
    public void startApp() {
        /* Setup code omitted for brevity */
        String username = getAppProperty("username");
        String password = getAppProperty("password");
        friendname = getAppProperty("friendname");
        ...
        if(friendname!= null && friendname.trim().length() == 0) {
            friendname = null;
        }
        // set eventListener
        SMClient smClient = null;
        try {
            smClient = new SMClient(new EventListener());
            console.println("Sending login info:" + username + ":" + password,
                           Console.GREEN);
            smClient.loginSingleSignon(username, password, null, null,
                                       gameClassID, null, 0, null, 0, false, null);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
    public void destroyApp(boolean unc) {
        smSession.logout();
    }
    public void setPresenceStatus(final Friend friend) {
        smSession.setPresence(Friend.PRESENCE_STATUS_BUSY, "I'M AWAY");
        console.println("My Presence changed to AWAY", Console.YELLOW);
    }
}
```

```

/* Listener methods */
class EventListener extends SnapEventListenerAdapter {
    public void loginSingleSignonSuccess(Session session,
        Friend[] friends, Message[] motd) {
        console.println("Login successful.", Console.GREEN);
        smSession = session;
        if(friends != null && friends.length != 0) {
            console.println("Sending message to " + friends[0],
                Console.GREEN);
            setPresenceStatus(friends[0]);
        }
        else {
            if (friendname != null && friendname.trim().length() > 0) {
                session.requestFriend(friendname, "Wanna Be My Friend?");
                console.println("Sent friend request to " + friendname,
                    Console.GREEN);
            }
        }
    }
    public void friendRequestReceived(User user, String message) {
        console.println("Friend request received from user" +
            user.getUsername(), Console.GREEN);
        // automatically accept incoming requests
        smSession.acceptFriendRequest(user);
    }
    public void friendRequestAccepted(Friend from) {
        console.println("Friend request accepted!", Console.GREEN);
        // set the presence
        setPresenceStatus(from);
    }
    public void friendRequestRejected(Friend friend) {
        console.println("friendRequestRejected():sender = " +
            friend.getUsername(), Console.YELLOW);
    }
    public void friendPresenceChanged(Friend friend) {
        int status = friend.getPresenceStatus();
        console.println(friend.getUsername() +
            " changed presence status to " + status +
            "-" + friend.getPresenceMessage(), Console.YELLOW);
        if(status == Friend.PRESENCE_STATUS_BUSY) {
            console.println("Logging out", Console.GREEN);
            smSession.logout();
            console.println("Presence client done!", Console.GREEN);
        }
    }
    public void errorReceived(int code, boolean mustReturnToMainMenu) {
        ...
    }
} // class EventListener
} // class Presence

```

After setting up the console and making the JAD parameters available to the SNAP Mobile API, we check whether the username and password were correctly configured in the JAD file. Then we check whether the friend name was configured or not. If it was, then the application sends an invitation to that name, asking to be friends with it; if not, the application

waits for event delivery, including friend requests, changes in presence, and so on.

We proceed to log in as explained in the previous example, by using the `SMClient.loginSingleSignon()` method. However, in this example, we implement the Presence functionality in the login event handler as follows:

```
public void loginSingleSignonSuccess(Session session, Friend[] friends,
                                     Message[] motd) {
    console.println("Login successful.", Console.GREEN);
    smSession = session;
    if(friends != null && friends.length != 0) {
        console.println("Sending message to " + friends[0], Console.GREEN);
        setPresenceStatus(friends[0]);
    }
    else {
        if (friendname != null && friendname.trim().length() > 0) {
            session.requestFriend(friendname, "Wanna Be My Friend?");
            console.println("Sent friend request to " + friendname,
                           Console.GREEN);
        }
    }
}
```

We print out a success message and save the session as usual. We then check whether we have friends or not, through the length of the `Friend[]` array passed in as a parameter. If we have any friends, we send them a Presence message setting ourselves as away:

```
public void setPresenceStatus(final Friend friend) {
    smSession.setPresence(Friend.PRESENCE_STATUS_BUSY, "I'M AWAY");
    console.println("My Presence changed to AWAY ", Console.YELLOW);
}
```

Otherwise, we retrieve the friend's name in the JAD file and send a request to it. This triggers the `friendRequestReceived()` event, which for simplicity we handle by accepting automatically and becoming friends with the requestor:

```
public void friendRequestReceived(User user, String message) {
    console.println("Friend request received from user " +
                   user.getUsername(), Console.GREEN);
    // automatically accept incoming requests
    smSession.acceptFriendRequest(user);
}
```

Doing this triggers another event, but this time on the requestor:

```
public void friendRequestAccepted(Friend from) {  
    console.println("Friend request accepted!", Console.GREEN);  
    // set the presence  
    setPresenceStatus(from);  
}
```

We are told that our request was accepted, and therefore we can send our Presence information to our new friend. In subsequent logins, this friend's information is included in the `Friend[]` array passed to the application on the successful login event handler. In all cases, whenever we receive some presence information from a friend, the `friendPresenceChanged()` method is invoked; we can then use the information passed as a parameter to make any necessary display or status changes in our side of the application.

Testing the Presence example is a bit of work, but is not complex. First, browse to the `samples\SampleApps\dist` folder of the SNAP Mobile SDK installation. You will see two versions of this example: `Presence.jad` and `Presence2.jad`. The first is configured with `username`, `password` and `friendname` parameters, which means it will log in and try to add `friendname` as a friend. The second is configured with `username` and `password` only, which means it will not invite anybody to be a friend. However, its `username` parameter is configured as `friendname` in the first version, meaning that the friend request sent by the first is handled by the second instance of the Presence application.

Open the `Presence2.jad` application in your emulator. The application logs in to the SNAP Mobile server and waits for events. Open a second instance of the emulator, executing the `Presence.jad` application. It sees the `friendname` parameter configured and proceeds to add it as a friend, setting the presence information and then logs out. Figure B.7 shows both applications running on the Sun WTK client emulator.

B.5.5 Instant Messaging

This final example demonstrates how to send and receive instant messages to and from friends. The Instant Messaging and Presence examples combined can be used as a basis for powerful games enabled with instant messaging to run on the SNAP Mobile community server.

After logging in, one instance of the application looks for friends in the `Friend[]` parameter passed by the `loginSingleSignonSuccess()` callback method. If we find some, it proceeds to send messages to the first friend in the list (`friend[0]`), using the `sendMessage()` method:

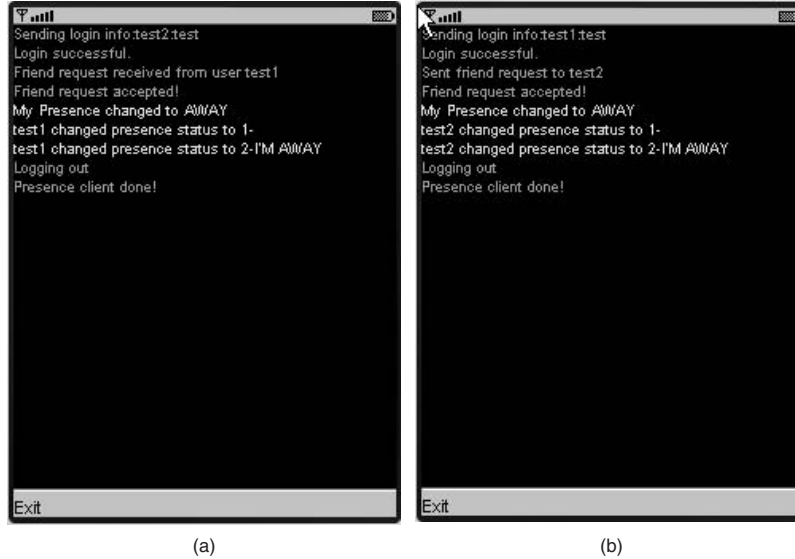


Figure B.7 Running the Presence example: a) Presence2 client and b) Presence client

```
public void sendMessage(Friend friend) {
    console.println("Sending message #" + numOfMessagesSent + " to " +
                    friend.getUsername(), Console.GREEN);

    //send message to friend
    friend.sendMessage(" " + numOfMessagesSent);
    console.println("Sent all messages. Waiting for reply...",
                    Console.WHITE);
}
```

Sending a message using `friend.sendMessage()` triggers the `friendChatMessageReceived()` event on the receiving end (in our case, the second instance of the application):

```
public void friendChatMessageReceived(Friend sender, String message) {
    console.println("Received message " + message, Console.WHITE);
}
```

For more details on using SNAP Mobile Instant Messaging, please refer to the `friendchat.jad` and `friendchat2.jad` applications in the SDK.

B.6 Summary

What we have seen in this appendix is a small fraction of the features available in the SNAP Mobile Client API. We have given you basic information on how to get started in the connected mobile game development world, but you need to read the complete documentation and check more examples from Nokia's SDK in order to have a full understanding of the power of SNAP Mobile. The `samples` folder of the SDK contains an example game, ***Maze Racer***, which uses many of the capabilities described here plus the features specific to games, such as matchmaking, ranking, player stats and more. As it is fairly complete, you can use it as the basis for your own games, and to learn the whole scope of what the Client API can do for you. You can add additional game features such as head-to-head connected games, versatile world games, lobbies, game rooms, and more.

The fast evolution of mobile phone capabilities means that games can make a transition from standalone and casual to online and multiplayer, so as to attract gamers used to the power of desktop and portable video game consoles. The SNAP Mobile service allows a smooth transition between these two worlds by providing a strong business model, powerful tools and APIs that shield you from having to build an entire online multiplayer games structure. You can leverage Nokia's robust, battle-proven SNAP Mobile technology to quickly add connected functionality to existing games or to create new multiplayer games from scratch.

For more information on SNAP Mobile, check out ***www.forum.nokia.com/snapmobile***.

References

- Allin, J. (2001) *Wireless Java for Symbian Devices*. Chichester: John Wiley and Sons.
- Arnold, K., Gosling, J., and Holmes D. (2005) *The Java Programming Language*, Fourth Edition. Prentice Hall.
- Astle, D. and Durnil, D. (2004) *OpenGL ES Game Development*. Thomson.
- Babin, S. (2007) *Developing Software for Symbian OS*, Second Edition. Chichester: John Wiley and Sons.
- Ballard, B. (2007) *Designing the Mobile User Experience*. Chichester: John Wiley and Sons.
- Gamma, E., Helm, H., Johnson R., Vlissides, J. (1994) *Design Patterns: elements of reusable object-oriented software*. Reading, MA: Addison Wesley.
- Fitzek, F. H. P. and Charaf, H. (Eds) (2009) *Mobile Peer to Peer: A tutorial guide*. Chichester: John Wiley and Sons.
- Idris, I. (2005) 'Avoiding HTTP connection limit issues'. DoJa Developer Network. Available from **www.doja-developer.net**.
- de Jode, M. (2004a) *Programming Java 2 Micro Edition on Symbian OS*. Chichester: John Wiley and Sons.
- de Jode, M. (2004b) 'Programming the MIDlet Lifecycle on Symbian OS'. Symbian Developer Network. Available from **developer.symbian.com/main/downloads/papers/midplifecycle/midplifecycle.pdf**.
- Knudsen, J. (2007) *Kicking Butt with MIDP and MSA: Creating great mobile applications*. Prentice Hall.
- Mahmoud, Q. H. (2004) 'Getting Started with the Mobile 3D Graphics API for J2ME'. Sun Developer Network. Available from **developers.sun.com/mobility/apis/articles/3dgraphics**.
