

Е. Буткевич

Пишем
ПРОГРАММЫ
и ИГРЫ
для СОТОВЫХ
ТЕЛЕФОНОВ

Е. Буткевич

Пишем ПРОГРАММЫ И ИГРЫ для Сотовых ТЕЛЕФОНОВ

 **ПИТЕР®**

Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара • Новосибирск
Киев • Харьков • Минск

2006

ББК 32.882.9
УДК 621.395.004.3
Б93

Буткевич Е. Л.
Б93 Пишем программы и игры для сотовых телефонов. — СПб.: Питер, 2006. —
204 с.: ил.
ISBN 5-469-01139-9

Эта книга — общедоступный самоучитель программирования для мобильных телефонов, с помощью которого любой, даже не искушенный в секретах программирования читатель, сможет овладеть базовыми знаниями языка Java, навыками работы с необходимыми инструментами и утилитами, написать свои первые программы, а главное — увидеть и показать друзьям результат на экране своего мобильного. Книга написана доступным, живым языком и ориентирована вовсе не на профессионалов, а на всех, кто хотел бы расширить возможности своего мобильного и испытывать себя в роли программиста.

Попробуйте — это совсем не сложно!



ББК 32.882.9
УДК 621.395.004.3

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Краткое содержание

Предисловие	10
Глава 1. С чего начинать?	14
Глава 2. Hello, World!	23
Глава 3. Пользовательский интерфейс высокого уровня	32
Глава 4. Работа с графикой	42
Глава 5. Структуры и форматы данных в J2ME	52
Глава 6. Организация многопоточных приложений	66
Глава 7. Сохранение данных и параметров приложения	74
Глава 8. Стандартные средства пользовательского интерфейса ..	85
Глава 9. Организация потоков, работа с файлами	96
Глава 10. Ввод текста, работа со строками	107
Глава 11. Работа со временем, датой и календарем	119
Глава 12. Работа над ошибками	127
Глава 13. Расширение функциональности с помощью API	137
Глава 14. Приложения для работы с Интернетом	147
Глава 15. Защита приложений	161
Вопрос. Распространение мобильных приложений.	
Практические советы по продажам мобильного контента	171
Приложение 1. Полный листинг программы «Змейка»	173
Приложение 2. Полный листинг программы «BookReader»	190
Приложение 3. Полный листинг программы «AddressBook»	195
Литература	203

Содержание

Предисловие	10
К читателям	10
От издательства	12
История языка Java	12
Глава 1. С чего начинать?	14
Оборудование	14
Программное обеспечение	18
Эмуляторы	20
Глава 2. Hello, World!	23
Класс MIDlet	23
Создание мидлета	25
Импорт классов и пакетов	28
Компиляция и запуск мидлета	29
Упаковка мидлета	31
Глава 3. Пользовательский интерфейс высокого уровня	32
Класс Form	33
Класс Display	34
Отображаем картинку	34
Класс Command	36
Интерфейс CommandListener	38
Глава 4. Работа с графикой	42
Класс Canvas	42
Класс Graphics	42
Класс Graph	45
Рисование текста	47
Обработка клавишных событий	48
Глава 5. Структуры и форматы данных в J2ME	52
Объекты	52
Примитивные типы	53
Массивы	53

Класс Vector	54
Класс Snake	55
Класс Stack	64
Глава 6. Организация многопоточных приложений	66
Многозадачность	66
Класс Thread	67
Приоритеты тредов	68
Синхронизация тредов	70
Класс Snake	72
Глава 7. Сохранение данных и параметров приложения	74
RecordStore — хранилище записей	75
RecordEnumeration — нумератор списка записей	76
RecordFilter и RecordComparator — фильтр и компаратор	78
RecordListener — лови момент	80
RecordStoreException — возможные проблемы	80
High Score — таблица рекордов	81
Глава 8. Стандартные средства пользовательского интерфейса	85
Класс List	85
Interface Choice	86
Класс SnakeGame	87
Класс Gauge	93
Глава 9. Организация потоков, работа с файлами	96
Класс InputStream	96
Класс DataInputStream	99
Класс BufferedReader	101
Класс ByteArrayInputStream	106
Класс OutputStream	106
Глава 10. Ввод текста, работа со строками	107
Класс TextBox	107
Класс TextField	109
Класс AddressBook	111
Глава 11. Работа со временем, датой и календарем	119
Класс Date	119
Класс DateField	120
Класс Calendar	122
Класс TimeZone	123
Класс BirthdayFilter	124

Глава 12. Работа над ошибками	127
Класс Exception	127
Формирование исключений	130
Класс Alert	132
Класс AlertType	133
Глава 13. Расширение функциональности с помощью API ...	137
Nokia API	137
Samsung API	141
Siemens API	144
Глава 14. Приложения для работы с Интернетом	147
Класс Connector	147
Интерфейс Connection	148
Интерфейсы InputConnection и OutputConnection	149
Интерфейс StreamConnection	149
Интерфейс StreamConnectionNotifier	153
Интерфейс DatagramConnection	153
Интерфейс ContentConnection	154
Интерфейс HttpConnection	155
Глава 15. Защита приложений	161
Java Decompiler	161
Защита приложений	167
Обфускация	167
Bonus. Распространение мобильных приложений. Практические советы по продажам мобильного контента	171
Приложение 1.	
Полный листинг программы «Змейка»	173
Приложение 2.	
Полный листинг программы «BookReader»	190
Приложение 3.	
Полный листинг программы «AddressBook»	195
Литература	203

Посвящаю эту книгу двум самым близким
мне людям — отцу Леониду Михайловичу
и супруге Ксюше.

Предисловие

К читателям

Этот самоучитель поможет вам сделать первые шаги в программировании для мобильного телефона. Книга предназначена для всех желающих научиться разрабатывать приложения для мобильных, причем адресована в первую очередь тем, кто уже владеет базовыми навыками программирования и основами объектно-ориентированных языков. Тем не менее, она написана так, что даже самый неопытный программист будет в состоянии реализовывать приведенные примеры и обучаться всем хитростям сразу на практике. В таком случае, конечно же, не помешает дополнительная литература, поскольку многие базовые вещи программирования остались за кадром.

Для разработки приложений мы будем использовать платформу J2ME компании Sun Microsystems, поскольку на сегодняшний день это единственный универсальный инструмент программирования для мобильных телефонов.

Книга построена в виде самоучителя, разделенного на главы-уроки. В нем вы найдете больше практических советов и примеров, нежели теоретических сведений из области объектно-ориентированного программирования. Я постарался изложить материал максимально доступно, на пальцах, зачастую не вдаваясь в подробности и тонкости. Многочисленные примеры, приведенные в книге, хоть и не являются образцом аккуратности кодирования или истиной в последней инстанции, но реализованы и протестированы с точки зрения доступности непрофессионалам. Опытные программисты могут найти книгу скучной, с избытком информации об азах программирования, местами несерьезной в отношении к сложным задачам, а кто-то будет с удовольствием читать этот букварь. Основной упор книги делается на практическом применении материала, на том, какие средства могут потребоваться во время разработок и как ими пользоваться.

Красивая, элегантная, аккуратно реализованная идея имеет хорошие шансы принести создателю некоторую прибыль. Поэтому после обучения собственно программированию мы поговорим о процедуре получения денег за свой программный продукт. Пусть перспектива возможного извлечения прибыли послужит для вас хорошим стимулом к скорейшему обучению!

Книга содержит 15 глав, каждая из которых обзорекает отдельную тему. Первые две главы практически не касаются собственно программирования, но содержат подробную вводную информацию о том, как начать разработки, какое оборудование и программное обеспечение вам понадобится для этого. Это базовая информация, основа, без которой дальнейшее программирование невозможно.

Главы, касающиеся программирования, содержат теоретическую и практическую части. В теоретической части обсуждаются классы и методы, реализующие воз-

возможности платформы, а в практической подробно рассмотрена реализация примеров, использующих описанные выше классы. Большинство примеров переходят из главы в главу, расширяя свою функциональность, поэтому рекомендуется не прогуливать уроки, а аккуратно выполнять все самостоятельные задания. В тексте глав примеры приведены не полностью, а только та их часть, которая реализует новую функциональность. Если вы все же запутаетесь, то полный листинг некоторых примеров с подробными комментариями вы найдете в составе приложений в конце книги.

Далее приведен список всех глав с их кратким описанием.

Глава 1. С чего начинать? Обзор необходимых средств для мобильного программирования: модели телефонов, поддерживающие J2ME; коммуникационные кабели; необходимое программное обеспечение; студии разработки; эмуляторы; загрузка приложений в телефон.

Глава 2. Первое приложение «Hello, World!» Описание создания первой программы в студии разработки J2ME Wireless Toolkit. Структура программы, компиляция, тестирование программы на эмуляторе.

Глава 3. Пользовательский интерфейс высокого уровня. Иерархия класса Screen, отображение формы и взаимодействие с ней. Классы Display, Screen, Form, Command. Пример программы фотоальбома («Slide Show»).

Глава 4. Работа с графикой. Классы Canvas, Graphics. Общая организация графики, рисование простых геометрических фигур. Разбор примера программы управления точкой, рисующей фигуры на экране («Float Point»).

Глава 5. Структуры и форматы данных в J2ME. Типы данных. Классы Vector, Stack. Работа с данными. Расширение программы «Float Point» до игры «Змейка».

Глава 6. Организация многопоточных приложений, параллельное программирование. Класс Thread. Вопросы синхронизации. Расширение программы «Змейка»: пример приобретает вид законченной, качественной игры.

Глава 7. Сохранение данных приложения, сохранение параметров программ (RMI, класс RecordStore). Расширение «Змейки», добавление запоминания настроек и таблицы рекордов.

Глава 8. Стандартные средства пользовательского интерфейса, организация меню. Классы List, ChoiceGroup. Управляющие сигналы высокого уровня. Реализация меню для «Змейки».

Глава 9. Организация потоков, работа с файлами. Класс InputStream. Вопросы локализации, проблема корректного отображения русского текста. Обзор программы «Book Reader». Класс Font.

Глава 10. Ввод текста, работа со строками. Классы TextBox, TextField. Средства ввода и редактирования текста. Разбор программы «AddressBook», реализующей записную книжку.

Глава 11. Работа со временем, датой и календарем. Классы Date, Calendar. Извлечение внутренней даты и времени телефона. Расширение записной книжки до «Birthday Reminder».

Глава 12. Работа над ошибками. Классы Alert, Error, Exception. Обработка исключительных ситуаций, процесс отладки программ.

Глава 13. Расширение функциональности MIDP 1.0 с помощью API фирмы-производителя мобильного телефона. Nokia API, Samsung API, Siemens API. Работа со звуком, дополнительные возможности.

Глава 14. Методы передачи и приема данных. Классы Connector, Connection, DatagramConnection, StreamConnection, ContentConnection, HttpConnection. Реализация и разбор примеров интернет-приложений.

Глава 15. Защита приложений (обфускация). Обзор декомпиляторов: просмотр, редактирование и усовершенствование чужих приложений.

Bonus. Распространение мобильных приложений. Практические советы по продажам мобильного контента.

Приложение 1. Полный листинг программы «Змейка».

Приложение 2. Полный листинг программы «Book Reader».

Приложение 3. Полный листинг программы «Birthday Reminder».

Весь этот путь от начала до конца я пройду вместе с вами и реализую все описанные примеры. В процессе реализации и тестирования примеров я буду приводить скриншоты, чтобы никто не усомнился в успешном тестировании. То есть все реализовано, все проверено, все работает, чего и вам желаю. А теперь закончим вступительную часть словами классика. За мной, читатель!

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу <http://www.piter.com/download>.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства: <http://www.piter.com>.

История языка Java

По сравнению со своими старшими братьями в большой семье языков программирования, *Java* — совсем еще ребенок, причем едва достигший десятилетнего возраста. Но посмотрите, какой это ребенок! Давайте вместе оглянемся на первые дни этого малыша.

Родители вундеркинда работали в компании Sun Microsystems, которая в начале 90-х годов стояла перед задачей создания передового программного обеспечения для различных бытовых приборов, грубо говоря — для чайников. Основная проблема состояла в том, что для каждой новой модели бытовой техники существовали свои технические стандарты.

Ведущему инженеру проекта Патрику Нотону приходилось поддерживать в разработке сотни интерфейсов, пока это ему окончательно не надоело. В один пре-

красный день он решил уволиться ко всем чертям и понес заявление директору. Нужно отдать должное дальновидному руководству в лице Скотта МакНили, который попросил изложить причины ухода в подробном письме. Тут уж Патрик Нотон оторвался от души и написал все, что он думает о компании и о ее технологиях, беспощадно раскритиковав недостатки Sun Microsystems.

Письмо возымело на удивление большой успех как у ведущих инженеров, так и у высшего руководства компании, а именно у Билла Джоя, основателя Sun Microsystems, и Джеймса Гослинга, непосредственного начальника Нотона. В день планируемого увольнения Патрик Нотон получает зеленый свет на реализацию всех своих идей и собственную группу ведущих разработчиков под кодовым названием Green, чтобы они делали что угодно, но создали нечто необыкновенное.

Команда приступила к разработке нового объектно-ориентированного языка программирования, который должен был стать ведущим на рынке бытовой электроники. Вскоре Нотон предложил использовать новые наработанные технологии для интернет-приложений, в связи с чем были написаны компилятор Java и браузер HotJava. 23 мая 1995 года компания Sun официально представила Java и HotJava на выставке SunWorld'95. Именно тогда наш «малыш» сделал первый шаг к вашему мобильнику.

Платформа **Java 2 Micro Edition (J2ME)** была разработана уже ближе к нашим дням для устройств с ограниченными ресурсами памяти и процессора, таких как сотовые телефоны, пейджеры, смарт-карты, органайзеры и миникомпьютеры. J2ME позволяет запускать Java-приложения на ресурсо-ограниченных вычислительных устройствах. Для данных целей J2ME адаптирует существующую Java-технологию.

Глава 1

С чего начинать?

Эта глава содержит обзор необходимых технических средств для программирования: моделей телефонов, поддерживающих J2ME, коммуникационных кабелей, программного обеспечения, студий разработки, эмуляторов.

Перед тем как перейти непосредственно к программированию, рассмотрим все, что может пригодиться нам в процессе разработки приложений для мобильных телефонов. Подавляющее большинство людей обычно пропускают введение в любой книжке, которая попадает к ним в руки. Если вы относитесь к этой категории, то ничего страшного не произошло: информация там была чисто познавательного характера. Но начиная с этой главы, приготовьтесь к серьезной работе, поскольку программирование — это такая наука, которую нельзя выучить в теории.