- Malizia, A. (2006) *Mobile 3D Graphics*. Springer.

- Mason, S. (2008) 'Writing MIDP Games'. Symbian Developer Network. Available from **developer.symbian.com/main/downloads/papers/Writing%20MIDP%20Games%20v1.0.pdf**.
- Mason, S., and Korolev, E. (2008) 'Native and Java ME Development on Symbian OS'. Symbian Developer Network. Available from **developer.symbian.com/main/downloads/papers/Native_And_Java_ME_Dev_On_SymbianOS_%20v1.1.pdf**.
- Nokia (2006a) 'At the Core of Mobile Game Usability – the Pause Menu'. Forum Nokia. Available from **www.forum.nokia.com**.
- Nokia (2006b) 'Evolution of Mobile Gaming'. Forum Nokia. Available from **www.forum.nokia.com**.
- Nokia (2006c) 'MIDP: FileConnection API Developer's Guide (with Example) v2.0'. Forum Nokia. Available from **www.forum.nokia.com**.
- Nokia (2006d) 'MIDP: PIM API Developer's Guide (with Example) v2.0'. Forum Nokia. Available from **www.forum.nokia.com**.
- Nokia (2006e) 'MIDP Scalable 2D Vector Graphics API Developer's Guide'. Forum Nokia. Available from **www.forum.nokia.com**.
- NTT DoCoMo (2007) *iAppli Contents Developer's Guide for DoJa-5.x/5.x LE*, p. 115.
- Powers, M. (2005) 'Getting Started with the Mobile 2D Graphics API for J2ME'. Sun Developer Network. Available from **developers.sun.com/mobility/midp/articles/s2dvg/index.html**.
- Ortiz, C. E. (2007) 'A Survey of Java ME Today'. Sun Developer Network. Available from **developers.sun.com/mobility/getstart/articles/survey**.
- Rome, A., and Wilcox, M. (2008) *Multimedia on Symbian OS: Inside the convergence device*. Chichester: John Wiley and Sons.
- Rutter, T. (2003) 'Cleaner, Sharper GIF, JPEG and PNG Images'. Site-Point. Available from **www.sitepoint.com/print/sharper-gif-jpeg-png-images**.
- Sony Ericsson. Mobile 3D Tips, Tricks and Code Archive. Available from **developer.sonyericsson.com/site/global/techsupport/tipstrickscodes/mobilejava3d/p_mobilejava3d_tips_new.jsp**.
- Stichbury, J. (2008) *Games on Symbian OS*. Chichester: John Wiley and Sons.
- Yuan, M. (2004) 'Enterprise J2ME: End-to-end best practices'. Java-World. Available from **www.javaworld.com/javaworld/jw-03-2004/jw-0322-j2me.html**.

Index

- 3D Studio 292
- 3G services 232–64
- 3GPP format 91
- ABNF *see* Augmented
 - Backus–Naur
 - Format
- abstract classes, best
 - practice 326–7
- abstraction layers 331–3,
353–84
- acceleration sensors, DoJa
 - 255, 260–2
- acceptAndOpen()
 - method 380–1
- access points 119–20,
135–6, 418
- Access Point Name (APN)
418
- access restrictions, protected
 - APIs 59–77
- Accredited Symbian
 - Developer (ASD)
100
- active state, MIDlets
 - 29–34, 45–8,
79–81, 126–31,
161–3, 251,
284–303, 306–8,
311–12
- adaptive code uses, diversity
 - issues 150,
153–8
- addCommand() method
 - 37, 42, 45–8,
154–8
- addictive aspects, mobile
 - devices 6–7
- Application Descriptor Files
(ADF) 234–53
- Adobe Flash technology 3,
10–11, 18, 24,
141, 144, 253
- advanced multimedia
 - supplements
(ISR-234) 17, 20,
109–13, 141,
211–29
- Ages of Empire*** 266
- Alert object 37–8, 44–8,
252
- AlertType 38
- AllowedSender 78–81,
98–100
- AMS *see* Application
Management
Software
- Android platform 5,
11
- animations 51–2, 217–18,
225–6, 253–4,
258–9, 275–7,
281–303
 - see also* sprites
- APDU 373–6
- APDUConnection
 - 109, 112, 373,
375
- ApiInfo 107–13
- APIs
 - abstraction layers
331–3
 - concepts 10–15, 27,
34–56, 331–51,
353–84
 - DoJa 231–64
 - eSWT 115–22
 - Game 49–56, 101,
265–303
 - IAPInfo 119–20,
135–6
 - integration 353–84
 - Java ME Detectors Suite
107–13, 151,
190–5
 - MIDP GUI interfaces
34–56
 - Nokia UI 114–15

- APIs (*continued*)
 - SNAP Mobile 22, 120–1, 303, 416, 417–18, 422–31
 - SWT 115–22
 - Symbian OS 8, 17–18, 82, 83, 94, 100, 107–22, 141–4, 147, 176, 190, 331–51, 353–84
 - WSL 389
- APISDetector 190–1
- APN *see* Access Point Name
- APPARC 365–8
- Appearance object 294–7
- append() method 38, 44–8, 52–6, 76–7, 171–3, 309–10, 319–22
- Apple Dashboard Widgets 385
- Application Management Software (AMS)
 - see also* MIDlets
 - components 336–7
 - concepts 28–34, 331–7
 - installing a MIDlet 63
 - moving an application to the background 161–2, 251
 - starting an application 98–9, 161–2, 251
- Application Protocol Data Unit (APDU) 373–6
- arcade-style games 266
- ARM code 349–50
- ASD *see* Accredited Symbian Developer
- asynchronous Symbian OS operations 346–8, 360–2, 375–6
- Atic-Atac** 266
- ATOM feeds 389, 391–2, 401
- audio
 - see also* Mobile Media API; multimedia...; music
 - Doja 253–4, 259–60
 - Media API 58, 82–95, 152–3, 166–8, 287–8, 316–22
 - Melody For i-mode (MFi) format 253
 - MIDI playback 82
 - MusicMate application 214–18
 - playing 89–91, 253–4, 259–60, 369–73
 - usage survey 18
- AudioManager class 286
- AudioPresenter class 253–4
- Augmented Backus–Naur Format (ABNF) 84, 94
- authentication methods, signing process 63
- automatic object finalization 10, 13
- automatic save points, games 270–2
- AWT toolkit 35, 237
- BACK command type 154–8
- BasicSVGMenuScreen 289–302
- batteries 22, 25–6, 268, 284–303
- BBC News widget 388
- best practice 305–27
- bin folder 34, 238–9
- BinaryMessage class 96–100
- biometrics 255, 263
- Blackberries 6, 11
- blanket user permission 65–9
- Blender** 292–3
- Bluetooth
 - applications 215, 216, 221, 226, 227, 284, 297–302
 - concepts 22, 297–303
 - connection-opening methods 297, 300–2
 - data-exchange methods 301–2
 - deploying a MIDlet 109–11
 - discovery phase 298, 299–302
 - Doja 233–4, 236, 253
 - establishing a connection 184–6, 197–201
 - games 265, 297–303
 - inquiry phase 298–9
 - JSR-82 12, 20, 108, 112, 159, 211, 212, 297–302
 - Nokia 6600 4
 - properties 153
 - Remote Control 203
- BluetoothManager 286–302
- bootstrap() method 341–2
- BREW 333
- Browser object 118–22
- buffer I/O, best practice 314
- BufferPool object 311–12
- building MIDlets 30–4
- business cards 262
- ByteArrayInputStream object 315–16
- C++ 3, 8, 9–10, 17–18, 141–4, 147, 176, 190, 233, 333, 339, 344–6, 356–9, 361–2, 373–6
- C 3, 18, 341, 357, 375–6
- callbacks, event model 36–7

- cameras 5, 6–7, 18, 82–95, 218–20, 225–6, 238, 293–302, 308, 316–22, 369–73
 - see also* images; multimedia...
- CANCEL 156–8
- cancelInquiry 299–302
- Canvas 21–2, 36–7, 39–40, 42–9, 53–6, 91–5, 160–1, 163–6, 171–3, 245–50, 252, 280–303, 306, 365–9
- capabilities of Java ME on Symbian OS 123–5
- capture() method 85–95, 168, 316–22
- capturing, (JSR-135 MMAPi) 83, 92–5, 152–3, 168, 287–8, 316–22, 369–73
- Capuchin framework 24
- CartesianListener object 259–60
- CAs *see* certificate authorities
- cascading menus 25
- cascading style sheets (CSS) 385, 394–409
- catch 55–7, 73–4, 89–93, 129–31, 159–60, 250, 313–14, 320
- CBS 95–101
 - see also* Wireless Messaging API
- CCoeControl 367
- CDC 11, 13, 26, 28, 140–1, 210–11, 341
- CDMA 121, 135, 153, 232
- Cellulite** 240–4
- certificate authorities (CAs) 60–9
 - see also* security...
- Certificate Signing Request (CSR) 66–8
- certification exam, Symbian OS Java ME 27, 100
- Choice interface 41–2, 43–8
- ChoiceGroup object 37, 39, 41–2, 43–8
- Cipher API 109
- class of device (CoD) 298–302
- class files 10
- class libraries 13–14
 - see also* configuration...
- CLDC
 - concepts 11, 13–15, 16, 20, 27–8, 31–2, 34, 58, 81–2, 95–6, 139–45, 210–11, 232–55, 332–51
 - JSR-139 17, 108–13, 211–29
 - other run-time environments 139–40
- CLDC 1.0 13–14, 16, 20, 34, 81–2, 416–17
- CLDC 1.1 20, 151–3, 256, 412, 417–19
 - concepts 14, 81–2, 210–11
 - packages list 14
 - Symbian OS 81–2
- CLDC-HI HotSpot 19, 338–42, 346–50, 382–3
- client socket connections 71–3, 125–6
- close() method 86–95, 320–2, 380–1
- CMSMessageSignatureService 109–13
- CoD *see* class of device
- collidesWith() method 281–303
- collision detection 21, 276–303
- colors 35, 43, 47–8, 52, 171–3, 249
- Command... class 37–9, 41–8, 128–31, 135–6, 154–8, 252, 307–8
- CommandAction 47–8, 128–31, 307–8
- CommandListener interface 37, 41–8, 128–31
- commit() method 319–22
- CommsDat 135–6
- community features, SNAP Mobile 414–16
- com.nttdocomo... 238, 244–5
- comparators, RMS uses 314–16
- compatibility issues, Java ME 13
- compilation methods
 - JIT 338, 349–50
 - widgets 398–400
- compliance tests, SNAP Mobile 121, 413, 420
- Component JSRs 17, 20, 177–8, 190–5, 203–5, 210–30
- ComponentListener 246–8
- concrete classes 84
- configuration Java ME
 - component 12–14, 27–8, 34, 332–51
 - see also* CDC; CLDC...
- connected mobile games 266, 297–303, 411–31
- connection constraints 125

- URLConnection 78–81, 98–100
- connectivity 4, 13–14, 184–6
 - see also* Bluetooth
- Connector 58, 70–3, 78–81, 96–100, 120, 378–81
- constraints 22, 25–6, 105–6, 123–5, 214–15, 233–4, 235–8, 250, 255, 268–70, 273, 282–3, 308–9, 323, 326–7
- content aggregators 23
- content handler (JSR-211) 109–13, 140, 211–29
- content types,
 - getSupportedContentTypes() method 167–8
- ContentHandler 109–13
- Control 84–95
- Controller 284–303
- conventions 165–6
- CPU cycles 279, 284–5, 308–9, 323
- create() method 360–2
- createPlayer() method 84–95
- critique
 - Java ME 16–17, 19, 113–14, 149–50, 349–50
 - mobile devices 6–7, 149–50
- CRYPTO 112, 212
- CSIP... objects 379
- CSR *see* Certificate Signing Request
- CSS *see* cascading style sheets
- culling technique 269, 294
- CustomItem 36, 39–40, 42–8, 108–13, 163–6, 372
- databases 9, 15, 17, 56–8, 405
 - see also* Record Management System
- datagrams 58, 65, 70, 73–4, 253
- DataInputStream 315–16
- DataSource 84–6
- DateField 40, 43–8
- deallocate 86–95, 320–2
- DebugAgent 180–9, 192–9, 348–51
- debugging 16, 178, 180–9, 192–5, 198–201, 348–9
- decks, network operators 22–3
- defineCollisionRectangle 281–303
- defineReferencePixel 281–303
- delete 10, 38, 361–2
- deployment process
 - best practice 322–5
 - SNAP Mobile 413–14
- descriptors, widgets 394–5, 399–402
- design considerations
 - best practice 305–27
 - games 270–1, 278–9, 283–5
- desktop computers, mobile devices 35
- destroyAPP() method 29–34, 251, 307–8
- destroyed state, MIDlets 29–34, 45–8, 251, 284–303, 306–8
- detectAPIs() method 107–13
- detection using properties 150–3
- Detectors Suite application 107–13, 151, 190–5
- device 85–95
- device capabilities 107–13, 149–74
 - see also* mobile devices
- device conventions 165–6
- Device Explorer tool 200–1
- DeviceClass 298–302
- Diagnostics window, emulator 182–4
- Dialog 245–8, 252
- Digg widget 391
- Digital Set-Top Box Profile 14–15
 - see also* profiles
- digital signatures 59–69
- Direct3D 291–302
- directional keys 164–6
- discovery phase, Bluetooth 298, 299–302
- DiscoveryAgent 300–2
- DiscoveryListener 299–302
- dismiss() method 156–8
- Display() method 32–4, 37, 40, 117–18, 172–3, 319–22
- Displayable 37–8, 45–8, 161–3, 252
- displays 32–4, 35, 37–8, 40, 45–8, 117–18, 161–3, 169–74, 252, 284–303, 319–22
- dispose() method 117–18
- diversity issues 149–58
 - see also* fragmentation...
 - adaptive code uses 150, 153–8
 - detection using properties 150–3
 - flexible/modular design uses 150, 153–8

- handling approaches
 - 150–8, 162–74
- input mechanisms
 - 162–6
- multimedia
 - formats/protocols
 - 166–8
 - screens/displays
 - 169–74
- DLLs *see* dynamically linked libraries
- DNS records 308
- Doja
 - acceleration sensors
 - 255, 260–2
 - animation 258–9
 - Application Descriptor
 - Files (ADF)
 - 234–53
 - audio 253–4, 259–60
 - basics 234–8
 - Bluetooth 233–4, 236
 - Cellulite** 240–4
 - constraints 233–4, 235–8, 250, 255
 - Developers Guide 239, 253
 - dialogs 245–8, 252
 - documentation shortfalls
 - 240
 - double-buffering issues
 - 248–9, 252
 - DTV 260
 - Eclipse 231, 234, 240–3, 250, 255–6
 - electronic compasses
 - 260–1
 - example applications
 - 240–51
 - FeliCa 262–3
 - fingerprint sensors 255, 263
 - games 237–8, 240–4, 254
 - high-level API 244–8
 - historical background
 - 231–2
 - i-Appli 234–8, 254–64
 - information sources
 - 239–40
 - lifecycle differences
 - 251
 - low-level API 248–50
 - MIDP 231–8, 250–4
 - packages 238, 256–8
 - porting considerations
 - 250–4
 - project directory structure
 - 238–9
 - Scratchpad 236–8, 251–2
 - security issues 255, 263–4
 - serial communications
 - (UART) 260
 - sockets 253
 - softkeys 237–8, 246–50, 252
 - speech recognition 255, 263
 - text positioning
 - differences 252
 - tools 234–8
 - ToruCa 262
 - versions 235–6
- Doja 5.1 Profile 231–3, 254–64
- Doom** 266
- dot.coms 4
- Double 14
- double buffering 248–9, 252, 269, 280–81
- downloaded music, usage
 - survey 18
- drawString() method
 - 48, 249–50
- DTV, Doja 260
- dynamic orientation
 - switching
 - 173
- dynamically linked libraries
 - (DLLs) 25–6, 335–6, 345–6, 366–73
- Eclipse 115–22, 231, 234, 240–3, 250, 255–6
- ECom plug-in framework
 - 381–2
- EditEntryView()
 - method 245–8
- efficiency issues, Java
 - 10–12, 19–21, 338–9, 349–50, 382–4
- EKA2 kernel 232
- electronic compasses, Doja
 - 260–1
- Embedded Standard Widget Toolkit (eSWT)
 - 115–22
- empty method
 - implementations
 - 43–4
- emulator 30–1, 32–3, 34, 44–8, 56, 65–8, 91
 - see also* SDKs, Wireless Toolkit
 - concepts 175–205, 348–9
 - Diagnostics window
 - 182–4
 - features 182–4
 - MIDlets 32–3, 180–2, 190–5
 - Preferences window
 - 182–4
 - S60 platform 178–89, 420–1
 - SNAP Mobile 413, 417–18, 419–22, 429
 - UIQ 3 platform
 - 189–205
 - Utilities window
 - 182–4
 - WidSets 399–400
- encapsulation 9–10
- encryption processes,
 - security model
 - 59–69, 264, 325, 350–1

- Enterprise Java Beans 11
- enumerateRecords()
 - method 315–16
- error handling 139,
 - 245–6, 308,
 - 313–14
- ESock C++ API 346,
 - 373–6
- eSWT *see* Embedded Standard Widget Toolkit
- event handler 36–7, 42–8,
 - 428–30
- event model, LCDUI
 - 36–7, 42–3
- events, Symbian OS
 - 346–8, 360–2,
 - 375–6
- Events tab 182–3
- exception handling, best
 - practice 313–14
- EXE files 335–6, 349–50,
 - 351
- execution lifecycle, best
 - practice 322–5
- Exit option 37
- EXIT command type
 - 154–5
- feature packs (FPs) 179–89
- FeliCa, DoJa 262–3
- field of view (FOV) 269,
 - 293
- FileConnection (JSR-75)
 - 17, 20, 101,
 - 108–13, 144–5,
 - 168, 211–29,
 - 324–5, 359–62,
 - 383–4
- FileSystemListener
 - 361
- FileSystemRegistry
 - 360
- fillLayer() method
 - 54–6
- fillRect() method 43,
 - 47–8, 52, 275
- filters, RMS uses 314–16
- finally keyword 313–14
- Finding Higgs** 282–303
- fingerprint sensors 255,
 - 263
- first-person shooter (FPS)
 - 266
- five-way jog dials
 - 270–303
- Flash (Adobe) 3, 10–11,
 - 18, 24, 141, 144,
 - 253
- flexible design, diversity
 - issues 150,
 - 153–8
- Flickr viewer widget
 - 218–19, 223–4,
 - 399–409
- Float object 14, 108
- floating-point units (FPUs)
 - 22, 268, 269
- flushGraphics()
 - method 49,
 - 55–6, 280–303
- FOMA 232–64
 - see also* NTT...
 - 905i series of handsets
 - 254
- Font 40
- foreground/background
 - transitions, MIDlets
 - 161–2
- Form object 32–4, 37,
 - 38–42, 43–8, 77,
 - 306, 319–22
- Forum Nokia 188, 270–1,
 - 297–8, 387, 409,
 - 413–22
- FOV *see* field of view
- FPS *see* first-person shooter;
 - frames per second
- FPUs *see* floating-point units
- fragmentation of JSRs
 - see also* diversity issues
 - approaches to 150,
 - 158–74
 - concepts 16–17, 149,
 - 158–74, 210–11,
 - 222–3, 234
- graceful functional
 - degradation
 - approach 158–9
- NetBeans preprocessing
 - 173
- optionally replaceable
 - module approach
 - 160–61
- frame rate *see* frames per
 - second
- FramePositioning-
 - Control 87
- frames per second (FPS)
 - 22, 269, 271–2,
 - 278, 284, 297
- Frogger** 23
- Fujitsu
 - F905i 233, 255–6
 - F906i 255–6
- Full MSA 17, 177–8,
 - 190–5, 203–5,
 - 210–30
 - see also* Mobile Service Architecture...
- fundamentals of Java ME
 - MIDP
 - programming 26,
 - 27–101
- Galaga** 266
- Game & Watch** devices 5
- Game API 49–56, 101,
 - 265–303
- game development and
 - publishing process
 - DoJa 254
 - SNAP Mobile 120–1,
 - 303, 412–14
- GameCanvas 21–2, 36,
 - 42–8, 49–50,
 - 53–6, 165–6,
 - 274–303
- GameController
 - 284–303
- GameEngine 286–302
- GameMIDlet 286–302
- games
 - advanced 282–303

- automatic save points
 - 270–2
- bad practices 277–9
- Bluetooth 265,
 - 297–303
- concepts 5–6, 7,
 - 15–16, 21–2,
 - 42–3, 49–56, 68,
 - 101, 231, 237–8
- constraints 268–70,
 - 273, 282–3
- definition 266
- design considerations
 - 270–1, 278–9,
 - 283–5
- DoJa 237–8, 240–4,
 - 254
- Finding Higgs** 282–303
- frames per second (FPS)
 - 269, 271–2, 278,
 - 284, 297
- GhostPong** 237,
 - 273–9, 303
- graphics 265–6,
 - 272–303
- historical background
 - 5–6, 15–16
- information sources
 - 273, 297–8
- input mechanisms 270,
 - 280, 281–3
- level of detail (LOD)
 - 269, 272–3
- loops 266–8, 272,
 - 275–7, 284–303
- menus 285–7
- mesh concepts
 - 269–70, 272–303
- MIDP 2.0 concepts
 - 279–82
- multiplayer 266,
 - 297–303, 411–31
- navigation 283, 285–7
- pixels and polygons
 - 272–303
- pseudo-code game loop
 - 267–8
- research findings
 - 270–1
- settings 271
 - simple 273–9
- simulation rates 269,
 - 271–303
- SNAP Mobile 22,
 - 120–1, 188, 303,
 - 411–31
- start-up time 271
- terminology 268,
 - 269–70
- types 266
 - UI components 36
 - Widsets 387, 389
 - writing MIDP games
 - 265–303
- games class ID (GCID) 419
- GameScreen 286–302
- garbage collector (GC) 10,
 - 14, 307–8,
 - 311–12, 342
- Gauge object 40, 43–8
- GCF *see* Generic
 - Connection
 - Framework
- GCID *see* games class ID
- General Inquiry Access
 - Code (GIAC)
 - 298–302
- generated tones 83, 93–5
 - see also* multimedia...