Нужно отметить, что сейчас начинается самый главный этап в нашем обучении. Не сделав всего, что написано в первой главе, дальше можете и не продолжать читать этот самоучитель, поскольку его чтение без реальной практики — пустая трата времени.

В любом самоучителе на любую тему есть упражнения, которые необходимо выполнять. Ваши упражнения — это реализация примеров, описанных в книге, а без первого этапа — поиска и установки всего необходимого оборудования и программного обеспечения — эти упражнения выполнять просто невозможно. Помните: чем строже учитель, тем большего результата он добьется. Запаситесь терпением и настойчивостью и не делайте себе поблажек.

Оборудование

Итак, начинаем! Первое и самое необходимое в нашем деле — это, конечно же, сотовый телефон с поддержкой языка Java. Тот, кто уже давно подключил свой телефон к компьютеру и успешно качает на него мелодии, картинки и Java-приложения, может смело пропустить вводную часть вплоть до программного обеспечения. Ну а те, у кого уже установлено и все описанное далее программное обеспечение, могут задуматься о том, не следует ли им отдать предпочтение более серьезной литературе.

Далеко не все телефоны поддерживают Java-приложения, и если вы не уверены относительно нужной вам модели, то стоит заглянуть в документацию телефона или на сайт разработчиков developers.sun.com/techtopics/mobility/device с полным списком моделей телефонов, поддерживающих Java-технологии.

Мобильные устройства не содержат собственных средств разработки — редакторов, компиляторов и прочих инструментов, да и не нужно это аппарату с двадцатью кнопками и однодюймовым экранчиком. Поэтому придется прибегнуть к межплатформенной разработке, то есть программировать на компьютере, а только потом переносить готовую программу на телефон. Таким образом, наличие компьютера — такое же необходимое условие в разработке мобильных приложений, как и наличие телефона. Надеюсь, что на компьютере у вас установлена операционная система Windows (которую ругают все кому не лень, но тем не менее продолжают ею пользоваться), поскольку все рассмотренное далее программное обеспечение предназначено именно для этой системы.

Связать компьютер с телефоном можно несколькими способами. Мы рассмотрим наиболее простой и наименее дорогой способ связи через дата-кабель. Бывает, что дата-кабель идет в комплектации с телефоном, считайте тогда, что вам повезло. Если нет, то придется купить его отдельно, выложив за него от 15 до 20 долларов. Покупайте кабель только для своей модели телефона (обычно совместимые модели указаны на упаковке). Даже если разъем кабеля подходит к вашему телефону, то это еще ни о чем не говорит. В комплектации кабеля должен быть диск, содержащий драйвера для этого кабеля.

Для начала следует установить драйвер для вашего кабеля. Большинство современных кабелей подключаются к USB-порту компьютера и при включении будут обнаружены системой как новое USB-устройство. Вслед за обнаружением нового устройства автоматически запустится мастер обнаружения нового оборудования, который помогает установить драйвер устройства (рис. 1.1).

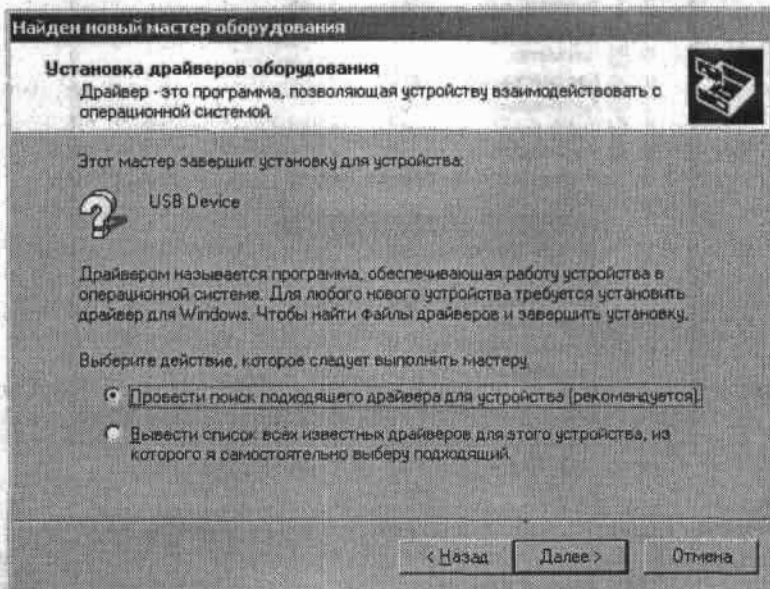


Рис. 1.1. Мастер оборудования поможет установить нужный драйвер

Вставьте диск с драйверами для кабеля, выберите в окне мастера переключатель Провести поиск подходящего драйвера для устройства, на следующем этапе мастера

выберите пункт размещение будет указано, а на следующем шаге укажите на диске драйвер для вашей операционной системы. Если поиск нужного драйвера на диске вызовет у вас затруднения, то можно вернуться к предыдущему этапу мастера и выбрать пункт дисководы компакт-дисков, тогда система выберет подходящий драйвер с диска автоматически.

Далее система установит драйвер и уведомит вас о том, что установка программного обеспечения для нового устройства завершена, указав его название, к примеру, Prolific USB-to-Serial Comm Port. Жмите кнопку Готово.

Теперь можно посмотреть результат установки устройства. Щелкните правой кнопкой мыши на значке Мой компьютер на рабочем столе и выберите в контекстном меню самый нижний пункт — Свойства. Перейдите на вкладку Оборудование, запустите диспетчер устройств, нажав соответствующую кнопку, и проверьте пункт Порты COM и LPT. Здесь должно появиться новое устройство, а в скобках — номер COM-порта, с которым оно связано (рис. 1.2). Запомните этот номер: он вам пригодится для настройки программы связи компьютера с телефоном.

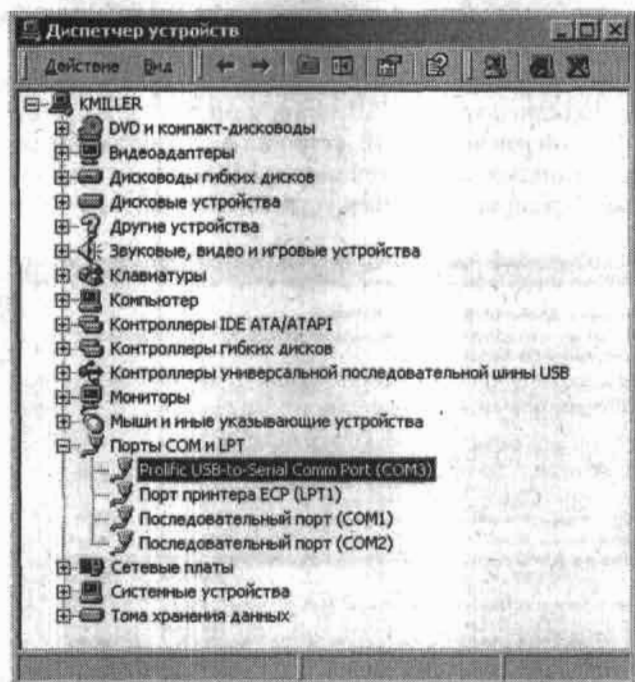


Рис. 1.2. В списке устройств появился дата-кабель

Если вы видите новое устройство, значит установка прошла успешно. На самом деле, дата-кабель может дать неограниченные возможности по управлению вашим телефоном, от загрузки новых игр, программ, мелодий и картинок до обеспечения компьютеру выхода в Интернет (технология GPRS) и прошивки исполняющей системы телефона. Кстати, с прошивками будьте аккуратны и всегда отдавайте себе отчет в том, что вы делаете — хотя бы для того, чтобы специалист, в случае чего, смог вам помочь.

Программы связи компьютера с телефоном не входят в комплект дата-кабеля, обычно они содержатся на диске, который идет в комплектации самого телефона. Если по каким-то причинам программа для загрузки приложений не обнаруживается на диске или отсутствует сам диск, то придется искать программу, совместимую с вашей моделью телефона, в дебрях Интернета. Вооружитесь терпением: если у вашей модели телефона есть возможность загрузки, значит должна быть и программа. Кто ищет, тот... сами знаете — что.

Для кого-то более легким покажется другой путь: зайти в киоск с программным обеспечением и купить себе один из многочисленных дисков типа «Все программы для телефона ...» (нужное вписать). К сожалению, в рамках данной главы мы не сможем рассмотреть программы для всех моделей телефонов, поэтому для примера ограничимся одной из подобных программ для аппаратов Nokia 7210, 3510i, 6100, 6610. Программа связи компьютера с телефоном, которую мы рассмотрим, называется Mobile Media Browser (MobiMB), ее можно скачать по адресу www.logomanager.co.uk, однако там она стоит денег, поэтому будем рассматривать ее только в качестве наглядного пособия для всех программ такого рода, поскольку все они обладают схожими настройками и управлением.

Для настройки программы связи в меню Connection settings (Параметры соединения) следует добавить новое соединение с помощью команды Add connection (Добавить соединение). Вам будет предложено на выбор несколько типов кабелей, из которых вы должны выбрать тип своего кабеля, указанный на упаковке (типы кабелей могут быть, например, такими: FBUS, DLR-3, DKU-5). В свойствах нового соединения следует также указать номер COM-порта, с которым операционная система связала наш кабель на этапе его установки.

Теперь подключаем телефон к компьютеру с помощью дата-кабеля, в программе MobiMB, может быть, потребуется нажать кнопку Refresh (Обновить), чтобы ускорить процесс определения телефона программой. После того как телефон будет обнаружен и определен, в левом окне программы появится изображение телефона со списком, содержащим имена нескольких папок (рис. 1.3). Если на этом этапе соединение не устанавливается, попробуйте подключить кабель к другому USB-порту, проверьте соответствие кабеля вашей модели телефона, убедитесь, что кабель установлен в операционной системе, а соединение настроено правильно.

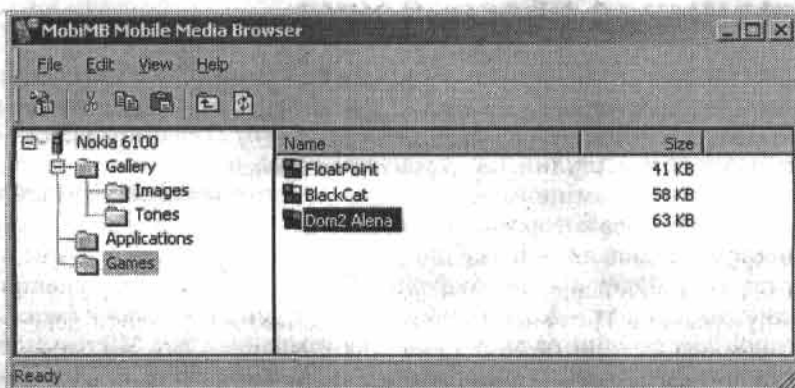


Рис. 1.3. Медиа-браузер дает доступ к памяти телефона

Мы видим, что программа успешно обнаружила аппарат и дала нам доступ к четырем папкам телефона: `Gallery/Images`, `Gallery/Tones`, `Applications`, `Games`, где хранятся картинки, мелодии, Java-приложения и Java-игры соответственно. В нижней строке выводится объем свободной памяти телефона. Нас, конечно же, интересуют две последние папки: именно там мы будем размещать созданные нами программы.

Немного о программах, которые мы будем разрабатывать. Приложение для мобильного телефона состоит, как правило, из двух файлов: архива с расширением `.jar`, содержащего байт-код исполняемой части приложения и его ресурсы, и файла-описателя с расширением `.jad` для первоначальной проверки совместимости с устройством и настройки различных атрибутов. После того как связь с телефоном установлена, с загрузкой Java-приложения в телефон справится даже ребенок: нужно выделить `.jar`- и `.jad`-файлы и перетащить их мышью в папку `Applications` или `Games`, после чего должен начаться процесс копирования.

Если копирование не началось, значит аппарат отвергает приложение, ориентируясь на параметры, указанные в `.jad`-файле описателя программы. Причин может быть несколько: либо размер приложения превосходит максимально допустимый (например, для аппаратов Nokia серии 40 максимально допустимый размер приложения составляет 64 Кбайт), либо недостаточно свободной памяти, либо данные в файле описателя некорректны.

Если приложение было успешно скопировано, то можно отсоединять телефон от провода, заходить в раздел, куда вы скопировали приложение, и запускать вашу программу, которая теперь не зависит от средств разработки, устройств связи и дополнительного оборудования. Теперь приложение всегда у вас в кармане и вы можете воспользоваться им в любой момент.

На этом обзор необходимого оборудования и его настройки закончен. Следующий этап — поиск и установка средств разработки мобильных Java-приложений. На этом этапе вам желательно иметь доступ в Интернет, поскольку именно там мы будем черпать необходимые программные ресурсы. Рассчитывайте на то, что в общей сложности вам придется скачать около 100 Мбайт. Если это неприемлемо, то вам прямая дорога в интернет-кафе или крупный магазин с программным обеспечением.

Программное обеспечение

Теперь рассмотрим необходимый минимум программного обеспечения, который должен быть установлен на вашем компьютере, чтобы перейти непосредственно к программированию. Сразу оговорюсь, что существует масса средств разработки, программ, студий, интегрируемых сред, и мы разберем лишь самый простой вариант программного обеспечения, своего рода необходимый «кандидатский минимум», без которого говорить дальше просто не о чем. Настоятельно рекомендую устанавливать именно то, что я скажу, а не то, что под руку попадется: так всем будет проще, поскольку на первом этапе объяснения я буду производить на пальцах — в стиле «нажми то, нажми это». Скажу также, что все программное обеспечение распространяется компанией Sun Microsystems абсолютно безвозмездно, то есть даром, поэтому, в худшем случае, вам придется платить лишь за трафик.

Технология языка Java связана с тем, что для запуска приложений требуется виртуальная машина Java. Наша операционная система, не буду лишним раз говорить какая, содержит свою версию виртуальной машины, которая, мягко говоря, нам не подходит, поэтому следует раздобыть и установить *исполняющую среду разработчика* Java™ 2 SDK, Standard Edition (J2SE SDK), version 1.4.2. Скачивается эта штукавина с сайта компании Sun Microsystems по адресу java.sun.com/j2se/1.4.2/download.html, «весит» 52 Мбайт и устанавливается без особых проблем: когда попросят, смело жмите Next и соглашайтесь на все предложенные условия. На всякий случай запомните, куда вы установили исполняющую среду, поскольку в дальнейшем какие-нибудь ленивые программы могут вас об этом спросить.

Второе, что нам понадобится, — это *набор средств для разработки Java-приложений* Sun Java Wireless Toolkit (J2ME WTK). На данный момент на том же сайте разработчиков по адресу java.sun.com/products/sjwtoolkit/ доступна версия WTK 2.3 Beta размером двадцать с небольшим мегабайт. Я же пользовался чуть ли не первой версией этого набора, которой оказалось вполне достаточно. Сам набор содержит все необходимые нам средства разработки: компилятор, упаковщик, несколько стандартных эмуляторов телефонов и еще некоторые полезные утилиты.

При установке WTK выполняет поиск виртуальной машины Java, уже нами установленной, и уточняет ее местоположение. Важно: устанавливать программу следует в папку, полный путь к которой не содержит пробелов и русских символов, поскольку в дальнейшем это может привести к проблемам. На все вопросы в процессе установки следует отвечать утвердительно.

Основная утилита набора, которой мы будем пользоваться на протяжении всей книги, называется KToolbar (рис. 1.4).

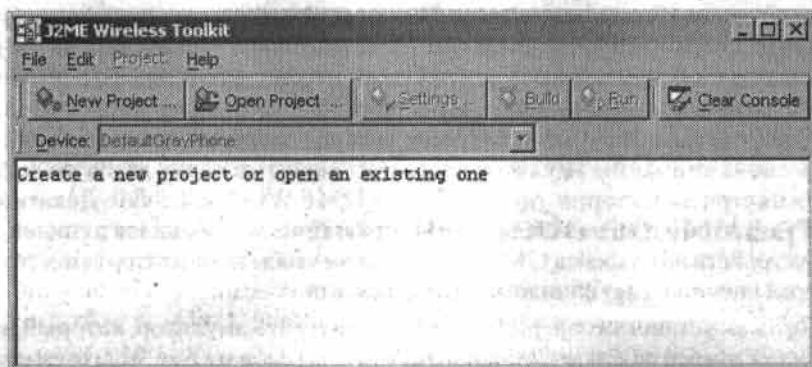


Рис. 1.4. Основное окно утилиты Ktoolbar

Это и есть, собственно, наше окно — если не в Европу, то в большой и увлекательный мир мобильного программирования: необходимый минимум функциональности, удобная навигация, в которой при всем желании сложно запутаться.

На этом «обязательные программы», без которых просто нельзя разработать мобильное приложение, заканчиваются, и начинаются «произвольные программы», которые не обязательны, но желательны.

Эмуляторы

При разработке приложений нам неоднократно потребуется их тестировать, чтобы вовремя выявить возможные ошибки и неточности. Можно, конечно, каждый раз копировать программу на телефон, но, боюсь, это сделает процесс разработки слишком трудоемким и утомительным. Поэтому в процессе отладки удобно воспользоваться эмулятором телефона — программой, которая имитирует реальное устройство на вашем компьютере. Эмуляторы предоставляют возможность запуска, отладки и управления мобильным приложением, что позволит избежать многократного копирования и поможет предотвратить ошибки, способные повредить содержимое памяти вашего мобильного устройства. Таким образом, весь цикл разработки приложения переносится на компьютер, а в телефон загружается только окончательная версия приложения.

Несколько эмуляторов содержатся в уже установленном нами наборе J2ME Wireless Toolkit. В дальнейшем мы будем их использовать, однако они имеют ряд серьезных недостатков. Во-первых, это эмуляторы «телефонов вообще», а не каких-то конкретных моделей, поэтому они не смогут отразить всей специфики конкретного аппарата, начиная с размеров экрана и расположения функциональных клавиш и заканчивая поддержкой специальных функций от разработчиков определенной марки. Во-вторых, эмуляторы встроены в J2ME WTK и не позволяют запускать приложения, созданные не в этой среде разработки, то есть просто скопированные на компьютер. При желании это ограничение можно обойти, но особого смысла я в этом не вижу, поскольку существуют более продвинутые эмуляторы, которыми мы и воспользуемся.

К более продвинутому относится эмулятор телефона Nokia 7210, которым можно воспользоваться при тестировании приложений для любых моделей самой распространенной серии 40 телефонов Nokia. Эмулятор входит в состав среды разработки приложений Nokia 7210 MIDP SDK 1.0, который можно скачать бесплатно на сайте разработчиков www.forum.nokia.com/tools (размер дистрибутива — 18 Мбайт). При установке пакета программа потребует серийный номер, который нужно отдельно получить на том же сайте. Соглашайтесь на все условия и вводите серийный номер. А теперь снова внимание: эмулятор нужно установить в папку `wtklib\devices`, находящуюся внутри директории, где установлен J2ME Wireless Toolkit. Делать это следует для того, чтобы наш новый эмулятор автоматически добавился в список имитируемых устройств интерфейса J2ME WTK, а также чтобы компилятор самостоятельно обнаружил специальные функции производителя.

После того как установка завершена, можно запускать эмулятор, который не только выглядит презентабельнее стандартных эмуляторов от Sun Microsystems и позволяет запускать Java-приложения, но и эмулирует все остальные функции телефона Nokia: записная книжка, будильник и даже настройки — все по-настоящему. Чтобы запустить приложение, нужно в меню File (Файл) выбрать команду Open (Открыть), после чего указать путь к приложению, которое следует запустить. Теперь, прежде чем загружать себе в телефон все подряд, можно убедиться, что игра или программа действительно того стоит. Этим эмулятором я и буду пользоваться на протяжении всей книги (рис. 1.5).

Кроме собственно запуска приложений, с его помощью можно следить за трафиком сетевых приложений и распределением выделяемой памяти в окне статуса

выполнения приложения, которое можно открыть через пункт меню Tools (Инструменты). Отметим также, что без исполняющей среды J2SE SDK установить эмулятор нельзя, поэтому не спешите, а делайте все по порядку, не забегая вперед.

Мобильный вариант языка Java разрабатывался как универсальный, то есть совместимый со всеми моделями телефонов, поддерживающими стандарт J2ME. Однако в каждом аппарате могут быть свои нюансы: различия в размерах экрана, ограничение на объем приложений и другие подводные камни. Так что если вы собирались продавать ваш программный продукт, будь то игра или просто полезное приложение, то обязательно нужно протестировать его на каждом аппарате из заявленного модельного ряда. Мало ли что — а подрыв репутации может быстро свести ваши заработки к нулю. Проверить нужно хотя бы на эмуляторах, однако найти, скачать и установить эмуляторы всех моделей с поддержкой Java — занятие, мягко говоря, кропотливое и требующее как минимум железной нервной системы.

Здесь на помощь может прийти сетевая система под названием Tira Developer Network от компании Tira Wireless, специализирующейся на разработке мобильных приложений. Данная система, доступная по адресу www.tiradeveloper.com, создана для тестирования приложений на совместимость со всеми распространенными моделями телефонов. Для тестирования предлагается около 150 мобильных устройств с поддержкой J2ME. В процессе проверки приложение может пройти около 60 специальных тестов, способных проверить совместимость приложения со всеми распространенными стандартами.

Для пользования системой вам достаточно зарегистрироваться на сайте, выбрать интересующее вас устройство, с помощью специальной формы заказать свое приложение и получить вердикт о соответствии вашего продукта выбранному устройству. Удобно, не правда ли, тем более что платить за это никому не нужно. Хотя немного смущает тот факт, что для проведения тестов приложения нужно закачивать к ним на сайт. Поэтому, если вы категорически не желаете, чтобы ваше приложение безвозмездно оказалось в чужих руках, то лишний раз подумайте, надеяться на их порядочность или нет.

Итак, чтобы начать разработку приложений для мобильных телефонов, нам требуется:

- телефон с поддержкой Java-технологии,
- компьютер с операционной системой Windows,
- кабель для связи компьютера с телефоном.

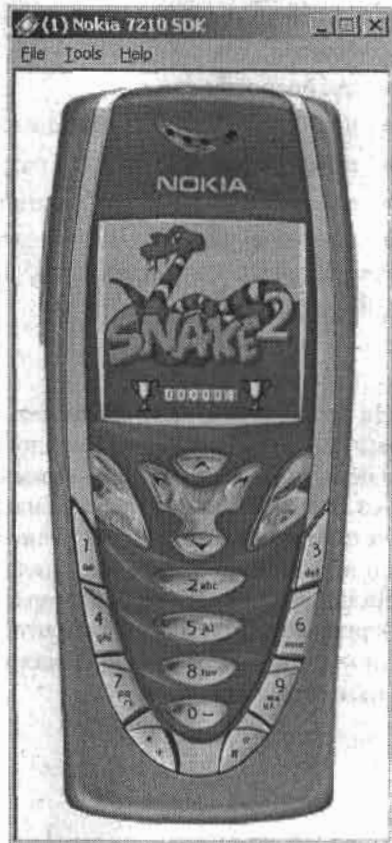


Рис. 1.5. Так выглядит эмулятор телефона с запущенной на нем игрой

При этом на компьютере должно быть установлено следующее программное обеспечение:

- драйвер кабеля,
- программа для загрузки приложений в телефон,
- исполняющая среда Java-разработки J2SE SDK (version 1.4.2),
- набор средств для разработки J2ME WTK,
- эмулятор вашей модели телефона — желателен, но не обязателен.

Объем всего перечисленного программного обеспечения составляет не более 100 Мбайт.

* * *

На этом самый важный подготовительный этап закончен. В следующей главе мы перейдем непосредственно к программированию. Базовый набор программ и оборудования, рассмотренный в этой главе, является той самой точкой опоры, без которой никакие рычаги нам не помогут. Я понимаю, что вам уже не терпится перейти к самому интересному, как в детективе — заглянуть в самый конец, но я не рекомендую этого делать, не усвоив всего, что написано в этой главе. Дальше вас ждет масса интересных примеров приложений. Реализуйте их, совершенствуйте, оптимизируйте — и помните, что не ошибается только тот, кто ничего не делает. Добро пожаловать в современный мир программирования для мобильных устройств!

Глава 2

Hello, World!

В этой главе мы рассмотрим структуру программы для мобильного телефона, а также все этапы ее создания и загрузки в телефон. Будем считать, что все необходимые провода уже приобретены и подключены к телефону, который поддерживает платформу J2ME, а соответствующее программное обеспечение успешно установлено на вашем компьютере.

Для создания полноценного мобильного приложения нам придется выполнить следующие этапы разработки:

- написание кода (текста) программы на языке J2ME;
- компилирование программы;
- тестирование приложения на эмуляторах;
- упаковка приложения в jar-архив;
- загрузка приложения в мобильный телефон.

Не будем спешить и для начала выясним, что же представляет собой приложение для мобильного телефона и что у него внутри.

Класс MIDlet

Поскольку Java 2 Micro Edition поддерживает большое количество разных устройств, то существуют различные конфигурации этой платформы. Не вдаваясь в подробности, мы будем использовать конфигурацию CDLC (Connected Limited Device Configuration), расширенную профайлом MIDP (Mobile Information Device Profile), который обеспечивает необходимую функциональность для программирования под мобильные телефоны, пользовательский интерфейс, постоянное хранение данных, таймеры, средства сетевой связи. *Мидлет* (MIDlet) — это и есть приложение Java, которое использует профайл MIDP и конфигурацию CLDC.

Название «мидлет» происходит от имени основного класса мобильного приложения `javax.microedition.midlet.MIDlet`. Основной класс приложения должен быть унаследован от абстрактного класса `MIDlet` и должен реализовывать методы `startApp()`, `pauseApp()` и `destroyApp(boolean)`. К сожалению, основы объектно-ориентированных языков остаются за рамками этой книги, поэтому те, кто считает, что ООП — это Организация Освобождения Палестины, могут на этом этапе почитать дополнительную литературу. Особо нетерпеливые могут сразу перейти к водным процедурам и подхватывать идеологию и терминологию языка на ходу, аккуратно разбирая приведенные в дальнейшем примеры программ.

Реализация MIDP дает программисту много возможностей и большую свободу действий, однако и эти возможности не безграничны, поэтому начнем с ложки дегтя — той головы, выше которой не прыгнуть в классической реализации языка. MIDP не поддерживает системного программирования, не дает доступа к системным функциям устройства и самой исполняющей системе. В компетенцию MIDP не входит управление пользовательскими приложениями и обеспечение средств для загрузки, хранения или запуска приложений.

При запуске загруженного в телефон приложения события развиваются следующим образом. Управляющая система создает экземпляр основного класса, наследованного от класса `MIDlet`, используя конструктор без аргументов, доставшийся в наследство от базового класса. Приложение переходит в состояние `PAUSED`, после чего система вызывает метод `startApp()`, который и будет являться для нас точкой входа в программу. «А как же `main()`?», — спросят опытные Java-программисты. Метода `main()` мы не видим, но он есть! Это часть реализации MIDP-профайла, и программное обеспечение телефона вызывает его без нашего ведома.

После вызова метода `startApp()` приложение переходит в состояние `ACTIVE`, захватывает необходимые ресурсы и начинает заниматься своими делами. Приложение выполняется до тех пор, пока система не переведет его в одно из состояний — `PAUSED` или `DESTROYED`. Три упомянутых состояния и составляют жизненный цикл приложения. Остановимся на каждом из них.

- **ACTIVE** — активное состояние. Приложение готово к запуску или уже выполняется. Система может приостановить выполнение приложения, вызвав метод `pauseApp()`, или совсем прервать выполнение с помощью метода `destroyApp(boolean)`.
- **PAUSE** — приостановленное состояние. Приложение прекращает выполнение задач и освобождает часть ресурсов. Выполнение приложения может быть возобновлено системой с помощью метода `startApp()` или полностью остановлено методом `destroyApp(boolean)`.
- **DESTROYED** — прерванное состояние. Приложение остановлено и уже не может перейти в другие состояния. Все используемые ресурсы освобождаются.

«Жизненный цикл» приложения может быть представлен схемой, показанной на рис. 2.1.