- Generic Connection
 - Framework (GCF)
 - 58, 69–77, 95–6,
 - 120, 338, 353,
 - 360–2, 379–82
- geotagging MSA example
 - 218–20, 226–7
- getAddress() method
 - 97–100
- getAppProperty()
 - method 108–13
- GetByteArray-
 - Elements()
 - method 344
- getColor() method 40
- getConnectionUrl()
 - method 301–2
- getConstraints()
 - method 41
- getControl() method
 - 87–95, 319–22
- getDate() method 40
- getFont() method 40
- getGameAction()
 - method 43,
 - 280–303
- getGraphics() method
 - 49
- getHeight() method 42
- getInputWeight()
 - method 248
- getInteraction-
 - Modes() method
 - 163–4
- getKeyCode() method 43
- getKeyStates() method
 - 49, 276–7,
 - 281–303
- getMajorDevice-
 - Class() method
 - 298–302
- getMinContentWidth()
 - method 39
- getPayloadData()
 - method 97–100
- getPrefContentHeight
 - () method 39
- getProperty() method
 - 151–3, 167–8
- getSnapshot() method
 - 93, 95
- getState() method 86
- getSupportedContent-
 - Types() method
 - 167–8
- getSupported-
 - Protocols()
 - method 167–8
- getURL() method 120
- getWidth() method 42
- GhostPong** 237, 273–9,
 - 303
- GIAC *see* General Inquiry
 - Access Code
- GIF files 258
- GlobalManager 109
- good programming practices
 - 308–22

- Google
 - Android platform 5, 11
 - Gadgets 385
 - GMail 75
- Gosling, James 9, 12
- GPRS 4, 135
- GPS 7, 211–12, 308
 - see also* location
- GPUs *see* graphics processing units
- graceful functional degradation
 - 158–61
- graphics 36, 42–56, 237–8, 248–50, 256–9, 365–9
 - see also* images
- games 265–6, 272–303
- hardware acceleration
 - 350
- input mechanism considerations 166
- low-resolution 286–7
- Mobile 3D (JSR-184)
 - 15–17, 19–20, 22, 101, 109–13, 153, 160–1, 211–29, 265, 284–303, 350
- SVG (JSR-226) 17, 20, 24, 109–13, 125–6, 160–1, 195, 211–29, 265, 285–303, 350, 357
- vector 17, 20, 24, 109–13, 125–6, 155–8, 160–1, 187, 195, 211–29, 265, 285–303, 350, 357
- Graphics class 36, 42–8, 55–6, 252, 256–9, 289–302, 365–9
- graphics processing units (GPUs) 268, 269
- Graphics3D class 109–13, 256–9, 295–302
- GSM/UMTS phones 68–9, 82, 95, 99, 121, 135, 153, 373–6
- GUI 7–8, 24, 30–1, 34–56, 106–46, 149–50, 154–8, 162–74, 233–4, 335–6
 - see also* MOAP...; S60; UI; UIQ
- information sources
 - 145–6
- MIDP profile 34–56
- GUIControl 87–95
- hashmap-like structures
 - 405–6
- heads-up display (HUD)
 - 269
- heap 19–20, 124–5, 278, 326–7
- hideNotify 161–2
- Higgs Particles 282–303
- HTML 187, 256, 385, 389, 392, 402–9
- Http-Connection
 - 70–1, 75–7, 364
- HTTP(S) 15, 58, 64–5, 68–9, 70–1, 75–7, 85–95, 121, 219–20, 233–4, 236, 238, 243, 253, 254–5, 256, 313–14, 325, 353, 357, 363–4, 377, 416–17, 419
- HUD *see* heads-up display
- i-Appli 234–8, 254–64
 - see also* DoJa...
- i-mode handsets 233–4
 - see also* NTT DoCoMo
- IAP *see* Internet Access Points
- IAPInfo API 119–20, 135–6
- IApplication 244–8, 251
- IBluetoothObserver
 - 286–302
- IBM 115, 338
- icons 25, 38, 392–3
- ID 119–20
- IETF protocols 377–8
- Ikivo 24
- Image object 40, 46–8, 52
- image best practice
 - 312–13
- ImageItem 40–1, 43–8
- images 33–5, 40–1, 43–8, 52, 82–95, 187, 218–20, 225–6, 238–9, 258–9, 272–303, 311–13, 369–73
 - see also* cameras; graphics; multimedia...
- IMC *see* inter-MIDlet communication
- IMEI 121–2
- IMenuSupplier
 - 286–302
- imported packages 43–4
- IMPS *see* instant messaging and presence
- in-game features, SNAP Mobile 414–15
- independent software vendors (ISVs)
 - 61–9, 210
- Information Module Profile (IMP) 14–15
 - see also* profiles
- information sources
 - 145–6, 239–40, 273, 297–8
- inheritance hierarchies, best practice 326–7
- initDisplayMode 91–2
- input mechanisms
 - diversity handling issues 162–6
 - games 270–303

- mobile devices 35,
 - 106–13, 162–6,
 - 237–8, 246–50,
 - 270–303, 306
- platform input
 - mechanisms 166
- user experiences 306
- InputStream 70–3,
 - 76–7, 84–95
- InputStreamReader 310
- inquiry phase, Bluetooth 298–9
- insert 38
- Installs 182
- instant messaging and
 - presence (IMPS) 121, 416–31
- integration of Java ME and
 - Symbian OS APIs 353–84
- intellectual property rights 234
- inter-MIDlet communication (IMC) 25–6, 229
- inter-process
 - communication simulation 140–5, 335–51
- interfaces, best practice 326–7
- internationalization (JSR-238) 109–13, 211–29
- Internet 17, 20, 71–3,
 - 108–13, 119–20,
 - 135–6, 211–29,
 - 342–3, 359,
 - 362–4
- Java applets 10–11
- SNAP Mobile 303,
 - 411–31
- usage survey 18
- WidSets 385–409
- Internet Access Points (IAP) 119–20, 135–6, 418
- interpreted languages 10,
 - 19, 338, 349–50
- INVITE String 379–80,
 - 383
- IOException 139,
 - 290–302
- IP addresses 71–3, 78–9
- Ipc... 141–3
- IPC *see* inter-process communication...
- iPhone 5–6
- iPods 5, 6
- Israel 233
- ISVs *see* independent software vendors
- Item object 38–42, 43–8
- ItemCommandListener
 - object 38–9, 42
- ItemStateListener
 - object 42
- J2SE *see* Java 2 Standard Edition, Java SE
- JAD files 55–6, 95,
 - 169–70, 173, 182,
 - 234–8, 418–19,
 - 422–31
- attributes 33–4, 122–3
- concepts 26, 28, 33–4,
 - 62–4, 66–8, 78–9,
 - 98–100, 107–8,
 - 161–3
- examples 34
- security model 62–4,
 - 66–8
- JAM *see* Java Application Management
- jam files 239–40
- Japan 3, 15, 231–64
- JAR files 28, 33–4, 40, 47,
 - 55–6, 60, 62–4,
 - 98, 107–13,
 - 123–5, 155–8,
 - 159–61, 168–9,
 - 173–4, 182,
 - 214–22, 234–8,
 - 250, 255, 273–7,
 - 282, 287, 288–92,
 - 319, 322, 323,
 - 326–7, 418–19
- Java 2 Standard Edition (J2SE) 31
- Java
 - characteristics 9–10
 - concepts 9–12
 - efficiency issues 10–12,
 - 19–21, 338–9,
 - 349–50, 382–4
 - four core technologies 11–12
 - security characteristic
 - 9–10, 13, 17, 20,
 - 30–4, 57, 58,
 - 59–81, 94–5,
 - 99–101, 109–13,
 - 130–5, 211–29,
 - 232–3, 239–40,
 - 255, 263–4,
 - 323–5, 336–7,
 - 350–1, 359,
 - 373–6
- Java applets, concepts 10–12
- Java Application Management (JAM) 238–9
- Java Community Process (JCP) 12–13, 82, 114, 119
- Java Debug Wire Protocol (JDWP) 348–9
- Java EE technology 11, 325
- Java Event Server (JES) 346–7, 350, 361–2, 375–6
- Java Hashtable 405
- Java layer, MIDP 343–4
- Java ME
 - see also* best practice; DoJa
 - abstraction layers 331–3
 - API selection 114
 - appropriate uses 18–19, 25–6
 - asynchronous Symbian OS operations 346–8, 360–2, 375–6

- Java ME (*continued*)
 - bumpy start 16
 - compatibility issues 13
 - components 12–17, 27–8, 332–51
 - computing capabilities on Symbian OS 123–5
 - concepts 3–26, 105–47, 232–3, 305–27, 331–51, 353–84
 - configuration component 12–14, 27–8, 34, 332–51
 - connection constraints 125
 - constraints 22, 25–6, 105–6, 123–5, 214–15, 233–4, 235–8, 250, 255, 268–70, 273, 282–3, 308–9, 323, 326–7
 - core logical entities 332–6
 - critique 16–17, 19, 113–14, 149–50, 349–50
 - debugging 16, 178, 180–9, 192–5, 198–201, 348–9
 - Detectors Suite 107–13, 151, 190–5
 - Developer Library 169, 187–8, 303
 - diversity issues 149–74
 - error handling 139, 245–6, 308, 313–14
 - fragmentation of JSRs 16–17, 101, 149, 158–74, 210–11, 234
 - fundamentals of programming 26, 27–101
 - future prospects 24–6, 227–9, 234
 - games environment 21–2, 265–303
 - historical background 4–6, 11–13, 16, 21–2, 232
 - inconsistent implementations 113–14
 - information sources 145–6
 - integration of Java ME and Symbian OS APIs 353–84
 - inter-process communication simulation 140–5, 335–51
 - interpreted languages 10, 19, 338, 349–50
 - Java SE 12–13, 237, 341–2
 - JIT 338, 349–50
 - launching methods 111–13, 336–7
 - management on devices 131–8
 - management of specifications 12–13
 - multitasking benefits of Symbian OS 126–31, 279, 325–6
 - native capabilities of Symbian OS 139–41, 353–84
 - Nokia UI API 114–15
 - optimization techniques 326–7
 - optional packages component 12, 15–17, 27–8, 100
 - Platform concepts 9–12, 27–8, 332–51
 - popularity 6, 11–12
 - POWER acronym 7
 - pre-verification concept 13, 30–4
 - profiles component 12, 14–15, 27–8, 34, 231–3, 332–51
 - reduced functionality 12–13
 - responsiveness factors 306–8
 - routes to market 22–3
 - run-time environment 333–51
 - SDKs 31, 109, 115, 131, 147, 175–205, 388–400, 413–31
 - SNAP Mobile 22, 120–1, 188, 303, 411–31
 - subsystem architecture 331–51, 353–84
 - support issues 23
 - Symbian OS 3, 17–21, 24–6, 27, 69–77, 82, 100, 105–47, 149–50, 161–2, 166–8, 175–205, 227–9, 282–303, 313–14, 325–7, 331–51, 353–84
 - Symbian OS API mappings 351, 353–84
 - telephony information retrievals 121–2
 - tools on Symbian OS 131, 175–205
 - user experiences 305–8, 327
 - WidSets 386–409
 - WTK 30–1, 34, 44–8, 56, 65–8, 91, 175–7, 189, 195–201, 234, 279, 420–1, 429
- Java Native Interface (JNI) 13, 140–5, 339–41, 343–4

- Java Objects heap
 - 124–5
- Java Platform, Micro Edition
 - see Java ME
- Java SE 11, 12–13, 237, 341–2
- Java Specification Requests (JSRs)
 - see *also* fragmentation... ; JSR-...