Система управляет сменой состояний приложения с помощью абстрактных методов класса `MIDlet`, обязательных к реализации. Кроме того, класс `MIDlet` предоставляет нам еще несколько методов для контроля над состоянием приложения и получения информации о нем:

- `abstract void startApp()` — вызывается системой при переходе мидлета в активное состояние. Метод может быть вызван только из приостановленного состояния;
- `abstract void pauseApp()` — вызывается системой при переходе мидлета в приостановленное состояние. Приложение должно освободить совместно используемые ресурсы;
- `abstract void destroyApp(boolean unconditional)` — вызывается при переходе мидлета в прерванное состояние. Приложение должно освободить все ресурсы и сохранить все необходимые данные в постоянном хранилище. Если аргумент `unconditional` задает значение `true`, то приложение будет прервано в любом слу-

чае. Если передан аргумент `false`, то система может отказаться переводить приложение в прерванное состояние и сформировать исключение `MIDletStateException`;

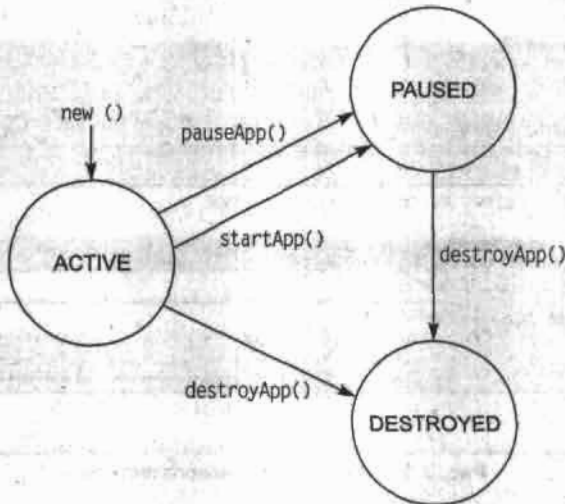


Рис. 2.1. «Жизненный цикл» приложения

- `void resumeRequest()` — используется для оповещения системы о желании приложения перейти в активное состояние, когда система будет готова перевести мидлет в активное состояние, она сама вызовет метод `startApp()`. Метод `resumeRequest()` вызывается приложением в приостановленном состоянии, в котором поддерживается обработка таймера;
- `void notifyPaused()` — приложение информирует систему о переходе в приостановленное состояние. Метод не будет иметь эффекта, если приложение еще не запущено или уже прервано;
- `void notifyDestroyed()` — мидлет информирует систему о переходе в прерванное состояние. С этого момента система будет считать все ресурсы свободными;
- `String getAppProperty(String key)` — возвращает параметр приложения, хранящийся в файле описателя мидлета, в виде строки. Аргумент `key` задает имя необходимого параметра. Если запрашиваемый параметр не задан, то возвращается значение `null`. Имена параметров мы рассмотрим позже вместе со структурой файла описателя.

Создание мидлета

Итак, для создания мидлета остановимся на одном из перечисленных в первой главе средств разработки J2ME Wireless Toolkit (WTK), с помощью которого реализованы все приведенные в этой книге примеры. После установки Wireless Toolkit запускаем его основную утилиту KToolbar. Доступные после запуска действия — это **NewProject** (Новый проект) и **OpenProject** (Открыть проект). Можно

открыть демонстрационные проекты и на готовых примерах рассмотреть возможности и особенности языка, но мы создадим новый проект и назовем его MyFirstMIDlet. Так же назовем и основной класс мидлета (MIDlet Class Name) в окне создания нового проекта (рис. 2.2).

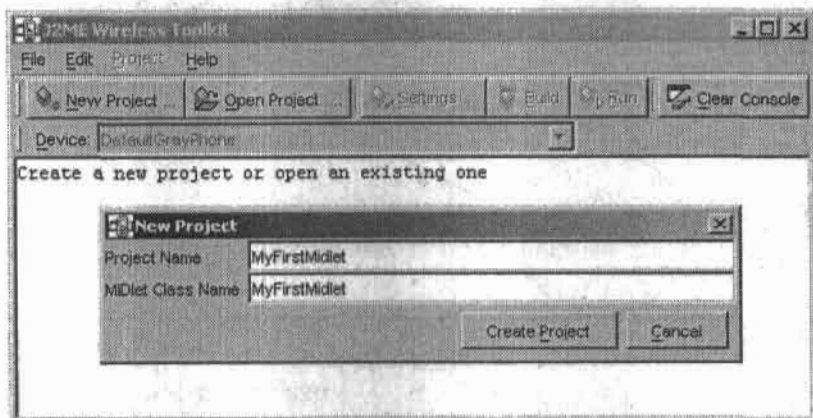


Рис. 2.2. Окно создания нового проекта

После нажатия кнопки Create Project (Создать проект) появится диалоговое окно ввода информации для файла — описателя приложения (рис. 2.3). Диалоговое окно состоит из нескольких вкладок, первая из которых — Required (Обязательные) — содержит обязательные атрибуты файла-описателя:

- MIDlet-Jar-Size — размер jar-файла в байтах. Обратите внимание, что данный атрибут прописывается в файле описателя автоматически средствами WTK при упаковке проекта. Будьте аккуратны при ручном редактировании jar-архива: несовпадение заявленного и реального размеров jar-файла послужит причиной ошибки при загрузке приложения в телефон;

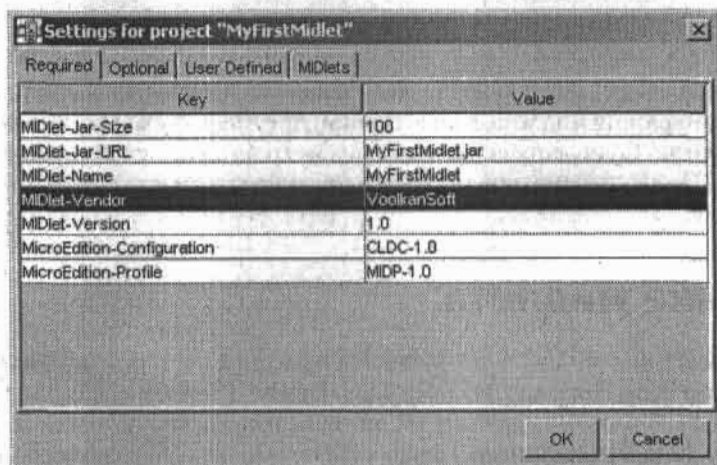


Рис. 2.3. Определение обязательных атрибутов приложения

- **MIDlet-Jar-URL** — сетевой адрес и название jar-файла. В нашем случае это поле содержит лишь имя jar-файла;
- **MIDlet-Name** — имя мидлета (или комплекта мидлетов), которое будет показано пользователю в меню запуска приложения;
- **MIDlet-Vendor** — разработчик приложения (организация или частное лицо);
- **MIDlet-Version** — версия мидлета в формате <major>.<minor>.<micro>; может быть использована при установке и обновлении приложения;
- **MicroEdition-Configuration** — требуемая конфигурация J2ME для выполнения мидлета. В рамках нашей книги используется конфигурация CDLC-1.0;
- **MicroEdition-Profile** — требуемый профайл J2ME для выполнения мидлета. Здесь мы рассматриваем работу с профайлом MIDP-1.0.

Поля ввода значений необязательных атрибутов расположены на вкладке **Optional (Факультативные)**:

- **MIDlet-Data-Size** — минимальное количество байт данных для постоянного хранения, требуемое мидлетом. Значение поля по умолчанию — ноль;
- **MIDlet-Delete-Confirm** — определяет, должна ли система запрашивать подтверждение пользователя при удалении приложения из устройства;
- **MIDlet-Description** — описание мидлета или набора мидлетов;
- **MIDlet-Icon** — имя графического файла .png в jar-архиве, который будет использован системой для данного мидлета в меню запуска приложений;
- **MIDlet-Info-URL** — сетевой адрес с описанием приложения или информацией о разработчике;
- **MIDlet-Install-Notify** — указывает, должна ли система уведомлять пользователя перед установкой нового мидлета.

Разработчик также имеет возможность определить собственные атрибуты приложения на вкладке **User Defined (Определенные пользователем)**. Имена атрибутов не должны начинаться с префикса **MIDlet-** и могут быть получены во время выполнения приложения с помощью метода `getAppProperty(String key)` класса **MIDlet**.

После того как все необходимые атрибуты указаны, нажимаем кнопку **OK** и получаем сообщение о том, что проект был успешно создан. Разберемся, что в данный момент произошло. Для этого перейдем в папку, где установлен **Wireless Toolkit**. В каталоге проектов **apps** появилась новая папка с именем нашего проекта **MyFirstMidlet**. В корневом каталоге нашего проекта были автоматически созданы еще несколько папок. Рассмотрим их назначение:

- **/bin** — папка для хранения файлов приложения: jar-архив приложения, файл описателя приложения .jad и файл манифеста **MANIFEST.MF**. Автоматически созданные файлы **MyFirstMidlet.jad** и **MANIFEST.MF** содержат указанные при создании проекта атрибуты;
- **/res** — файлы ресурсов, используемые приложением во время выполнения. Файлы могут содержать текстовые или бинарные данные, картинки в формате **PNG (Portable Network Graphics)**;
- **/src** — исходные файлы приложения, содержащие собственно код программы. Файлы должны иметь расширение **.java**.

Пришло время перейти непосредственно к программированию, уверен, что вам уже надоело читать это скучное введение и уже не терпится перейти со своим телефоном «на ты». К сожалению, среда Wireless Toolkit не содержит текстового редактора, поэтому нам потребуется любой другой редактор, на худой конец обычный блокнот. Достаточно простые и удобные в обращении, с моей точки зрения, такие редакторы, как UltraEdit или EditPlus, хотя опять же это дело вкуса, данный вопрос не принципиален, можете пользоваться и блокнотом.

Импорт классов и пакетов

Начнем с того, что классы, предоставляемые нам реализацией J2ME, объединены в пакеты, представляющие группу классов определенной тематики. Для компилятора имя класса составляют название класса и полное название пакета, где он содержится. Такое имя может быть достаточно длинным, поэтому можно импортировать класс с помощью оператора `import`, а в дальнейшем пользоваться лишь названием класса. Синтаксис оператора `import` достаточно простой:

```
import <имя пакета>.<имя класса>;
```

Таких операторов может быть сколько угодно. Они должны располагаться в самом начале программы. Существует возможность импортировать сразу все классы, содержащиеся в пакете. Для этого в операторе `import` вместо определенного имени класса следует указать символ `*`:

```
import <имя пакета>.*
```

Для знатоков языка C++ и основ объектно-ориентированного программирования отметим, что `import` не является аналогом директивы `include` языка C, поскольку никакие файлы во время импорта не подключаются. Реализация J2ME, которую мы рассматриваем, включает в себя следующие пакеты, классы которых мы будем рассматривать подробно на протяжении всей книги:

- `java.io` — классы работы с потоками ввода и вывода данных;
- `java.lang` — классы типов данных языка J2ME, а также некоторые системные классы;
- `java.util` — классы дополнительных утилит, реализующие структуры данных, а также работу со временем;
- `javax.microedition.io` — классы, обеспечивающие сетевые соединения;
- `javax.microedition.lcdui` — классы пользовательского интерфейса профайла MIDP;
- `javax.microedition.midlet` — содержит единственный класс `MIDlet`, являющийся основным классом мобильного приложения;
- `javax.microedition.rms` — классы поддержки долговременного хранения данных.

Итак, первой строчкой нашей программы мы импортируем основной класс мобильного приложения:

```
import javax.microedition.midlet.MIDlet;
```

Далее реализуем свой собственный класс `MyFirstMidlet`, расширяющий класс `MIDlet`. Как уже говорилось, обязательны к реализации три абстрактных метода класса

MIDlet. Они могут быть и пустыми, но присутствовать в нашем классе обязаны. В методе `startApp()` получим атрибут имени приложения с помощью метода `getAppProperty(String key)` класса MIDlet и отобразим его области системных сообщений вместе с сакраментальным «Hello, World!».

Область системных сообщений доступна нам с помощью поля `out` системного класса `System`. Поле `out` является потоком вывода системной информации и поддерживает вывод данных всех примитивных типов, а также строковых данных в область системных сообщений с помощью методов `print` и `println`, которые различаются лишь форматом вывода. Кроме этого, класс `System` содержит поле `err`, являющееся потоком для вывода сообщений об ошибках, а также несколько методов для доступа к системной информации. Как оформляются комментарии, вы без труда догадаетесь сами. Таким образом, весь код нашей программы будет выглядеть следующим образом:

```
import javax.microedition.midlet.MIDlet;

public class MyFirstMidlet extends MIDlet
{
    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
        // уведомить систему о завершении
        notifyDestroyed();
    }

    public void startApp() {
        // получить строку атрибута имени мидлета
        String name = getAppProperty("MIDlet-Name");
        // вывести сообщение в системную область
        System.out.println("MIDlet " + name + " says: Hello, World!");
    }
}
```

Обратим внимание на то, что мы не импортировали классы `String` и `System` из пакета `java.lang`, но без особых проблем используем их в нашей программе. Все верно. Дело в том, что пакет `java.lang` просматривается компилятором всегда и в импорте не нуждается.

Компиляция и запуск мидлета

После того как весь код нашего первого приложения написан, сохраним получившийся файл в папке исходных файлов приложения `/src` с именем `MyFirstMidlet.java`. Заметим, что имя файла должно полностью совпадать с именем основного класса приложения, иначе компилятор выдаст следующую ошибку: `Class MyFirstMidlet is public, should be declared in a file named MyFirstMidlet.java`.

Половина дела сделана, хотя пока что написанная программа является не более, чем простым текстом, который теперь нужно преобразовать в настоящее мобильное приложение. Этим вопросом занимается компилятор — специальная программа, переводящая написанный нами текст в машинные команды, понимаемые устройством.

Для того чтобы откомпилировать наше приложение, следует выбрать команду Build (Компилировать) на панели управления главного окна KToolbar. В основном окне утилиты KToolbar мы увидим результат компиляции — количество ошибок в программе, местонахождение и описание каждой из них. Если все в порядке, то отобразится сообщение о том, что компиляция прошла успешно (рис. 2.4).

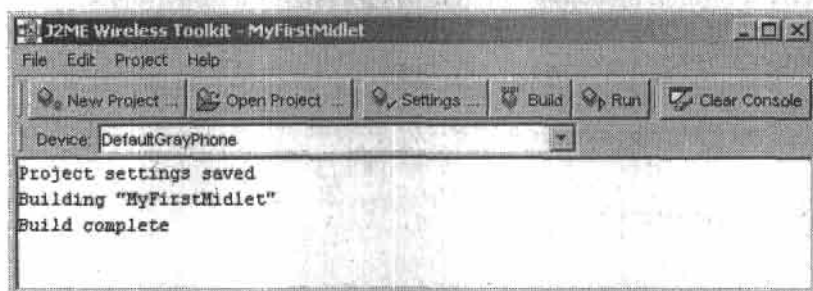


Рис. 2.4. Сообщение об успешном завершении компиляции

На этом этапе в папке нашего проекта было создано еще несколько директорий, в одну из которых, /classes, и было помещено откомпилированное приложение, представленное файлом MyFirstMidlet.class.

Сгенерированный компилятором файл уже является программой, готовой к исполнению на эмуляторе телефона. Чтобы запустить написанное нами приложение, в списке Device (Аппарат) панели управления утилиты KToolbar нужно выбрать один из стандартных или установленных нами эмуляторов. Эмулятор вместе с созданным нами приложением запускается с помощью команды Run (Запустить) панели управления. По умолчанию приложение запустится в эмуляторе с многообещающим названием «DefaultGrayPhone».