 - concepts 12–13, 15–17, 19–21, 25–6, 81–2, 83, 94, 100, 107–13, 146–7, 151–3, 158–74, 211–29, 282–303, 345–51, 353–84
 - games on Symbian OS 282–303
 - integration of Java ME and Symbian OS APIs 353–84
 - Java ME Detectors Suite
 - example 107–13, 151, 190–5
 - supported JSRs 82, 83, 94, 100, 107–13, 146–7, 151, 152–3, 159–61, 190–5, 282–303, 345–6, 351, 353–84
 - Symbian OS API
 - mappings 351, 353–84
 - Symbian OS support
 - 82, 83, 94, 100, 107–13, 146–7, 151, 190–5, 282–303, 345–6
- Java Technology for the Wireless Industry (JTWI) 16–17, 20, 68–9, 81–3, 95–101, 158, 210, 312
- Java Verified issues, MIDlet 23, 30–1
- Java virtual machine (JVM)
 - 10–14, 310–11, 331–51, 361–2, 375–6
- JavaCardRMICConnection 109–13
- java.io 14
- java.lang 14
- JavaScript 385–409
- JavaScript Object Notation (JSON) 224–5
- javax packages 13, 14, 28–34
- javax.microedition.io 14, 96
- javax.microedition-
 - .lcdui... 32–4, 36, 41–3, 47, 49, 68, 92
 - .microedition-
 - .media 84
 - .midlet.MIDlet 28–34
 - .rms 57
- JAXRPCException 108–13
- JBenchmark application 188
- JDWP see Java Debug Wire Protocol
- JES see Java Event Server
- JIT see Just-In-Time compiler
- JNI see Java Native Interface
- JPEG files 16–17, 82, 312–13, 320
- JSON see JavaScript Object Notation
- JSR-75 Files & PIM 17, 20, 101, 108–13, 144–5, 168, 211–29, 324–5, 359–62, 383–4
- JSR-82 Bluetooth API 12, 20, 101, 108–13, 153, 159–60, 211–29, 253, 265, 284–303
- JSR-118 MIDP 2.0 16–17, 58, 81–3, 108–13, 115, 209–13, 359, 369–73
- JSR-120 Wireless Messaging API (WMA) 16, 20, 81–2, 95–101, 107–13, 127–31
- JSR-124 Java EE Client Provisioning Specification 325
- JSR-135 Mobile Media API (MMAPI) 15–17, 19–20, 58, 81–95, 100, 108–13, 152–3, 166–8, 211–29, 253–4, 265, 287–303, 316–22, 353, 359, 369–73, 384
- JSR-139 CLDC 17, 108–13, 211–29
- JSR-172 Web Services 17, 20, 108–13, 211–29, 342–3, 359, 362–4
- JSR-177 Security & Trust (SATSA) 17, 20, 109–13, 211–29, 359, 373–6
- JSR-179 Location 15–17, 20, 109–13, 212–29
- JSR-180 SIP API 17, 20, 109–13, 211–29, 359, 377–84
- JSR-184 Mobile 3D Graphics 15–17, 19–20, 22, 101, 109–13, 153, 160–1, 211–29, 265, 284–303, 350
- JSR-185 JTWI 16–17, 20, 68–9, 81–3
- JSR-205 Messaging 17, 20, 100, 109–13, 195, 211–29
- JSR-211 CHAPI 109–13, 140, 211–29

- JSR-226 SVG 17, 20, 24, 109–13, 125–6, 160–1, 195, 211–29, 265, 285–303, 350, 357
- JSR-229 Payment 109–13
- JSR-234 AMMS 17, 20, 109–13, 141, 211–29
- JSR-238 Internationalization 109–13, 211–29
- JSR-248 Mobile Service Architecture (MSA) 17, 20, 24–6, 100, 107–13, 209–30, 359
- JSR-249 MSA 2.0 210, 227–9
- JSR-258 Mobile User Interface Customization 229
- JSR-271 MIDP 3.0 227–9, 322
- JSR-297 Mobile 3D Graphics API 2.0 22
- JSR-307 Network Mobility and Mobile Data 229
- JSRsDetectorMIDlet 107–13, 151–3, 159–61
- JSRsView 107–13
- JTWI *see* Java Technology for the Wireless Industry
- Just-In-Time compiler (JIT) 338, 349–50
- JVM *see* Java virtual machine
- JVM::run 341
- Kauai virtual machine (KVM) 12–13, 19, 338–9, 348–9, 351
- KDDI 231–2
- KDWP 348–9
- Kernel 349
- key codes 39, 42–3, 49, 253, 276–303
- keypads 35, 39, 42–3, 49, 253, 276–303, 306
- keyPressed() method 39, 42–3, 49, 253, 280–303, 307–8
- keyReleased() method 281
- keyRepeated() method 281
- KeyState 276–7, 281
- kiosk mode 25
- KJNI layer 339–40, 344, 346, 361–2, 375–6
- KNI interface 339–40
- KVM *see* Kauai virtual machine
- Large Hadron Collider in CERN 282–3
- launching methods, Java ME applications 111–13, 336–7
- Layer 49, 50–1, 52–6
- layered model of Symbian OS 8
- LayerManager 49, 51, 52–6, 276–7
- LCDUI 100, 154–66, 387, 402–3, 405–6
- categories 35–7, 364–9
- concepts 35–48, 68, 91–2, 115, 119–22, 154–5, 196, 216–18, 285–303, 350–1, 364–9, 384
- event model 36–7, 42–3
- Game API 49–56, 101, 265–303
- High-Level API 35–42, 43–8, 119, 163, 364–9
- interfaces 41–2
- Low-Level API 35–7, 42–8, 49–56, 119, 163–6, 364–9
- MenuCanvas 47–8
- Symbian OS 364–9, 384
- UIExampleMIDlet example 43–8
- level of detail (LOD) 269, 272–3
- LG 6, 8
- see also* S60
- LIAC *see* Limited Inquiry Access Code
- LIBlets 25, 229, 322
- lifecycle guidelines, MIDlets 28–34, 126–31, 161–3, 251, 270, 284–303, 306–8, 311–12, 336–7
- lightweight libraries, best practice 322
- lightweight threading (LWT) 340–1
- Limited Inquiry Access Code (LIAC) 298–302
- Linux 7, 31, 333
- List object 37, 38, 39, 42, 43–8, 306
- Loader class 293–7
- LocalDevice 108–13, 299–302
- localhost 141–2
- location (JSR-179) 15–17, 20, 109–13, 212–29
- see also* GPS
- lock 248–50, 252
- LOD *see* level of detail
- low-resolution graphics 286–7
- LWT *see* lightweight threading
- LWUIT 178
- M3G *see* Mobile 3D Graphics
- magic numbers 278

- Majinate 100
- management on devices,
 - Java ME on
 - Symbian OS 131–8
- Manager 84–95, 108–13
- manufacturers
 - major 6, 231–3
 - SDKs 175–6, 189
 - trusted domain 69
- MApplication 237–8
- Mascot plug-in 237, 255–6
- massively multiplayer,
 - online, role-playing games (MMORPGs) 266, 297–303, 411–31
- Material object 294–7
- Maze Racer** 419–22, 431
- MD5 message digest 224–5
- Media API 58, 68, 82–3
 - see also* Mobile Media API
- media locator 84–6
- media players 83–95, 320–1, 369–73, 387
 - see also* Mobile Media API
- media streaming 83–95, 320–1, 377–8
- Melody For i-mode (MFi)
 - audio format 253
- memory 4, 13–14, 35, 124–5, 308–9
 - see also* heap; RAM; ROM
- MenuCanvas 47–8
- menus 25, 285–7
- meshes 269, 271, 272, 291–7
- message-passing paradigm 9–10
- MessageConnection 96–100, 109–13
- MessageListener 96–100, 127–31
- MetaDataControl 87–95
- MFi audio format 253
- Microsoft Windows
 - File Explorer 110
 - Live Gallery 385
 - Mobile 11, 178, 333
 - Pocket PC 11
 - XP 31
- MIDI playback 16–17, 82, 253–4, 270, 287–8, 371–3
- MIDIControl 87–95
- MIDlet... 32–4
- MIDlet suites 33–4, 57–69, 95, 106–46
- MIDletClassName 78, 98
- MIDlets 6, 57–69, 106–46
 - see also* Application Management Software
- active state 29–34, 45–8, 79–81, 126–31, 161–3, 251, 284–303, 306–8, 311–12
- audio playing 89–91, 253–4
- background/foreground transitions 161–2
- best practice 305–27
- building 30–4
- concepts 15, 19–20, 23, 25–6, 28, 57–8, 100, 106–13, 123–46, 161–2, 180–2, 190–5, 305–27, 334–51
- constraints 123–5, 214–15, 236–7, 250, 282–3, 308–9, 323, 326–7
- definition 28
- destroyed state 29–34, 45–8, 251, 284–303, 306–8
- development methods 30–1, 109–13
- emulator 32–3, 180–2, 190–5
- eSWT 115–22
- event model 36–7, 42–3
- example applications 31–4, 51–2, 53–6, 106–13, 128–31, 285–303
- future prospects 25–6
- games on Symbian OS 285–303
- installation/removal on Symbian OS 136–7, 140, 190–5, 336–7
- Java Verified issues 23, 30–1
- lifecycle guidelines 28–34, 126–31, 161–3, 251, 270, 284–303, 306–8, 311–12, 336–7
- MenuCanvas 47–8
- monitoring 30–4
- multitasking benefits of Symbian OS 126–31, 279, 325–6
- native capabilities of Symbian OS 139–41
- NetworkDemo example 75–7
- other run-time environments 139–40
- packaging 30–4
- paused state 29–34, 46–8, 126–31, 161–3, 251, 270–1, 277–303, 306–8

- MIDlets (*continued*)
 - Push Registry API 58, 65, 78–82, 96, 98–100, 126, 337–8, 383
 - running 30–3, 55–6, 106–13, 128–31, 180–2, 190–5, 337–8
 - S60 emulator 180–2
 - SDKs 180–2, 190–5
 - security settings 132–5
 - signing process 30–4, 57–77, 95, 99, 232–3, 323–5
 - size constraints 123–5, 214–15, 236–7, 250, 255, 282–3, 308–9, 323, 326–7
 - states 28–34, 45–8, 79–81, 126–31, 161–3, 251, 270, 277–303, 306–8, 311–12
 - trusted MIDlet suites 59–77, 80–1, 95, 99, 133–5, 323–5
 - UIExampleMIDlet example 43–8
 - UIQ emulator 190–5
 - untrusted MIDlets 59, 65, 68–77, 80–2, 95, 99, 323–5
 - using 28–34, 63
 - versions 33–4
 - video playing 90–2, 258–9
- MIDletStateChanged-Exception 30, 307–8
- MIDP 1.0, concepts 15, 20, 21, 34, 114, 222–7, 265
- MIDP 2.0 (JSR-118) 56–101, 108–13, 151–3, 386–409, 412, 417–19
 - concepts 4–5, 11–12, 15, 16–17, 20, 21–2, 34, 56, 81–2, 95, 98, 100, 101, 112, 115, 126, 130–1, 158, 209–13, 222–7, 273–303, 333–51, 359, 369–73
 - Game API 49–56, 101, 265–303
 - historical background 4–5, 15, 20–1
 - JTWI compliance 16–17, 68–9, 81–3, 100–1, 158
 - MIDP 2.1 112
 - MIDP 3.0 24–6, 227–9, 322
 - MIDP
 - see also* LCDUI; MIDP versions; profiles; Record Management...