Поскольку никаких особенных действий наша программа не выполняет, то при ее запуске на эмуляторе ничего не произойдет. Однако, если все прошло успешно, то мы увидим желаемый результат в области системных сообщений утилиты KToolbar (рис. 2.5).

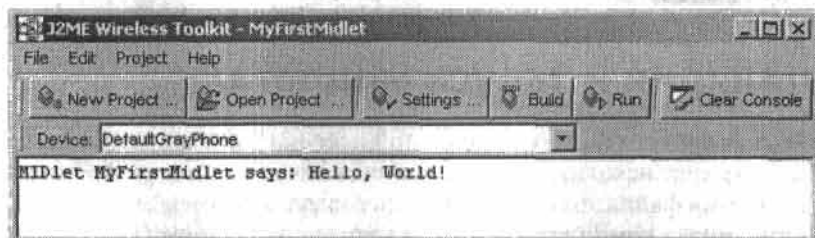


Рис. 2.5. Результат работы программы «Hello World!»

После закрытия эмулятора в область системных сообщений выводится разнообразная статистика. Оставим пока ее в покое, так как пользы сейчас нам от нее никакой.

Упаковка мидлета

Заключительным этапом создания полноценного мобильного приложения, готового для загрузки в телефон, является упаковка созданных файлов в jar-архив, который уже пригоден для загрузки в мобильный телефон. Для упаковки приложения следует выбрать в меню Project (Проект) утилиты KToolbox команду Create Package (Создать архив).

После упаковки в папке /bin нашего приложения появился файл MyFirstMidlet.jar. Это и есть нужный нам jar-архив. Современные архиваторы типа WinZIP или WinRAR поддерживают этот формат и позволяют нам заглянуть внутрь сформированного архива. Ничего нового там мы не увидим, только все тот же файл MyFirstMidlet.class и файл манифеста в папке META-INF. В таком виде приложение уже готово к загрузке в телефон с помощью средств, рассмотренных в предыдущей главе. Особого смысла загружать наше приложение в телефон нет, поскольку никакой реакции аппарата мы не увидим, так что отложите эту процедуру до следующей главы.

На этом процесс создания мобильного приложения завершен. Мы рассмотрели все этапы, от написания текста программы до тестирования приложения на эмуляторе, упаковки и загрузки в телефон. Все эти этапы, кроме написания текста, мы проделали с помощью среды разработки J2ME Wireless Toolkit.

Обратим внимание, что эмулятор, конечно, необходим в разработке приложений, но далеко не достаточен. Как показывает практика, реальные аппараты могут в любой момент «подложить нам свинью» в том месте, где ее совсем не ждешь. Поэтому нужно как следует протестировать все функции программы на реальном телефоне.

* * *

Итак, если у вас все получилось, то самое трудное уже позади. В программировании, как и в любом деле, главное — начать, а дальше будет только легче. Не рекомендую продолжать чтение книги, не выполнив первого задания, хотя это ваше личное дело. В дальнейшем вас ждет еще масса интересных примеров с их подробным описанием. Не лишайте себя удовольствия, учитесь на чужих ошибках.

Глава 3

Пользовательский интерфейс высокого уровня

После того как мы познакомились с основной структурой мидлета, будем постепенно продвигаться в наших познаниях. Java — язык объектно-ориентированный, поэтому, изучив все его объекты, мы сможем конструировать из них, как из кубиков, большие многофункциональные приложения. Первым делом разберемся с тем, как выводить информацию на экран телефона, чтобы демонстрировать результат работы наших приложений. Выводимая информация может быть разнообразной: текст, картинки, средства общения (интеракции) с пользователем.

Все объекты, которые можно отобразить на экране, наследуются из общего класса, название которого говорит само за себя — `javax.microedition.lcdui.Displayable`. Дальнейшая эволюция классов, наследуемых из класса `Displayable`, идет в двух направлениях:

1. Класс `Canvas` предоставляет пользователю интерфейс низкого уровня, организует непосредственный доступ к прорисовке экрана. Подробнее на этом классе мы остановимся в следующей главе.
2. Класс `Screen` является родителем для классов пользовательского интерфейса высокого уровня. Это четыре класса: `Alert`, `Form`, `List`, `TextBox`, объекты которых являются стандартными средствами связи с пользователем. Внешний вид таких объектов зависит от конкретной модели телефона. Свобода действий над этими объектами, изменение их вида, свойств и поведения ограничены функциями реализации J2ME. Иерархию отображаемых объектов можно представить следующей схемой (рис. 3.1).

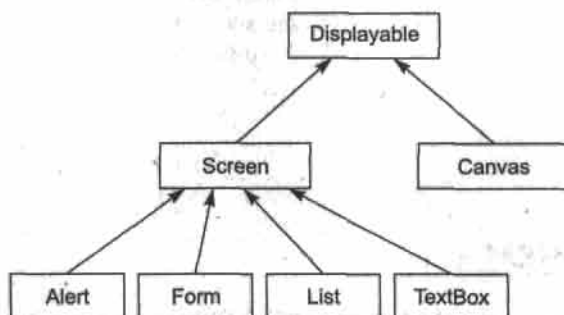


Рис. 3.1. Иерархия отображаемых объектов J2ME

Класс Form

В этой главе мы рассмотрим работу с формой (объектом класса `Form`) на примере программы фотоальбома. Форма является определенного рода контейнером, который может содержать различные визуальные элементы, такие как текстовые строки, поля символьного ввода, картинки, меню выбора, шкалы.

С точки зрения объектов, элементы, которые можно добавить в форму, должны быть порождены одним из следующих классов: `String`, `Image` или `Item`. Элемент класса `String` представляет собой статическую строку, автоматически отформатированную согласно размерам экрана конкретного аппарата. Элемент класса `Image` представляет собой картинку. Фактически, оба этих элемента дублируются объектами классов, порожденных из класса `Item`.

Класс `Item` является базовым для целой группы элементов, которые могут быть добавлены в форму: `ChoiceGroup` (меню выбора), `DateField` (поле ввода даты и времени), `Gauge` (шкала), `ImageItem` (изображение), `StringItem` (статическая строка), `TextField` (поле ввода).

Внутренняя реализация формы скрыта от программиста, мы можем контролировать лишь порядок отображения элементов. Расположение элементов на экране, навигация между элементами, вертикальная прокрутка экрана при необходимости контролируются внутренней реализацией и зависят от конкретной модели аппарата. Работа с формой осуществляется с помощью следующих методов класса `Form`:

- `Form(String title)` — конструктор; создает пустую форму с заголовком `title`;
- `Form(String title, Item[] items)` — конструктор; создает форму с заголовком `title`, содержащую элементы массива `items`;
- `int append(Image img)` — добавляет объект изображения `img` в форму, возвращает индекс, присвоенный формой элементу;
- `int append(String str)` — добавляет статическую строку `str` в форму, возвращает индекс, присвоенный формой элементу;
- `int append(Item item)` — добавляет элемент `item` в форму, возвращает индекс, присвоенный формой элементу. Следует отметить, что один экземпляр элемента может быть добавлен лишь единожды и только в одну форму;
- `void delete(int itemNum)` — удаляет из формы элемент с индексом `itemNum`. Нумерация элементов начинается с нуля;
- `Item get(int itemNum)` — возвращает элемент формы с индексом `itemNum`, сама форма при этом остается неизменной;
- `void insert(int itemNum, Item item)` — вставляет в форму элемент `item` в позицию, заданную переменной `itemNum`;
- `void set(int itemNum, Item item)` — заменяет в форме элемент с индексом `itemNum` на элемент `item`;
- `void setItemStateListener(ItemStateListener iListener)` — задает блок прослушивания элемента `iListener`. Поскольку реализация элементов скрыта от нас, блок прослушивания элемента — это единственное средство контроля непосредственно во время ввода информации. Как только данные элемента изменяются пользователем, который выбирает новый пункт меню или вводит очередной символ,

автоматически вызывается функция блока прослушивания `itemStateChanged(Item item)`. Следует отметить, что функция вызовется только при изменении элемента пользователем. Если изменение происходит программно, то вызова не произойдет;

- `int size()` — возвращает количество элементов в форме.

Все объекты иерархии класса `Screen`, в том числе и форма, обладают необязательными заголовком и бегущей строкой. При создании формы в конструкторе можно указать заголовок отображаемого объекта, а можно оставить заголовок пустым, передав аргумент `null`. В процессе работы с формой можно получить ее заголовок с помощью метода `String getTitle()` и установить новый заголовок, используя метод `void setTitle(String s)`.

Бегущая строка представлена объектом класса `Ticker`. Текст бегущей строки передается в конструкторе объекту класса. Поместить бегущую строку в отображаемый объект можно с помощью метода `void setTicker(Ticker ticker)`, получить объект бегущей строки — используя метод `Ticker getTicker()`. Программное редактирование бегущей строки осуществляется с помощью методов класса `Ticker`:

- `void setString(String str)` — изменить текст бегущей строки;
- `String getString()` — получить текст бегущей строки.

Класс Display

Остановимся на вопросе, каким образом рассмотренные объекты отображаются на экране. Этим процессом руководит менеджер дисплея — объект класса `Display`. Он создается автоматически реализацией `MIDP` при запуске мидлета и сохраняется до вызова функции `destroyApp()`. Самим такой объект создать нельзя, зато можно получить ссылку на него с помощью метода `getDisplay(MIDlet)`, который принимает в качестве аргумента ссылку на текущий мидлет. В языке Java, как и в C++, ссылка на текущий объект обозначается ключевым словом `this`.

Когда мы получили ссылку на `Display`, любой объект, наследованный из класса `Displayable`, может быть выведен на экран с помощью метода `setCurrent(Displayable)`. Ввиду ограниченности ресурсов мобильных аппаратов одновременно на экране может быть отображен только один объект, перекрытие и наложение окон не поддерживаются. Таким образом, любое приложение может быть рассмотрено как заданная последовательность отображаемых объектов.

Во время отображения формы через менеджер дисплея обновление экрана происходит автоматически. Это значит, что при изменении содержания формы или ее элементов нет необходимости вызывать какие-либо дополнительные функции, так как изменения сразу будут отображены на экране. Это относится ко всем объектам, порожденным классом `Screen`.

Отображаем картинку

Приведенной информации вполне достаточно, чтобы написать программу, отображающую картинку на экране телефона. Заметим, что не все картинки одина-

ково полезны. Во-первых, изображение должно быть в формате PNG (Portable Network Graphics). Во-вторых, размер картинки не должен превышать размера дисплея вашего аппарата. Если для работы над мидлетом выбран WTK (Wireless Toolkit), то после создания нового проекта картинка должна быть помещена в папку ресурсов приложения /res.

Структуру мидлета мы рассмотрели в прошлой главе. Помним, что основной класс мидлета должен быть наследован из абстрактного класса MIDlet. Не забываем также о его функциях, обязательных к реализации.

Для того чтобы вывести картинку на экран, нам потребуется несколько объектов, которые будут объявлены как члены класса. Предварительно импортируем необходимые библиотеки:

```
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.Display;
```

```
...
```

```
private Form form;           // форма, отображаемая на экране
private Image image;         // картинка для отображения
private Display display;      // менеджер дисплея
```

Далее в функции startApp() реализуем следующие действия:

- получим ссылку на менеджер дисплея;
- создадим объект картинки;
- создадим новую форму;
- добавим картинку в форму;
- отобразим форму на экране через менеджер дисплея.

Код функции будет выглядеть следующим образом:

```
public void startApp() {
    // получить ссылку на менеджер дисплея
    display = Display.getDisplay(this);

    try {
        // создать картинку из файла flower.png
        image = Image.createImage("/flower.png");
    }
    catch (IOException ioe) {
        // обработать исключительную ситуацию.
        // если файл не может быть открыт
        System.out.println(ioe.getMessage());
    }

    // создать новую форму
    form = new Form("SlideShow");
    // добавить картинку в форму
    form.append(image);
    // вывести форму на экран
    display.setCurrent(form);
}
```



Рис. 3.2. Вывод картинки на экран телефона

Компилируем приложение, как это было рассмотрено ранее, запускаем на стандартном эмуляторе из WTK и получаем вот такой результат (рис. 3.2).

Все это просто замечательно, но ничего нового мы пока что не изобрели. С таким же успехом можно было просто поместить картинку в галерею, если таковая имеется. Мы пойдем дальше, задействуем клавиши телефона, чтобы организовать смену картинок в нашем фотоальбоме.

Класс Command

Итак, сейчас мы работаем с формой, относящейся к иерархии класса `Displayable`, который является базовым для всех отображаемых объектов. Кроме стандартных визуальных средств интеракции, объекты, порожденные этим классом, поддерживают обработку событий, возникающих в результате действий пользователя. К отображаемому объекту, в том числе и к форме, могут быть привязаны различные команды, вызов которых приведет к некоторым действиям.

Команды представлены объектами класса `Command` из пакета `javax.microedition.lcdui`. Создается команда с помощью конструктора: `Command(String label, int commandType, int priority)`, где:

- `label` — название команды для отображения ее на экране;
- `commandType` — тип команды;
- `priority` — приоритет команды.

Поскольку команды относятся к пользовательскому интерфейсу высокого уровня, то при создании команды мы не можем заранее связать ее с какой-то определенной клавишей телефона. Порядок вызова команды определит за нас реализация MIDP, мы лишь можем высказать свои пожелания с помощью параметров типа команды и ее приоритета.

Тип команды задает внутреннее представление намеченного использования команды. MIDP предусматривает следующие типы команд, определяемые константами класса `Command`:

- `BACK` — возврат к логически предыдущему экрану;
- `CANCEL` — выбор отрицательного ответа в диалоге, реализуемом текущим экраном;
- `EXIT` — выход из приложения;
- `HELP` — вызов помощи или подсказки;
- `ITEM` — команда имеет отношение к определенному элементу экрана;

- OK — выбор положительного ответа в диалоге, реализуемом текущим экраном;
- SCREEN — команда имеет отношение к отображаемому в настоящее время экрану;
- STOP — остановка выполняемой в настоящее время операции.

Заметим, что константы задают лишь тип команды. Они не предписывают никаких действий. Смена экрана, выход из приложения и прочие действия не выполняются автоматически, они должны быть реализованы в соответствующем месте. Тип лишь ориентирует реализацию MIDP, как разместить команды на экране и организовать их вызов в соответствии с общим стилем меню и функциональными клавишами конкретной модели телефона.

Приоритет задает преимущество перед другими командами. Это влияет на порядок отображения команд при объединении их в группу. Приоритет задается целым числом. Наименьшее число означает наивысший приоритет.