 - concepts 4–5, 14–15, 16–17, 20–1, 34–101, 108–13, 115, 119–22, 130–5, 209–13, 222–7, 231–8, 250–4, 265–303, 312–13, 332–51, 359, 369–73, 377–8
 - Doja 231–8, 250–4
 - fundamentals of Java ME MIDP programming 26, 27–101
 - games 49–56, 101, 265–303
 - GCF 58, 69–77, 95–6, 120, 338, 353, 360–2, 379–82
 - GUI application development 34–56
 - implementation layer 342–6
 - introduction 27–8
 - Java layer 343–4
 - JNI layer 343–4
 - legacy MIDP applications 222–7
 - Media API 58, 82–3
 - networking APIs 58, 69–77
 - non-GUI APIs 56–8
 - Push Registry API 58, 65, 78–82, 96, 98–100, 126, 335–8, 383
 - security model 57, 58, 59–69, 94–5, 130–1, 132–5, 350–1
 - Symbian C++ layer 343, 344–6
 - Symbian OS 130–1, 232–3, 343, 344–6, 359, 369–73, 377–8
 - Symbian Signed 130–1, 232–3
 - MIME types 92–3
 - MMA 82
 - MMAPI *see* Mobile Media API (JSR-135)
 - MMF *see* Multimedia Framework
 - MMID... 366–7
 - MMORPGs *see* massively multiplayer, online, role-playing games
 - MMS 7, 100
 - MOAP(S) 3, 7–8, 11, 205, 233–4
 - Mobile 3D Graphics API 2.0 (JSR-297) 22
 - Mobile 3D Graphics (JSR-184) 15–17, 19–20, 22, 101, 109–13, 153, 160–1, 211–29, 265, 284–303, 350
 - mobile devices
 - addictive aspects 6–7

- capabilities 107–13, 149–74
- concepts 3–14, 145–6, 149–74, 231–3, 308–9
- constraints 22, 105–6, 123–5, 214–15, 233–4, 235–8, 250, 255, 268–70, 273, 282–3, 308–9, 323, 326–7
- critique 6–7, 149–50
- desktop computers 35
- diversity issues 149–58, 162–74
- fragmentation of JSRs 16–17, 101, 149, 158–74, 210–11, 222–3
- future prospects 24–6, 227–9, 234
- information sources 145–6, 239–40, 273, 297–8
- input methods 35, 106–13, 162–6, 237–8, 246–50, 270–303, 306
- major manufacturers 6, 231–3
- power constraints 22, 25–6, 105–6, 123–31, 268, 284–303, 308–9
- privacy concerns 6–7
- social change 6–7, 26, 308–9
- statistics 5–6, 231–3
- survey of usage 18
- typical specifications 35, 308–9, 320
- uses 5, 6–7, 17–18, 25–6, 308–9
- mobile games 266, 268–303, 411–31
 - see also* games
- mobile generation 5, 6–7
- Mobile Media API (JSR-135) 15–17, 19–20, 58, 81–95, 100, 108–13, 152–3, 166–8, 211–29, 253–4, 265, 287–303, 316–22, 353, 359, 369–73, 384
 - see also* audio; generated tones; video
 - architecture 83–5
 - best practice 316–22
 - capturing 83, 92–5, 152–3, 168, 287–8, 316–22, 369–73
 - concepts 81–95, 108–13, 152–3, 166–8, 211–29, 253–4, 265, 287–303, 353, 359, 369–73, 384
 - obtaining media content 84–6
 - playing 83, 86–95, 253–4, 287–303, 316–22, 369–73
 - recording 83–95, 168, 316–22, 369–73
 - security model 94–5
 - Symbian OS 83, 94, 108–13, 166–8, 287–303, 353, 359, 369–73, 384
 - system properties 317–18
- mobile offices 6–7
- Mobile Service Architecture (JSR-248 MSA) 17, 20, 24–6, 100, 107–13, 126, 158, 177–8, 209–30, 265, 359
 - see also* Java Specification Requests
- benefits 209–13
- concepts 17, 209–30, 359
- example applications 214–22
- Full MSA 17, 177–8, 190–5, 203–5, 210–30
- geotagging example 218–20, 226–7
- legacy MIDP applications 222–7
- MobileAerith 222–3, 226–7
- MSA 2.0 24–6, 210–11, 227–9
- MSA Subset 17, 20, 210–30
- MusicMate example 214–18, 230
- radio tuner example 220–1
- Symbian OS 229–30, 359
 - uses 213–22
- mobile TV, usage survey 18
- MobileAerith 222–3, 226–7
- model–view–controller (MVC) 160–1, 285–303
- modular design, diversity issues 150, 153–8
- modular design, Symbian OS 8
- monitoring, MIDlets 30–4
- Monty VM 338, 349
- Motorola 6
 - MOTO Z8 202
 - MOTO Z10 202, 203–5
 - MOTODEV Studio 201–5
 - Z10 189
- MP3 players 5, 6, 18
 - see also* music
- MPEG files 91–2

- MSA *see* Mobile Service
 - Architecture (JSR-248)
- MSA Subset 17, 20, 210–30
- multimedia applications
 - 15–17, 19–20, 58, 81–95, 108–13, 152–3, 166–8, 211–29, 270–303, 316–22, 353–84, 389
 - see also* audio; generated tones; images; Mobile Media API; video
- concepts 81–95, 211–29, 270–303, 316–22, 353–84, 389
- diversity handling issues 166–8
- games 270–303
- Symbian OS 83, 94, 108–13, 166–8, 287–303, 353–84
- WidSets 389
- Multimedia Framework (MMF) 166–8, 369–73
- multiplayer games 266, 297–303, 411–31
- multitasking benefits, Symbian OS 126–31, 279, 325–6
- multithreading characteristic of Java 9–10, 124–5
- music 18, 214–18, 287–303
 - see also* audio
 - usage survey 18
- MusicMate MSA example 214–18, 230
- MVC *see* model–view–controller
- MySQL database
 - management 253, 254–5, 256, 313–14, 325, 353, 357, 363–4, 377, 416–17, 419
 - system 9
- MyThread 307–8, 311–12
- N-Gage 21, 411–12
- native capabilities of Symbian OS 139–41, 353–84
- navigation considerations, games 285–303
- Navigator 286–302
- Net Access function group 75
- NetBeans
 - concepts 107–13, 173–4, 178, 179–80, 184–6, 188–9, 192–5, 198–201, 217–18, 223–6, 255–6
- fragmentation of JSRs 173
- SDKs 178, 179–80, 184–6, 188–9, 192–5, 198–201, 217, 223–6
- Update Center 223–4
- Visual Designer 217–18, 225–6
- network operators 22–3, 231–3
- NetworkDemo example 75–7
- networking
 - concepts 68–81
 - datagrams 58, 65, 70, 73–4, 253
 - examples 75–7
 - GCF 58, 69–77, 95–6, 120, 338, 353, 360–2, 379–82
 - HTTP(S) 15, 58, 64–5, 68–9, 70–1, 75–7, 85–95, 121, 219–20, 233–4, 236, 238, 243,
- 253, 254–5, 256, 313–14, 325, 353, 357, 363–4, 377, 416–17, 419
- NetworkDemo example 75–7
- Push Registry API 58, 65, 78–82, 96, 98–100, 126, 383
- security policy 75, 130–1
- serial ports 70, 260
- servers 58, 65, 70–3, 125–6, 230
- sockets 58, 65, 70, 71–3, 125–6, 230
- Symbian OS 69–77, 356–7
- new keyword 10, 29–34, 54–6
- nextFrame() method 281–2
- Nokia 24, 37, 122–3, 165, 169–70, 317–18
 - see also* S60
- 3650 100
- 5800 XpressMusic 106–13, 132–46, 162–3
- 6600 4, 5, 20, 101, 114, 320, 338
- Energy Profiler 177, 188
- Eseries 3
- games research 270–1
- N-Gage 21
- N70 20
- N95 5, 75, 77, 106–13, 115, 132–46, 151–8, 179–89, 324
- N96 20, 101, 179–89, 308–9
- Nseries 3
- S40 models 11, 386
- Series 40 3rd Edition 317
- SNAP Mobile 22, 120–1, 303, 411–31

- Symbian OS 3
- Nokia... 122–3, 165, 169–70
- notify() method 347–8
- notifyBlocking-Operation-Completed() method 347–8
- NotifyChange() method 361–2
- notifyDestroyed() method 30, 45–8
- notifyPaused() method 29
- NTT DoCoMo 3, 7–8, 11, 15, 231–64
 - see also* DoJa; MOAP...
- nttdocomo... 238, 244–5, 248–9, 256–8
- numerical keypads 35

- OBEX 112, 236, 238, 243, 256
- obfuscators 250, 323
- object-orientation
 - characteristic of Java 9–10, 100
- Object3D 293–7
- objects
 - best practice 311–12
 - garbage collector 10, 14, 307–8, 311–12, 342
 - images 311–13
 - pools 311–12
- ODD *see* on-device debugging
- on-device debugging (ODD)
 - 178, 184–6, 198–201
- oneshot user permission 65–9
- online games *see* connected mobile games
- open source 7–8, 18, 325–6, 332–3

- OpenGL ES 19–20, 22, 257–9, 292–3
- Operator trusted domain 69
- optimization techniques 326–7
 - see also* performance issues
- optional packages Java ME
 - component 12, 15–17, 27–8, 100
 - see also* Java Specification Requests
- optionally replaceable module approach, fragmentation of JSRs 158–61
- OTA enhancements 26, 325
- out-of-game community features, SNAP Mobile 414, 415–16
- OutputStream 71–3, 76–7

- package 32–4, 44–8, 229
- PacMan** 266
- paint() method 36–7, 39–40, 42–3, 47–8, 52–4, 280–303
- Panel 245–8
- PANPOT 259–60
- pauseApp() method 29–34, 46–8, 126–31, 161–3, 251, 307–8
- paused state, MIDlet 29–34, 46–8, 126–31, 161–3, 251, 270–1, 277–303, 306–8
- PC video cards 272
- PDA's 5, 11–12, 13–14, 216–18

- performance issues 10–12, 19–21, 338–9, 349–50, 382–4
 - see also* optimization techniques
- permissions
 - modes 65–9
 - protected APIs 59–77, 80–1, 95, 99, 239–40, 323–5, 350–1
- Phoenix** 266
- PhoneMe project 332–3
- photos folder 152–3
- pick() method 296–7
- PIM 17, 20, 101, 108–13, 152–3, 202–3, 211–29, 236, 324–5
- PIN operations 373–6
- pipe function 314
- PitchControl type 87–95
- pixels and polygons, games 272–303
- PKI 59–69
- planning process, SNAP Mobile 412–13
- platform 151–3
- platform input mechanisms 166
- platformRequest() method 118–22
- Player 84–95, 287–302, 316–22, 369–73
- PlayerListener 84–95, 287–302
- PlayerUpdate() method 90, 288–302
- playing
 - audio 89–91, 253–4, 259–60, 369–73
 - Mobile Media API (JSR-135 MMAPi) 83, 86–95, 253–4, 287–303, 316–22, 369–73
 - video 90–2, 258–9, 316–22

- PNG files 47, 124, 274–7, 312–13
- podcasts 7
- pointerDragged 308
- pointers 42–3, 163–6, 306, 308
- PolygonMode 294–7
- pools of objects 311–12
- portability 9–10, 250–4
- POSIX 3, 18, 333
- POWER acronym 7
- power constraints 22, 25–6, 105–6, 123–31, 268, 284–303, 308–9
- pre-verification concept 13, 30–4
- Preferences window, emulator 182–4
- prefetch 86–95, 320–1
- Presence feature, SNAP Mobile 416, 426–30
- prevFrame 281–303
- primitive types 9
- privacy concerns, the mobile generation 6–7
- private keys 59–61, 65–8
- processEvent 249–50, 253
- processor power 13–14, 35
- profiles 151–3
- profiles Java ME component 12, 14–15, 27–8, 34–56, 231–3, 332–51
 - see also* MIDP...