Такой вариант значительно ограничивает свободу наших действий, но решает массу проблем, связанных с транспортабельностью приложений. Теперь мы можем быть уверены, что на любом аппарате с поддержкой J2ME команды будут предложены пользователю корректно, с учетом общей политики отображения и стиля команд конкретного аппарата.

После создания объекта команды необходимо добавить его в форму, которую мы собираемся отображать на экране. Методы добавления и удаления команд реализует класс `Displayable`, то есть команды могут быть добавлены к любому отображаемому объекту:

- `void addCommand(Command cmd)` — добавляет команду `cmd` к отображаемому объекту. Будет ли команда привязана к функциональной клавише или объединена с другими командами в меню, зависит от реализации MIDP конкретного аппарата. Команда может быть добавлена в объект не более одного раза, но может быть включена одновременно в несколько объектов;
- `void removeCommand(Command cmd)` — удаляет команду `cmd` из отображаемого объекта.

Для примера проведем эксперимент на эмуляторе Nokia. Добавим в форму четыре однотипных команды с разным приоритетом:

```
Command c1 = new Command("Command1", Command.OK, 4);  
form.addCommand(c1);  
Command c2 = new Command("Command2", Command.OK, 3);  
form.addCommand(c2);  
Command c3 = new Command("Command3", Command.OK, 2);  
form.addCommand(c3);  
Command c4 = new Command("Command4", Command.OK, 1);  
form.addCommand(c4);  
display.setCurrent(form);
```

В результате все команды объединились в меню Options, привязанное к левой функциональной клавише телефона, при нажатии на которую появился список наших команд. Это аппаратно реализованный список, позволяющий выбрать одну из команд клавишей Select или вернуться к предыдущему экрану клавишей Back (рис. 3.3).



Рис. 3.3. Аппаратно реализованный список команд

Заметьте, что в соответствии с заданными приоритетами последняя добавленная команда находится первой в списке. Использование интерфейса высокого уровня приводит к тому, что та же самая программа, запущенная на другом аппарате, будет выглядеть совершенно иначе.

Интерфейс CommandListener

Как уже было замечено, сам объект команды не предписывает программе выполнение каких-либо действий, он является лишь связующим звеном между исполнителем команды и формой. Приемником и исполнителем команды может являться любой объект, реализующий интерфейс `commandListener`, который имеет всего лишь один метод: `void commandAction(Command c, Displayable d)`. Этот метод нам нужно написать самим и реализовать там связанные с командами действия. Метод `commandAction` вызовется автоматически при выборе пользователем одной из команд, которая и будет передана в метод в качестве аргумента.

В нашем случае удобно сделать так, чтобы основной класс мидлета содержал блок прослушивания команд в себе самом. В общем случае блок может содержаться в любом объекте. Таким образом, основной класс мидлета будет расширять класс `MIDlet` и одновременно реализовывать интерфейс `commandListener`:

```
public class SlideShow extends MIDlet implements CommandListener {
```

```
...
```

Приемник команд, как и сами команды, должен быть привязан к объекту типа `Displayable`, в нашем случае это форма. В каждый момент времени отображаемый объект может иметь только один приемник. Связь приемника команд с отображаемым объектом осуществляется с помощью метода класса `Displayable`: `void setCommandListener(CommandListener l)`. В нашем случае, когда основной класс мидлета реализует блок прослушивания команд, в качестве аргумента передается ссылка на главный объект. Так же как и в случае с менеджером дисплея, мы опять воспользуемся ключевым словом `this`.

Теперь у нас есть все необходимые средства для реализации полноценного фотоальбома. Подготовим картинки для фотоальбома, учитывая размер экрана и максимальный объем приложения вашей модели телефона. Назовем картинки, которые мы хотим демонстрировать, так: `1.png`, `2.png` и т. д. — и разместим их в папке `/res`. Заодно добавим два члена класса `SlideShow`: `int slideNum` и `int maxSlideNum`,

которые будут содержать текущий номер картинки и общее количество картинок соответственно.

Далее реализуем обработку команд и добавим логику для смены картинок на экране при нажатии клавиш. В конечном итоге программа полноценного фотоальбома будет выглядеть следующим образом:

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.Ticker;
import java.io.IOException;

public class SlideShow extends MIDlet implements CommandListener
{
    private Display display;    // менеджер дисплея
    private Form form;         // отображаемая форма
    private Command next;      // команда перехода к следующей картинке
    private Command back;      // команда перехода к предыдущей картинке
    private Image image;       // объект картинки
    private int slideNum=1;     // номер текущей картинки
    private int maxSlideNum=5; // общее количество картинок
    ...

    public void startApp()
    {
        // получить ссылку на менеджер дисплея
        display = Display.getDisplay(this);
        // создать новую форму без заголовка
        form = new Form(null);
        // установить приемник команд для формы
        form.setCommandListener(this);
        // создать команду перехода к следующей картинке
        next = new Command("Next", Command.OK, 1);
        // добавить команду в форму
        form.addCommand(next);
        // создать команду возврата к предыдущей картинке
        back = new Command("Back", Command.BACK, 1);
        // добавить команду в форму
        form.addCommand(back);
        // создать объект бегущей строки
        Ticker t = new Ticker("My Photoalbum");
        // добавить бегущую строку в форму
        form.setTicker(t);
        // добавить в форму первую картинку
        setImage("/1.png");
    }
}
```

```

        // отобразить форму на экране
        display.setCurrent(form);
    }
    // метод обработки команд
    // реализует действия, предписанные командами
    public void commandAction(Command c, Displayable d) {
        // команда перехода к следующей картинке.
        // если текущая картинка не последняя
        if (c == next && slideNum < maxSlideNum)
            // увеличить номер текущей картинки
            slideNum++;
        // команда возврата к предыдущей картинке.
        // если текущая картинка не первая
        if (c == back && slideNum > 1)
            // увеличить номер текущей картинки
            slideNum--;
        // удалить из формы текущую картинку
        form.delete(0);
        // получить имя файла картинки из ее номера
        // и добавить полученную картинку в форму
        setImage("/"+Integer.toString(slideNum)+".png");
    }

    // функция setImage принимает имя файла картинки.
    // создает объект картинки и добавляет в форму
    public void setImage(String path)
    {
        try {
            // создать картинку из файла, переданного в аргументе
            image = Image.createImage(path);
        }
        // обработать исключительную ситуацию.
        // если файл не может быть открыт
        catch (IOException ioe) {
            System.out.println(ioe.getMessage());
        }
        // добавить картинку в форму
        form.append(image);
    }
}

```

Отметим, что для смены изображения на экране мы лишь помещаем новую картинку в отображаемую на экране форму, после чего картинка сразу появляется на экране (рис. 3.4). Заметим, что в данном эмуляторе изображение не поместилось на экране, поэтому реализация организовала вертикальную прокрутку экрана,

обозначенную стрелочкой вниз. Горизонтальная прокрутка экрана в реализациях, как правило, отсутствует.

Поздравляю! Если вы не поленились оторваться от книжки и реализовать вышеописанное, то загрузите получившийся мидлет в телефон, и вы будете всегда иметь под рукой маленький фотоальбом. Можно пойти дальше и добавить нехитрую логику перехода к первой картинке после демонстрации последней и наоборот.

К сожалению, стандартная реализация J2ME не поддерживает доступ к внутренним параметрам телефона, к записной книжке или экранной заставке, поэтому, в общем случае, установить отображаемую картинку в качестве фона-заставки телефона программно мы не можем. Но не стоит отчаиваться, потому что подобная функция может быть реализована производителем аппарата в специализированном пакете функций API. На некоторых таких пакетах подробнее мы остановимся в одной из заключительных глав.

С картинками нужно быть аккуратными, ведь многие модели телефонов накладывают серьезные ограничения на размер приложения, а картинки в этом плане очень ресурсоемки. Здесь придется искать оптимальное соотношение параметров цена и качество: платим в данном случае мы объемом памяти, занимаемым картинкой. Именно поэтому оптимизацию мидлетов начинают с оптимизации картинок, различными средствами стараясь уменьшить занимаемый ими объем памяти.

На сегодняшний день существует масса различных средств для обработки картинок, от классического PhotoShop до специализированных ImageOptimizer, которые порой дают лучшие результаты в плане оптимизации, чем монстры дизайна. Часто производители телефонов предлагают свои средства для редактирования и оптимизации изображений. Например, Nokia PC Suite содержит свой замечательный Nokia Image Converter.

Итак, основа проектирования любого приложения — создание структуры демонстрирования различных отображаемых объектов и реализация логики переходов между этими объектами. Логiku переходов можно организовать с помощью пользовательского интерфейса высокого уровня, который мы рассмотрели в этой главе. Для этого нужно спроектировать отображаемые объекты, добавить к ним требуемые команды и реализовать интерфейс прослушивания и исполнения команд. Приложения, организованные таким образом, более транспортабельны и могут быть перенесены на различные модели телефонов. Пользовательский интерфейс низкого уровня мы рассмотрим в следующей главе.



Рис. 3.4. Окончательный вид фотоальбома

Глава 4

Работа с графикой

В прошлой главе мы рассмотрели пример работы с пользовательским интерфейсом высокого уровня, который представлен иерархией класса `Screen`. При работе с объектами такого типа мы не имели прямого доступа к экрану аппарата, а пользовались формой как визуальной абстракцией высокого уровня. Изменения изображения на экране осуществлялись с помощью методов формы и добавленных в нее объектов. То есть, работая с высокоуровневым интерфейсом, при всем желании нарисовать поперек экрана, например, жирную синюю полосу мы не сможем.

Решить проблему непосредственного доступа к экрану нам поможет низкоуровневый пользовательский интерфейс, который представлен второй веткой иерархии отображаемых объектов типа `Displayable` — классом `Canvas`.

Класс `Canvas`

Класс `Canvas` является абстрактным. Это значит, что мы не можем просто создать объект этого класса. Если все же попытаться создать такой объект, то компилятор подтвердит вышесказанное вот такими словами:

```
javax.microedition.lcdui.Canvas is abstract; cannot be instantiated  
Canvas c = new Canvas();
```

В объектно-ориентированном программировании абстрактными называются такие классы, которые содержат в себе хотя бы один абстрактный метод. Использовать абстрактные классы можно только как основу для новых классов, в которых обязательна реализация абстрактного метода.

В классе `Canvas` абстрактным является метод `abstract void paint(Graphics g)`, который отвечает за перерисовку экрана и получает в качестве аргумента объект класса `Graphics`, который создается автоматически при инициализации объекта `Canvas`.

Класс `Graphics`

Собственно класс `Graphics` и предоставляет возможности низкоуровневого графического рисования. Возможности класса `Graphics` далеко не безграничны: он содержит методы для рисования и заливки базовых геометрических фигур, таких как линии, прямоугольники и дуги. Также поддерживаются возможности отображения текстовых символов и смены цвета для рисования.

Теперь все в наших руках, включая и все точки экрана, каждая из которых, кстати, имеет свои координаты по осям x и y . Начало координат (точка 0, 0) находится в левом верхнем углу экрана. То есть нужно учесть, что ось y направлена вниз.

А теперь перейдем к рисованию и рассмотрим методы класса Graphics, которые могут нам понадобиться. В первую очередь заметим, что размеры экрана у разных моделей могут серьезно различаться, поэтому сразу будем писать транспортабельные приложения. Таким образом, первым делом нам требуется получить размеры экрана конкретного аппарата. Делается это с помощью двух методов:

- `int getClipHeight()` — возвращает высоту текущей области для рисования;
- `int getClipWidth()` — возвращает ширину текущей области для рисования.

Сам класс Canvas также содержит методы для определения размеров отображаемой области экрана, которые не зависят от текущей области для рисования и остаются неизменными во время работы приложения:

- `int getHeight()` — возвращает высоту отображаемой области экрана;
- `int getWidth()` — возвращает ширину отображаемой области экрана.

Теперь, получив ширину и высоту области для рисования, мы можем использовать все точки с соответствующими координатами. По умолчанию текущей областью для рисования является весь экран, но и это можно изменить, используя метод `SetClip`:

- `void setClip(int x, int y, int width, int height)` — устанавливает новую прямоугольную область рисования с левым верхним углом в точке (x, y) , высотой `height` и шириной `width` пикселей.

Иногда удобно сместить начало координат из левого верхнего угла в более удобное место. Делается это с помощью метода `void translate(int x, int y)`, который переносит начало координат в точку с координатами (x, y) . Получить текущее положение начала координат относительно левого верхнего угла можно следующими методами: `int getTranslateX()` и `int getTranslateY()`.

После того как мы определились с областью для рисования и началом координат, рассмотрим несколько методов для отображения на экране базовых геометрических фигур:

- `void drawLine(int x1, int y1, int x2, int y2)` — рисует линию из точки с координатами (x_1, y_1) в точку с координатами (x_2, y_2) . В классе Graphics нет метода для рисования одной точки, поэтому отдельная точка рисуется с помощью линии, начало и конец которой заданы одними и теми же координатами;
- `void drawRect(int x, int y, int width, int height)` — рисует прямоугольник с левым верхним углом в точке (x, y) , шириной `width` и высотой `height` пикселей;
- `void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)` — рисует прямоугольник с закругленными углами. Первые четыре аргумента аналогичны аргументам предыдущего метода, `arcWidth` задает диаметр закругляющей дуги вокруг оси x , `arcHeight` задает диаметр закругляющей дуги вокруг оси y ;
- `void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)` — рисует дугу, вписанную в прямоугольник с левым верхним углом в точке с координатами (x, y) . Параметр `startAngle` задает угол, с которого будет начи-

наться дуга, а параметр `arcAngle` задает угловое расстояние, на которое будет простирается дуга. Углы задаются в градусах. Таким образом, чтобы нарисовать окружность, нужно в квадрат вписать дугу с угловым расстоянием 360.

Линии, прямоугольники и дуги могут быть нарисованы двумя способами штрихования — сплошной и пунктирной линиями. Стиль штрихования задается с помощью метода `void setStrokeStyle(int style)`, в качестве аргумента `style` передается одна из констант класса `Graphics`: `SOLID` — для рисования сплошных линий, `DOTTED` — для рисования пунктирных линий. Получить текущий стиль штрихования позволяет метод `int getStrokeStyle()`.

До сих пор, по умолчанию, мы рисовали черным по белому, но в настоящее время аппаратов с цветными дисплеями, которые поддерживают технологию J2ME, гораздо больше, чем их монохромных собратьев. Поэтому не будем ограничивать свою фантазию и раскрасим экран нашего телефона веселыми красками.

Для начала необходимо узнать, цветной ли дисплей у телефона, который достался в распоряжение нашей программе. Эту информацию можно извлечь из менеджера дисплея, используя метод `boolean isColor()`, который вернет значение `true`, если аппарат поддерживает работу с цветом. Количество доступных цветов или, в некоторых моделях, количество градаций одного серого цвета можно получить с помощью метода `int numColors()`.

В отличие от других языков программирования, в J2ME константы для определения цвета не заданы, поэтому придется нам самим, как настоящим художникам, наносить на палитру три цветовых составляющих: красный, зеленый и синий (модель образования цвета RGB). Доля каждого из этих цветов задается числом от 0 до 255. Например, комбинация (0,0,255) задает синий цвет, а (0,0,0) — черный. Наши краски не кончаются, поэтому экспериментировать мы можем сколько угодно. Цвет рисования устанавливается с помощью двух методов класса `Graphics`:

- `void setColor(int red, int green, int blue)` — задает цвет рисования, заданный цветовыми составляющими `red`, `green`, `blue`;
- `void setColor(int RGB)` — устанавливает цвет, заданный цветовыми составляющими, объединенными в одно число RGB, где каждые два байта представляют свою составляющую. Например, красному цвету (255,0,0) будет соответствовать число RGB `0xFF0000`.

Узнать текущий цвет рисования можно опять же двумя способами: либо получить каждую цветовую компоненту по отдельности, используя методы `int getRedComponent()`, `int getGreenComponent()` и `int getBlueComponent()`, либо получить число RGB с помощью метода `int getColor()`.

Теперь мы можем рисовать не только черным, что, несомненно, скрасит нашу жизнь, но пока что только по белому. Изменить цвет фона или какой-то его части можно с помощью методов рисования закрашенных фигур:

- `void fillRect(int x, int y, int width, int height)` — рисует закрашенный прямоугольник;
- `void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)` — рисует закрашенный прямоугольник с закругленными углами;
- `void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)` — рисует закрашенный сектор.

Все аргументы методов рисования закрашенных фигур идентичны аргументам аналогичных методов рисования простых фигур. Фигуры закрашиваются текущим цветом, и если мы хотим очистить экран, то нужно установить белый текущим цветом и нарисовать закрашенный прямоугольник размером на весь экран.

Класс Graph

Самое время остановиться и попробовать использовать полученные знания на практике. В качестве примера нарисуем график параболы. Красная парабола на желтом фоне — разве это не прекрасно?!

Представим наш график новым классом Graph, который расширит класс низкоуровневого интерфейса Canvas и реализует его абстрактный метод paint. В методе paint мы зальем экран желтым фоном, нарисуем рамку и оси координат, сместим начало координат в центр экрана, а затем в цикле вычислим координаты для каждой точки параболы по формуле. Наш класс будет выглядеть следующим образом:

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Graphics;
...
public class Graph extends Canvas {
    // функция прорисовки экрана
    public void paint(Graphics g) {
        // получить ширину экрана
        int width = g.getClipWidth();
        // получить высоту экрана
        int height = g.getClipHeight();
        // установить текущий цвет желтым
        g.setColor(255,255,0);
        // нарисовать закрашенный прямоугольник
        // размером на весь экран
        g.fillRect(0,0,width,height);
        // установить текущий цвет черным
        g.setColor(0,0,0);
        // нарисовать рамку
        g.drawRect(0,0,width-1,height-1);
        // нарисовать оси координат
        g.drawLine(width/2,0,width/2,height);
        g.drawLine(0,height/2,width,height/2);
        // сместить начало координат в центр экрана
        g.translate(width/2,height/2);
        // установить текущий цвет красным
        g.setColor(255,0,0);
```



```

// для каждой точки по оси x
for(int x=-width/2; x<width/2; x++) {
    // вычислить значение y по формуле
    int y = -x*x/40;
    // нарисовать точку параболы
    g.drawLine(x,y,x,y);
}
}
}