- ProGuard 250, 323
- properties, detection
 - diversity using properties 150–3
- proprietary solutions, screen/display
 - diversity issues 169–70
- protected APIs 59–77, 80–1, 95, 99, 239–40, 323–5, 350–1
 - see also* security...
- protected domains 59, 64–9
- protocols, getSupported-Protocols 167–8
- pseudo-code game loop 267–8
- pseudo-grammars 5
- Psion Organiser 5
 - see also* Symbian OS
- public keys 59–61, 65–8
- published widgets 392–3, 408
- publishing process, SNAP Mobile 412–14
- Push Registry API 58, 65, 78–82, 96, 98–100, 126, 335–8, 383
- PushRegistry 78–81
- Python 3, 18, 144, 292–3
- Quake** 266
- QWERTY keyboard 35
- radio tuners 220–1, 260
- RAM 4, 268, 279, 308
 - see also* memory
- RateControl 87–95, 371–3
- Rawsocket.org widget 392
- RayIntersection 296–7
- RChunk 349
- real-time strategy (RTS) 266
- Real-Time Streaming Protocol (RTSP) 85–95, 320–1, 377–8
- realize() method 86–95, 320–2
- receive() method 97–100
- receiving messages, Wireless Messaging API (JSR-120 WMA) 97–8
- Recommended Security Policy 68–9, 95, 99
- Record 57–8, 168
- Record Management System (RMS) 15, 17, 101, 202–3, 405
 - best practice 314–16
 - concepts 56–8, 68, 100, 168, 236, 251–2, 319
- RecordComparator 316
- RecordControl 87–95, 316–22, 371–2
- RecordEnumeration 315, 316
- recording (JSR-135 MMAPI) 83–95, 168, 316–22, 369–73
- RecordStore 57–8, 315–16
- reflection functionality 13
- refresh rate 269
- registerConnection() method 78–81
- remote portals, best practice 322–3
- repaint 37, 52, 280–303
- resilience expectations, user experiences 308
- ResourceManager 109–13, 215–18
- responsiveness factors, user experiences 306–8
- resume 251
- RFC... 377–8
- RFile 360–2
- RFs 360–2
- rich widgets 393–400
- RIM libraries 11
- ringtones 5, 266

- RMS *see* Record
 - Management 178–89, 233, 312–13, 336
 - System emulator 178–89, 420–1
- robustness characteristic of
 - Java 9–10
- role-playing games (RPGs) 266
- ROM 331
- Route tab 182–3
- routes to market, Java ME 22–3
- RPGs *see* role-playing games
- RSA-SHA1 signing
 - algorithm 62–3
- RSocket 373–6
- RSS feeds 389, 391–2, 401
- rtp 85–95
- RTS *see* real-time strategy
- RTSP *see* Real-Time Streaming Protocol
- run() method 36–7, 55–6, 76–7, 89–95, 129–31, 267–8, 341
- running MIDlets 30–3, 55–6, 106–13, 128–31, 180–2, 190–5, 337–8
- running widgets 398–400
- S60
 - see also* Nokia
 - 2nd Edition 317, 320
 - 3rd Edition FP1 179–80
 - 3rd Edition FP2 115, 119–23, 133–5, 149, 153, 161–2, 169–74, 176–7, 178–89, 220, 229–30, 338–40, 348–9
 - 5th Edition 115, 122–3, 149, 153, 161–3, 169–74, 176–7, 178–89, 229–30
 - concepts 7–8, 39, 90, 100, 106–13, 145–6, 151–3, 178–89, 233, 312–13, 336
 - emulator 178–89, 420–1
 - JAD attributes 122–3
 - SDKs 178–89
 - WRT widgets 386
- Samsung 6, 8
 - see also* S60
- SATSA 112, 213–16, 221, 224–7, 359, 373–6
- save point 269, 270
- Scalable Vector Graphics (SVG) 17, 20, 24, 109–13, 125–6, 155–8, 160–1, 187, 195, 211–29, 265, 285–303, 350, 357
- Scratchpad, DoJa 236–8, 251–2
- Screen command type 154–5, 364–9
- Screen object 37, 364–9
- screens
 - see also* displays
 - diversity handling issues 169–74
 - games 268–303
 - LCDUI APIs 35–42, 43–8, 49–56, 91–2, 119, 163, 364–9
 - screensavers 25–6, 237–8, 266
 - typical specifications 35, 308–9, 320
- SDKs
 - see also* emulator, Wireless Toolkit
 - concepts 109, 115, 131, 147, 175–205
 - debugging, on-device (ODD) 178, 180–9, 192–5, 198–201, 348–9
 - Device Explorer tool 200–1
- generic 176, 177–8, 190–5
- J2SE 31
- Java ME Developer Library 169, 187–8
- Java ME SDK 3.0 176–8
- manufacturers 175–6, 189
- MIDlets 180–2, 190–5
- Motorola MOTODEV Studio 201–5
- NetBeans 178, 179–80, 184–6, 188–9, 192–5, 198–201, 217, 223–6
- recommendations 176–7
- S60 178–89, 348–9
- SNAP Mobile 188, 413–31
- Sony Ericsson SJP-3 195–201, 348–9
- SVG-T Converter 187
- tools for Java on Symbian OS 131, 175–205
- UIQ 3 189–205
- WidSets 388–400
- secure socket connections 65, 70, 72–3, 125–6
- SecureConnection 73
- security characteristic of Java
 - best practice 323–5
 - CAs 60–9
 - concepts 9–10, 13, 30–4, 57, 58–81, 94–5, 99–101, 130–5, 232–3, 239–40, 255, 263–4, 323–5, 336–7, 350–1, 373–6
 - illustrative signed MIDlet suite example 65–8
- JSR-135 MMAPi 94–5

- security characteristic of
 - Java (*continued*)
 - JSR-177 SATSA 17, 20, 109–13, 211–29, 359, 373–6
 - networking 69–81, 130–1
 - protected APIs 59–77, 80–1, 95, 99, 239–40, 323–5, 350–1
 - Recommended Security Policy 68–9, 95, 99
 - trusted MIDlet suites 59–77, 80–1, 95, 99, 133–5, 323–5
 - untrusted MIDlets 59, 65, 68–77, 80–2, 95, 99, 323–5
 - Wireless Messaging API (JSR-120 WMA) 99–101
 - X.509 PKI 59–61
- SecurityException 323–5
- sending messages (JSR-120 WMA) 96
- SendInviteL() method 379–80
- serial ports 70, 260
- Series 40 3rd Edition 317
- server sockets connections 65, 70, 71–3, 125–6, 230
- servers 58, 65, 70–3, 125–6, 230, 322–3, 346–7, 374–6
- ServerSocket... 65, 72–3, 125–6
- ServiceRecord() method 299–302
- serviceRepaints() method 37, 280–303
- servicesDiscovered() method 299–302
- Session Initiation Protocol (SIP, JSR-180) 17, 20, 109–13, 211–29, 359, 377–84
- session user permission 65–9
- set() method 38, 97–100
- setActive() method 290–302
- setAddress() method 97–100
- setColor() method 43, 47–8, 52, 249, 275, 294–302
- setCommandListener() method 38, 45–8
- setConstraints() method 41
- setCurrent...() methods 37, 290–302, 307–8, 319–22
- setDefaultCommand() method 39
- setDisplayable() method 45–8
- setFrameSequence() method 54–6
- setPosition() method 53–6
- setRecordLocation() method 94–5
- setRecordStream() method 94–5
- setSize() method 41
- setText() method 41
- Settings 286–302
- setViewWindow() method 53–6
- Sharp, SH905iTV 233
- short-link connections 109–13
- showNotify() method 161–2
- Siemens 6
- signing process
 - see also security...
 - authentication methods 63
- illustrative example 65–8
- MIDlets 30–4, 57–77, 95, 99, 232–3, 323–5
- untrusted MIDlets 59, 65, 68–77, 80–2, 95, 99, 323–5
- SIM card 61, 66, 69, 203–4, 373–84
- simplicity and familiarity characteristic of Java 9–10
- The Sims** 2 5
- simulation rate 269, 271–2, 284
- Sip... 109–13, 378–9
- SIP see Session Initiation Protocol
- SIS files 140, 184–5, 197–8
- size constraints 123–5, 214–15, 236–8, 250, 255, 282–3, 308–9, 323, 326–7
- sizeChanged 40
- SizeExceededException 109–13
- smart cards 11
- smartphones
 - see also mobile devices
 - concepts 3–12, 17–18, 145–6, 149–74, 308–9
 - information sources 145–6, 239–40, 273, 297–8
 - usage survey 18
- SMS 6–7, 18, 29, 65, 79–80, 82, 95–101, 154–5, 203–5, 238–40, 268–9
- see also Wireless Messaging API (JSR-120 WMA)
- receiving messages 97–8

- sending messages 96
- SNAP Mobile
 - APIs 416, 417–18, 422–31
 - benefits 411–12
 - business model 411–12
 - Certification Emulator 413
 - Client API 416–31
 - community features 414–16
 - compliance tests 121, 413, 420
 - concepts 22, 120–1, 188, 303, 411–31
 - deployment process 413–14
 - Device and Network Test application 412, 418–19
 - Emulation Environment 413, 417–18, 419–22, 429
 - game development and publishing process 120–1, 303, 412–14
 - getting started 417–31
 - in-game features 414–15
 - Instant Messaging 121, 416, 429–31
 - Live Development Server 417
 - Maze Racer** 419–22, 431
 - out-of-game community features 414, 415–16
 - planning process 412–13
 - Presence feature 416, 426–30
 - SDKs 188, 413–31
 - technical architecture 416–18
 - SnapshotControl 218–20
- social change and mobile devices 6–7, 26
- socket 71–3
- SocketConnection 71–3, 125–6
- sockets 58, 65, 70, 71–3, 125–6, 230, 253, 373–6
- Softbank 231–2
- SoftKey... 246–50
- softkeys 154–8, 237–8, 246–50, 252, 276–303
- software as a service (SaaS) 323
- Solaris operating system 9
- Sony 262–3
- Sony Ericsson 6, 270
 - Capuchin framework 24
 - Developer World 200–1
 - G900 101
 - P900 101
 - SJP-3 SDK 115, 195–201, 348–9
 - W950 195–6
 - W960i 106–13, 115, 133–46, 151–8, 164–5, 189–205
- source code creation
 - methods, widgets 395–409
- Space Invaders** 266
- Spacer 40
- speech recognition, DoJa 255, 263
- splash screens 217–18, 271–2, 307–8, 311–12
- SplashScreen... 217–18, 307–8, 311–12
- Spotless project 12–13
- sprites 21, 42–3, 49, 50, 51–54, 55, 238, 270, 273–8, 281–2, 293
- see also* animations; Canvas
- secure sockets layer (SSL) 65, 72–3
- Standard Widget Toolkit (SWT) 115–22
- start 55–6, 86–95, 129–30, 275–303, 321–2, 341, 380–1
- start-on-boot services 25
- start-up time, games 271
- startApp 29–30, 32–4, 45–8, 79–81, 99, 117–22, 126–31, 161–3, 251, 279, 307–8, 311–12
- startRecording 317–22
- states, MIDlets 28–34, 45–8, 79–81, 126–31, 251, 270, 277–303, 306–8, 311–12
- static URLs 406–7
- stop 55–6, 86–95, 321–2
- StopTimeControl 87–95
- StorageTek 9
- String 309–11, 315
- StringBuffer 76–7, 309–10
- String.equals 310–11
- String.intern 310–11
- StringItem 41, 43–8, 76–7
- strings, best practice 309–11
- stylus events 163–6, 270–303
- subsystem architecture, Java ME 331–51, 353–84
- Sun Microsystems
 - see also* Java...