```

Класс, отвечающий за рисование графика, готов. Осталось в стартовом методе приложения получить ссылку на менеджер дисплея и, как в случае с формой, вывести отображаемый объект на экран:

```

public void startApp() {
    // создать объект графика
    Graph graph = new Graph();
    // получить ссылку на менеджер дисплея
    display = Display.getDisplay(this);
    // вывести график на экран
    display.setCurrent(graph);
}

```

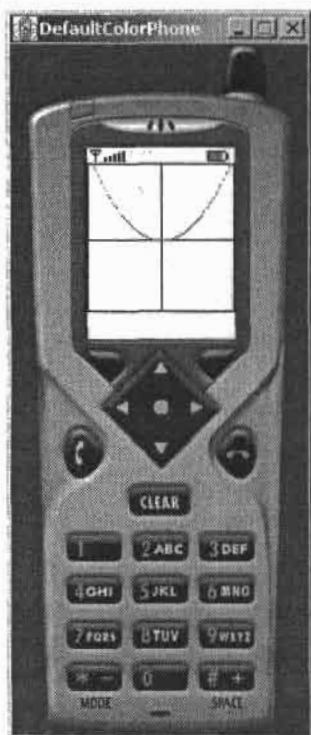


Рис. 4.1. Парабола, созданная средствами J2ME

Компилируем, запускаем и наслаждаемся полученным результатом (рис. 4.1).

Обратим внимание на формулу, по которой мы считывали координаты точек параболы: $-x^2/40$. Теперь попытаемся вспомнить школьный курс алгебры: если первый коэффициент квадратного многочлена отрицательный, то ветви параболы должны быть направлены вниз. Однако на эмуляторе мы видим, что ветви параболы направлены вверх. Дело в том, что в телефоне ось y направлена не вверх, как в школьных учебниках, а в обратном направлении.

Вторая проблема в том, что, как мы видим, область для рисования, предоставленная нам аппаратом, меньше, чем реальные размеры экрана, а сверху пристроилась незваная полоска с названием основного класса мидлета. Действительно, зачастую разработчики оставляют себе часть экрана для вспомогательной информации.

Полный экран можно получить двумя способами: используя функцию профайла MIDP2, обзор которого останется за переплетом этой книги, или задействовав специальную API-функцию производителя. На втором способе мы остановимся в одной из заключительных глав.

Рисование текста

Рисовать, конечно, здорово, но иногда на картинке требуется что-нибудь написать, например **GAME OVER**. В прошлой главе, когда мы отображали текст при помощи интерфейса высокого уровня, форматирование строки происходило автоматически. Теперь ответственность за размещение текста на экране, выравнивание и перенос строк целиком лежит на плечах программиста.

Позиция для текста задается при помощи координат точки размещения текста и точки привязки. Отображаемая строка заключается в воображаемый прямоугольник, точка привязки которого будет помещена в заданные координаты. Точка привязки определяется комбинацией вертикальной и горизонтальной констант класса `Graphics`.

- **BASELINE** — размещение нижней границы текста по координате *y*.
- **TOP** — размещение верхней границы текста по координате *y*.
- **LEFT** — размещение левой границы текста по координате *x*.
- **RIGHT** — размещение правой границы текста по координате *x*.
- **HCENTER** — размещение середины текста по координате *x*.

На рисунке показано расположение точек привязки на прямоугольнике, ограничивающем границы текста.

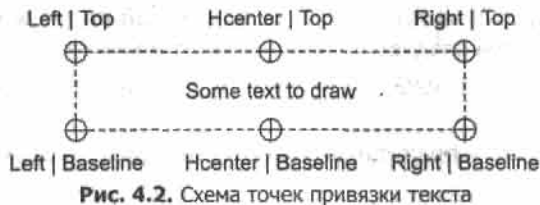


Рис. 4.2. Схема точек привязки текста

Класс `Graphics` предоставляет следующие методы рисования текста:

- `void drawChar(char character, int x, int y, int anchor)` — отображает символ `character`, размещая его точку привязки, заданную аргументом `anchor`, в точке с координатами `(x,y)`;
- `void drawChars(char[] data, int offset, int length, int x, int y, int anchor)` — рисует `length` символов массива `data`, начиная с символа с индексом `offset`. Логика размещения текста на экране остается прежней;
- `void drawString(String str, int x, int y, int anchor)` — рисует строку `str` в указанной позиции;
- `void drawSubstring(String str, int offset, int len, int x, int y, int anchor)` — рисует `len` символов строки `str`, начиная с символа с индексом `offset`.

Работу с текстом мы рассмотрим подробнее в дальнейшем на примере программы для чтения книг `BookReader`. Кроме базовых геометрических фигур и текста, как и в прошлой главе, мы можем вывести на экран картинку, представленную файлом в формате `PNG`. Теперь мы можем четко указать положение картинки на экране, используя константы привязки для рисования текста, а также одну допол-

нительную константу `VCENTER`, которая размещает точку привязки в середине изображения по оси y . Метод для отображения картинки:

- `void drawImage(Image img, int x, int y, int anchor)` — рисует картинку, заданную аргументом `img`, разместив точку ее привязки `anchor` в координатах (x, y) .

Обработка клавишных событий

Интерфейс низкого уровня позволяет нам отслеживать нажатия клавиш. Если в интерфейсе высокого уровня события формировались только для добавленных команд, то интерфейс низкого уровня формирует события при нажатии любой клавиши, причем теперь можно узнать, какая конкретно клавиша была нажата.

Класс `Canvas` также определяет блоки прослушивания событий другого уровня. Теперь в зависимости от действий пользователя будет автоматически вызван один из следующих методов класса `Canvas`:

- `void keyPressed(int keyCode)` — вызывается, если какая-либо клавиша была нажата;
- `void keyReleased(int keyCode)` — вызывается, если какая-либо клавиша была отпущена;
- `void keyRepeated(int keyCode)` — вызывается при длительном нажатии клавиши.

Все методы принимают аргумент, представляющий код нажимаемой клавиши. Коды клавиш задаются следующими константами класса `Canvas`:

- `KEY_NUM0, KEY_NUM1, KEY_NUM2, ..., KEY_NUM9` — представляют цифры на клавиатуре телефона;
- `LEFT, RIGHT, UP, DOWN` — представляют клавиши телефона, обозначенные стрелками;
- `KEY_POUND` — представляет клавишу с символом `#` (решетка);
- `KEY_STAR` — представляет клавишу с символом `*` (звездочка).

Для того чтобы привязать какие-то действия к желаемому событию, в классе-потомке класса `Canvas` необходимо переписать соответствующую функцию и там реализовать необходимые действия.

Перейдем от теории к практике. Рассмотрим подробнее обработку событий низкоуровневого интерфейса на примере программы управляемой точки, рисующей фигуры на экране мобильного телефона.

Реализуем класс `Point`, наследованный из класса `Canvas`, который будет содержать членами класса текущие координаты точки, размеры отображаемой области экрана, команду очистки экрана, а также флаг, управляющий очисткой экрана. Блок прослушивания событий низкого уровня будет отслеживать нажатия клавиш и менять текущие координаты точки соответственно нажатой цифре. Блок прослушивания команд будет поднимать флаг в случае вызова команды очистки экрана.

Код класса `Point` выглядит следующим образом:

```
public class Point extends Canvas implements CommandListener{
    private int x;           // координата x точки
```

```
private int y;      // координата y точки
private int width;  // ширина экрана
private int height; // высота экрана
private boolean clrFlag = true; // флаг очистки экрана
private Command clear; // команда очистки экрана
// конструктор класса Point
public Point() {
    // вызов конструктора родительского класса
    super();
    // получить ширину экрана
    width=getWidth();
    // получить высоту экрана
    height=getHeight();
    // установить текущие координаты точки в центр экрана
    x=width/2;
    y=height/2;
    // создать объект команды очистки экрана
    clear = new Command(«Clear», Command.OK, 1);
    // добавить команду очистки экрана
    addCommand(clear);
    // установить блок прослушивания команды
    setCommandListener(this);
}
// метод перерисовки экрана
public void paint(Graphics g) {
    // проверить флаг очистки экрана
    if (clrFlag) {
        // установить текущий цвет белым
        g.setColor(0xffffffff);
        // залить прямоугольник размером с экран
        g.fillRect(0, 0, width, height);
        // сбросить флаг очистки экрана
        clrFlag = false;
    }
    // установить текущий цвет точки красным
    g.setColor(255,0,0);
    // отобразить точку в текущих координатах
    g.drawLine(x,y,x,y);
}
// метод, реализующий блок прослушивания команды
public void commandAction(Command c, Displayable s) {
    // если сработала команда очистки
    if (c == clear) {
        // поднять флаг очистки экрана
        clrFlag = true;
        // установить те
```

```

        x=width/2;
        y=height/2;
        // инициировать перерисовку экрана
        repaint();
    }
}

// блок прослушивания событий низкого уровня
public void keyPressed(int keyCode) {
    switch (keyCode) {
        // сместить текущие координаты точки
        // в соответствии с нажатой клавишей
        case KEY_NUM1: x--; y--; break;
        case KEY_NUM2: y--; break;
        case KEY_NUM3: x++; y--; break;
        case KEY_NUM4: x--; break;
        case KEY_NUM6: x++; break;
        case KEY_NUM7: x--; y++; break;
        case KEY_NUM8: y++; break;
        case KEY_NUM9: x++; y++; break;
    }
    // инициировать перерисовку экрана
    repaint();
}
}

```

Обратим внимание на то, что в отличие от работы с формой, изменения изображения не сразу отражаются на экране. Перерисовка экрана, реализованная в методе `paint`, должна быть вызвана явно, с помощью одного из следующих методов класса `Canvas`:

- `void repaint(int x, int y, int width, int height)` — перерисовка прямоугольной области экрана шириной `width` и высотой `height`, левый верхний угол которой находится в точке с координатами `(x,y)`;
- `void repaint()` — перерисовка экрана целиком. Аналогична такому вызову предыдущего метода: `repaint(0, 0, getWidth(), getHeight())`.

Сам же метод `paint` вызывается внутренней реализацией, мы никогда не будем вызывать его явно.

Основная работа выполнена. Отображение управляемой точки на экране выполняется в стартовом методе мидлета точно так же, как и в предыдущем случае:

```

public void startApp() {
    // создать объект управляемой точки
    Point point = new Point();
    // получить ссылку на менеджер дисплея
    display = Display.getDisplay(this);
    // вывести управляемую точку на экран
    display.setCurrent(point);
}

```

Немного поиграв в то, что получилось, мне удалось нарисовать вот такую картинку (рис. 4.3).

С этого примера мы начнем разработку законченной, качественной игры, которой уже и не стыдно будет похвастаться даже перед профессионалами. Идея далеко не оригинальна: это знакомая нам с детства «Змейка», так любимая разработчиками корпорации Nokia. Самое время дать наш ответ заграничным Чемберленам и укротить непослушную змею. На этом примере мы и будем постигать тонкости программирования под мобильные телефоны, постепенно усложняя и совершенствуя нашу игру. Но это будет уже в следующих главах...