 - concepts 9–13, 17, 24, 30–1, 145–6, 297, 338–40

- supports.mixing
 - 152–3, 168
- supports.recording
 - 318–22
- supports.video.capture
 - 318–22
- SVG *see* Scalable Vector Graphics
- SVG-Tiny (SVG-T) 187, 217–18, 288–90
- SVGAnimator 225–6
- SVGImage 109–13
- SVGMenu 217–18
- SVGPlayer 217–18, 225–6
- Swing toolkit 35
- swipe interaction 166
- SWT *see* Standard Widget Toolkit
- Symbian C++ 3, 8, 17–18, 141–4, 147, 176, 190, 333, 343, 344–6, 356–9, 361–2, 373–6
- Symbian Foundation 7–8
- Symbian Ltd 359
- Symbian OS
 - see also* Symbian OS versions
 - abstraction layers 331–3
 - APIs 8, 17–18, 82, 83, 94, 100, 107–22, 141–4, 147, 176, 190, 331–51, 353–84
 - architectural viewpoint 333–51
 - asynchronous operations 346–8, 360–2, 375–6
 - background/foreground transitions 161–2
 - benefits 19–21, 105–6, 125–31, 308–9, 333, 384
 - best practice 325–7
 - certification exam 27, 100
 - CLDC 1.1 81–2
 - computing capabilities of Java ME 123–5
 - concepts 3, 7–8, 17–21, 24–6, 27, 69–77, 82, 100, 105–47, 227–9, 308–9, 325–7, 331–51, 353–84
 - connection constraints 125
 - constraints 22, 25–6, 105–6, 123–5, 214–15, 233–4, 282–3, 308–9, 323, 326–7
 - error handling 139, 308, 313–14
 - events 346–8, 360–2, 375–6
 - FOMA phones 232–3
 - games 282–303
 - historical background 7–8
 - information sources 145–6
 - installation/removal of MIDlet suites 136–7, 140, 190–5, 336–7
 - integration of APIs with Java ME 351, 353–84
 - inter-process communication 140–5, 335–51
 - JAD attributes 122–3
 - Java ME Detectors Suite example 107–13, 151, 190–5
 - Java ME management on devices 131–8
 - Java ME subsystem architecture 331–51, 353–84
 - JSR-75 Files & PIM 359–62, 383–4
 - JSR-118 MIDP 2.0 359, 369–73
 - JSR-120 WMA 100, 107–13
 - JSR-135 MMAPAPI 83, 94, 108–13, 166–8, 287–303, 353, 359, 369–73, 384
 - JSR-172 Web Services 359, 362–4
 - JSR-177 SATSA 359, 373–6
 - JSR-180 SIP API 359, 377–84
 - JSR-185 JTWI support 82, 100
 - JSR-248 MSA 229–30, 359
 - JSRs supported 146–7, 151, 152–3, 159–61, 190–5, 345–6, 351
 - JTWI (JSR-185) 82, 100
 - launching Java ME applications 111–13
 - layered model 8
 - LCDUI 364–9, 384
 - MIDlets 106–13, 128–31, 180–2, 190–5
 - MIDP 130–1, 232–3, 343, 344–6, 359, 369–73, 377–8
 - Mobile Media API (JSR-135) 83, 94, 108–13, 166–8, 287–303, 353, 359, 369–73, 384
 - modular design 8
 - MSA 229–30, 359
 - multimedia applications 83, 94, 108–13, 166–8, 287–303, 353–84
 - multitasking benefits 126–31, 279, 325–6

- native capabilities
 - 139–41, 353–84
- native UI controls 333
- networking 69–77, 356–7
- Nokia 3
- non-JSR supported APIs 113–22
- open platform 7–8, 18, 109–13, 325–6, 332–3
- popularity 3, 7–8
- power benefits 125–31
- running a MIDlet
 - 106–13, 128–31, 180–2, 190–5
- SDKs 175–205
- security settings 130–1, 132–5, 232–3, 350–1, 359, 373–6
- short-link connections 109–13
- statistics 7–8, 233
- Task Manager 137–8, 190–2
- telephony information
 - retrievals 121–2
- tools for Java on Symbian OS 131, 175–205
- variety of hardware 8
- Wireless Messaging API (JSR-120 WMA) 100, 107–13, 127–31
- Symbian OS v7 4, 20, 338, 349
- Symbian OS v8 19–20, 232
- Symbian OS v9 3, 20, 82, 125–6, 179, 189, 233, 288–9, 333, 367
- Symbian Signed 130–1, 232–3
- synchronization
 - characteristic of Java 9–10
- system threads 306–7, 340–1
- SystemAMS 336–51
 - see also* Application Management Software
- System.gc 322
- T-Mobile 5
- Task Manager 137–8, 190–2
- TCK 178
- TCP 15, 126, 141–5, 197–201, 230, 357, 373–6, 377–8, 416–17, 419
- technical architecture, SNAP Mobile 416–18
- telephony information
 - retrievals 121–2
- TempoControl() method 87–95
- TEntry 361–2
- text messages 5–6, 16, 18, 20, 79–80, 81–2, 95–101, 107–13, 211–29, 233–4, 238–9
 - see also* SMS
- TextBox object 37, 38, 43–8, 82
- TextField object 41, 46–8, 82, 368–9
- TextMessage 96–100
- Thawte 69
- third-party developers 3, 8, 225–6, 358
- threads 13, 16–17, 124–5, 271–303, 306–7, 336–7, 340–1, 346–8
- Ticker object 41, 43–8
- TiledLayer object 49, 50–1, 52–6, 282
- TNotifyType 361–2
- Tom-Tom personal navigation devices 6
- Tone Sequence file format 82, 274–303
- ToneControl 87–95, 274–7
- tools for Java on Symbian OS 131, 175–205
 - see also* SDKs
- ToruCa, DoJa 262
- touch screens 35, 106–13, 162–6, 306
- TransactionModule 109–13
- TRequestStatus 361–2
- trusted MIDlet suites
 - 59–77, 80–1, 95, 99, 133–5, 323–5
- Trusted Third-Party domain 69
- try keyword 55–7, 73–4, 89–93, 129–31, 159–60, 250, 313–14, 319–20
- TunerControl 220–1
- turn-based games 266
- UART 260
- UDP 15, 73–4, 253, 377–8
- UI *see* user interface
- UIExampleMIDlet
 - example 43–8
- UIQ 7–8, 39, 106–13, 115, 133–5, 145–6, 149, 151–3, 162–74, 189–205, 220, 233, 270, 336
- UIQ 3 SDKs 189–205, 220
- UltraSPARC processor 9
- Unicode 17, 43, 82, 310
- unlock() method 248–50, 252
- unregisterConnection() method 79–81

- untrusted MIDlets 59, 65, 68–77, 80–2, 95, 99, 323–5
- URLs 83–100, 118–22, 219–20, 238, 243, 256–7, 301–2, 372
- JAR files 33–4
- SNAP Mobile 424–31
- syntax 71–4, 97
- widgets 385–407
- user experiences
 - best practice 305–8, 327
 - input mechanisms 306
 - resilience expectations 308
 - responsiveness factors 306–8
- user interface (UI)
 - see also* GUI; LCDUI; MOAP...; S60; UI; UIQ
 - concepts 8, 24–5, 34–56, 106–46, 149–50, 154–8, 162–74, 233–4, 335–6, 350, 373, 387, 402–3, 405–6
 - event model 36–7, 42–3
 - information sources 145–6
- user permissions, protected domains 59, 64–9, 75
- user-defined class loaders 13
- UTF-8 310
- Utilities 248
- Utilities window, emulator 182–4
- vector graphics 17, 20, 24, 109–13, 125–6, 155–8, 160–1, 187, 195, 211–29, 265, 285–303, 350, 357
- vendors, MIDlets 33–4
- Verisign 69
- version 152–3
- versions, MIDlets 33–4
- vibration events 284–303
- video 5, 6, 18, 82–95, 166–8, 218–20, 258–9, 287–302, 316–22, 369–73
 - see also* Mobile Media API; multimedia... playing 90–2, 258–9, 316–22
- VideoCanvas 91–2
- VideoControl 87–95, 218–20, 316–22, 371–3
- video.encodings 318–22
- VideoPlayer 90–3
- viewport 270, 288–91
- virtual machines (VMs) 10–14, 19, 310–11, 331–51, 361–2, 375–6
- VisualPresenter 258
- VMs *see* virtual machines
- Vodafone 231–2
- voice calls, usage survey 18
- VoIP 308, 383
- volatile 278
- VolumeControl 87–95
- W-CDMA protocol 232
- wait() method 347–8
- waitForNotify() method 129–31
- wallpaper 5, 266
- WAV files 58, 253–4, 284–303
- Web 2.0 services 385–409
 - see also* Internet
- Web Runtime 18
- Web Services (JSR-172) 17, 20, 108–13, 211–29, 342–3, 359, 362–4
- WeightEntry 244–8
- while 267–8
- widgets
 - see also* WidSets
 - Apple Dashboard 385
 - BBC News 388
 - concepts 25, 115–22, 216–18, 238, 366–9
 - creation methods 391–3
 - Digg 391
 - dynamic development 409
 - Flickr viewer 218–19, 223–4, 399–409
 - published widgets 392–3, 408
 - Rawsocket.org 392
 - S60 WRT 386
- WidSet Scripting Language (WSL) 386–409
- WidSets
 - architecture and features 387–93
 - ATOM feeds 389, 391–2, 401
 - benefits 385–6
 - compilation method 398–400
 - concepts 18, 385–409
 - descriptor creation methods 394–5, 399–402
 - emulator 399–400
 - Flickr viewer widget 218–19, 223–4, 399–409
 - getting started 390–1
 - icons 392–3
 - publishing widgets 392–3, 408
 - relevance to Java 386–7
 - rich widgets 393–400
 - RSS feeds 389, 391–2, 401

- running widgets
 - 398–400
- SDK 388–400
- source code creation
 - methods
 - 395–409
- usage methods
 - 389–93
- widget creation methods
 - 391–3
- WidSets Mobile Dashboard
 - 387–8, 390–400
- WidSets Server 388–9,
 - 399–400
- WidSets.com 387–8
- WIM/SIM cards 61, 66, 69,
 - 203–4
- Windows *see* Microsoft. . .
- Wireless Messaging API
 - (JSR-120 WMA)
 - concepts 16, 20, 81–2,
 - 95–102, 203–5
- JSR-205 17, 20, 100,
 - 109–13, 195,
 - 211–29
- Push Registry API 96,
 - 98–100
- receiving messages
 - 97–8
- security model
 - 99–101
- sending messages 96
- Symbian OS 100,
 - 107–13, 127–31
- Wireless Toolkit (WTK)
 - 30–1, 34, 44–8,
 - 56, 65–8, 91,
 - 175–7, 189,
 - 195–201, 234,
 - 279, 420–1, 429
 - see also* emulator, SDKs
- Wireless Toolkit (WTK)
 - 2.5.2 177–8
- WLAN 186
- WMA *see* Wireless
 - Messaging API
 - (JSR-120 WMA)
- ‘Write Once Debug
 - Everywhere’ 16
- ‘Write Once Run Anywhere’
 - 9, 16
- writing MIDP games
 - 265–303
- WSL *see* WidSet Scripting
 - Language
- WTK *see* Wireless Toolkit
- WURFL-device search
 - database 178
- X.509 PKI 59–61
- XML 212, 215, 288–9,
 - 362–3, 394–409
- Yahoo! widgets 386–5

Indexed by Terry Halliday