В этой главе мы рассмотрели работу с пользовательским интерфейсом низкого уровня, а также графические возможности языка J2ME и логику обработки событий. Поскольку класс `Canvas` не является потомком класса `Screen`, он не использует ни одной высокоуровневой абстракции, определяемой иерархией класса `Screen`, например добавление заголовка или бегущей строки невозможно. С другой стороны, класс `Canvas` принадлежит к иерархии отображаемых объектов типа `Displayable`, поэтому он может задействовать обработку команд высокого уровня и реализовать блок прослушивания команд.

Для использования интерфейса низкого уровня необходимо создать новый класс, расширяющий класс `Canvas`, и реализовать его абстрактный метод `paint`, который и будет осуществлять прорисовку экрана. В этой главе мы также рассмотрели большинство возможностей и методов классов `Graphics` и `Canvas`. Кое-что осталось на самостоятельное изучение, благо корпорация Sun Microsystems предоставляет подробную документацию языка.

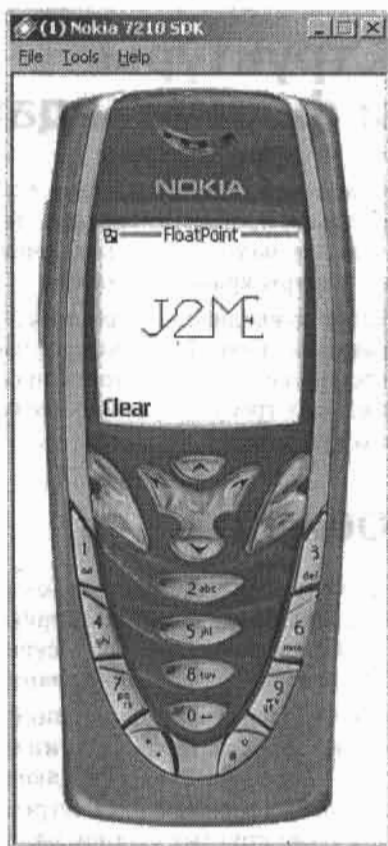


Рис. 4.3. Текст, написанный с помощью «управляемой точки»

Глава 5

Структуры и форматы данных в J2ME

Самое время перейти к теме, с которой, наверное, стоит начинать изучение любого языка программирования: типы и структуры данных. Какие-то типы мы уже использовали в примерах, теперь рассмотрим все существующие типы, а также структуры хранения данных.

Кому-то вводная информация о типах данных и контейнерах языка Java может показаться скучной. В таком случае лучше сразу перейти к водным процедурам — ко второй части главы, где мы продолжим разработки нашей игры в змейку. С теми же, кому требуется дополнительный ликбез в области общего программирования, мы продолжим.

Объекты

Язык Java является объектно-ориентированным языком даже в большей степени, чем остальные языки, оперирующие понятием «объект». Любые данные представляются в виде объектов — сущностей, имеющих набор свойств-полей и методов для работы с этими свойствами.

Создается новый объект с помощью операции `new`, где указывается тип создаваемого объекта. При создании объекта автоматически вызывается его конструктор — особый метод, отвечающий за инициализацию объекта. Конструктор может иметь входные параметры инициализации, которые передаются в операции `new` через круглые скобки, например:

```
String str = new String("some text");
```

Объект может иметь несколько конструкторов. Вызывается тот конструктор, параметры которого совпадут по количеству и типу с параметрами, переданными через `new`.

В двух словах коснемся наследования классов. На основе любого класса можно создать свой собственный класс, который будет содержать все поля и методы родительского класса, а также те свойства и методы, которые мы решим придать новому классу. Реализуется наследование с использованием ключевого слова `extends`, которое мы уже применяли в прошлой главе:

```
public class Graph extends Canvas
```

Для знатоков C++ заметим, что в языке Java множественное наследование запрещено вообще. Вместо этого предлагается сущность, именуемая интерфейсом.

Интерфейс отличается от обычного класса тем, что содержит только заголовки методов и не содержит их реализации. Любой создаваемый класс может реализовать один или несколько интерфейсов, как мы это делали в прошлой главе:

```
public class Point extends Canvas implements CommandListener
```

К сожалению, мы не имеем возможности глубоко вникать в идеологию объектно-ориентированного программирования, поэтому на данном этапе можно почитать дополнительную литературу по этому вопросу.

Примитивные типы

Несмотря на то, что Java является объектно-ориентированным языком, представляющим любые данные в виде объектов, все же существуют особые типы данных, называемые примитивными, которые очень широко используются в мобильном программировании.

Поддержка таких типов обусловлена тем, что зачастую малоэффективно создавать полноценный объект, используя оператор `new` и всю систему объектного захвата памяти, для одной маленькой циферки. Далее приведены примитивные типы, которые поддерживает J2ME, с объемом памяти, выделяемым каждому типу:

<code>byte</code>	1 байт
<code>short</code>	2 байта
<code>char</code>	2 байта
<code>int</code>	4 байта
<code>long</code>	8 байт

В прошлых главах мы рассматривали набор методов различных классов. Перед каждым методом указан тип данных, который возвращает тот или иной метод. Например, метод класса `Canvas` `getHeight()` возвращает высоту экрана в виде числа, представленного примитивным типом `int`. Часто методы выполняют какие-то действия, но не возвращают никаких данных или значений. В таком случае тип возвращаемых данных в заголовке метода должен быть определен как `void` (пустой тип данных). В этом случае оператор `return` предписывает лишь выход из метода и не возвращает никаких значений.

Справедливости ради заметим, что любой примитивный тип данных может быть представлен соответствующим объектом одного из классов: `Byte`, `Character`, `Short`, `Integer`, `Long`.

Массивы

Массивом называется набор однотипных данных, хранимых в смежных ячейках оперативной памяти. Работа с массивами в языке Java производится в три этапа.

1. Объявление массива. На этом этапе задается переменная и тип массива. Квадратные скобки указывают на то, что объявляется именно массив. Например:

```
int array[];  
Image images[];
```

2. Определение массива осуществляется с помощью оператора `new`, в котором требуется указать точный размер массива, называемый длиной. На этом этапе происходит захват памяти для размещения массива. Ссылка на захваченную память присваивается переменной массива:

```
array = new int[50];
```

```
images = new Image[20];
```

3. Инициализация массива — присвоение значений элементам массива. Каждый элемент имеет свой индекс — порядковый номер от нуля до длины массива без единицы. Обращение к элементам массива выглядит так:

```
array[0] = 5;
```

```
array[1] = 10;
```

Одна из основных проблем обычных массивов состоит в том, что при создании массива нужно точно указать количество хранимых в нем данных, что далеко не всегда удобно или вообще невозможно. Кроме обычных массивов существуют классы-контейнеры, позволяющие хранить произвольное количество данных. Такие контейнеры позволяют удобно решать задачи, требующие частого добавления и удаления неизвестного заранее количества данных. Все контейнеры содержатся в пакете `java.util`. Класс `Vector` содержится в языке Java с самых первых ее версий.

Класс Vector

Класс `Vector` предназначен для хранения произвольного числа аргументов самого общего типа `Object`, то есть хранимые данные должны быть тем или иным образом наследованы из класса `Object`. Под эту категорию попадают любые объекты языка J2ME, кроме примитивных типов данных.

Размер вектора (`size`) — это количество хранимых в нем элементов. Каждый элемент при добавлении получает свой порядковый индекс, начиная с нуля. Таким образом, индекс последнего элемента вектора на единицу меньше его размера. Метод `int size()` возвращает текущий размер вектора, а метод `boolean isEmpty()` возвращает значение `true`, если вектор не содержит элементов.

Метод `void setSize(int newSize)` устанавливает новый размер вектора, причем если значение `newSize` превышает текущий размер вектора, то все новые элементы принимают значение `null`. Если значение `newSize` меньше, чем текущий размер вектора, то все элементы с индексами `newSize` и выше удаляются.

Добавить новый элемент в вектор можно с помощью трех методов:

- `void addElement(Object obj)` — добавляет новый элемент в конец вектора. Добавленный элемент получает наивысший индекс, размер вектора увеличивается на единицу;
- `void insertElementAt(Object obj, int index)` — добавляет новый элемент с заданным индексом `index`. Индекс элемента, находившегося на этом месте, а также все следующие индексы увеличиваются на единицу. Аргумент `index` должен быть больше или равен нулю и меньше, или равен размеру вектора;

- `void setElementAt(Object obj, int index)` — заменяет элемент с индексом `index` на новый элемент `obj`, при этом старый элемент удаляется из вектора.

Удалить элементы из вектора можно также следующими методами:

- `void removeElementAt(int index)` — удаляет из вектора элемент с индексом `index`, при этом все индексы элементов, выше удаляемого, уменьшаются на единицу;
- `boolean removeElement(Object obj)` — удаляет из вектора первое вхождение элемента `obj` с логикой сдвига индексов, как и в предыдущем случае. Метод возвращает значение `true`, если удаляемый элемент действительно находился в векторе;
- `void removeAllElements()` — удаляет все элементы вектора; аналогично вызову `setSize(0)`.

Получить конкретный элемент вектора можно с помощью следующих методов:

- `Object firstElement()` — возвращает первый элемент вектора;
- `Object lastElement()` — возвращает последний элемент вектора;
- `Object elementAt(int index)` — возвращает элемент вектора с индексом `index`.

Следующие методы предоставляют возможности поиска элемента в векторе:

- `boolean contains(Object elem)` — возвращает значение `true`, если элемент `elem` содержится в векторе;
- `int indexOf(Object elem)` — возвращает индекс первого вхождения элемента `elem` в вектор; в случае отсутствия элемента в векторе возвращает `-1`;
- `int indexOf(Object elem, int index)` — возвращает индекс первого вхождения элемента `elem` в вектор, начиная поиск с элемента с индексом `index`;
- `int lastIndexOf(Object elem)` — возвращает индекс последнего вхождения элемента `elem` в вектор;
- `int lastIndexOf(Object elem, int index)` — возвращает индекс последнего вхождения элемента `elem` в вектор. Поиск осуществляется в обратном направлении, начиная с элемента с индексом `index`.

Класс Snake

Самое время вернуться к нашей змее. В прошлой главе мы рассматривали пример программы управляемой точки. Теперь расширим это приложение до управляемой змейки. Сейчас мы заложим основу и реализуем скелет нашей будущей игры.

Первым делом нарисуем собственно управляемую змейку, которая будет состоять из маленьких кусочков, каждый из которых будет представлен готовой картинкой. По сценарию змея должна не просто ползать по экрану, а охотиться на воображаемую добычу, поэтому также нам потребуется дополнительная логика для управления добычей и процессом «поедания». И самое главное: если змея укусит свой хвост или упрется в стену, то она неминуемо погибнет.

Основной класс, который будет реализовывать всю логику игры, будем наследовать от рассмотренного в прошлой главе класса `Canvas`, который отвечает нашим

требованиям рисования произвольной графики, а также организации управления на низком уровне:

```
private class Snake extends Canvas
```

Для реализации гибкого змеиногo тела нам потребуется пять различных элементов: голова, хвост, тело и два варианта сгибов тела, различающихся направлением поворота. Эти типы частей в нашем классе будут представлены следующими константами:

```
// константы, определяющие тип части тела змеи
private int HEAD = 0;           // голова змеи
private int TAIL = 1;           // хвост змеи
private int BODY = 2;           // тело змеи
private int ACLOCKWISE_TURN = 3; // сгиб тела против часовой стрелки
private int CLOCKWISE_TURN = 4; // сгиб тела по часовой стрелке
```

В каждый момент времени мы точно знаем, в какую сторону ползет змея, и в соответствии с этим делаем необходимые проверки и продвигаем змею в нужном направлении. Возможные направления представлены следующими константами:

```
// константы, определяющие направление движения
private int UP = 0;             // вверх
private int LEFT = 1;           // влево
private int RIGHT = 2;          // вправо
private int DOWN = 3;           // вниз
private int direction;          // текущее направление движения змеи
```

Текущее направление движения представлено полем `int direction`, которое принимает значение одной из констант, определяющих направление движения.

Поскольку мы взяли за красивую качественную игру, то и змея наша должна выглядеть достаточно реалистично, поэтому в зависимости от направления движения различные части змеи будут выглядеть по-разному. Заготовим для каждой возможной части отдельные картинки и разместим их в папке ресурсов приложения `/res`. Здесь можно проявить фантазию и художественное мастерство. Я нарисовал вот такие части (рис. 5.1).

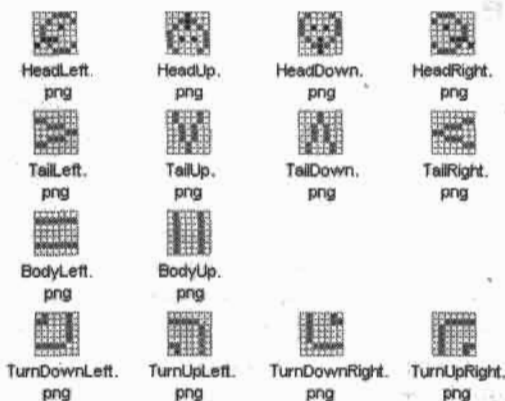


Рис. 5.1. Части змеи выглядят по-разному в зависимости от направления

Частей змей у нас фиксированное количество, не требующее динамического добавления или удаления, поэтому в данном случае достаточно обычного массива, который также будет являться членом класса Snake:

```
private Image[] images;
```

Размер картинки также будем хранить в отдельном поле:

```
private int imageSize;
```

Каждая часть тела змей будет представлена объектом класса SnakePart, который будет содержать тип части, координаты и направление движения. Если для управления точкой нам было достаточно хранить только текущие координаты, то теперь нам потребуется хранилище для всех частей. Класс Vector полностью отвечает нашим требованиям, поэтому класс SnakePart должен быть наследован от класса Object. Согласно правилам хорошего тона программирования работу с внутренними полями нового класса организуем через дополнительные методы. Код класса SnakePart выглядит следующим образом:

```
private class SnakePart extends Object {
    private int x:           // координата x
    private int y:           // координата y
    private int part:        // код части тела змеи
    private int dir:         // код направления движения
    // конструктор, принимающий все поля как аргументы
    public SnakePart(int _x, int _y, int _part, int _dir) {
        x = _x;
        y = _y;
        part = _part;
        dir = _dir;
    }
    // получить код части тела змеи
    private int getPart() { return part; }
    // получить код направления движения
    private int getDir() { return dir; }
    // получить координату x
    private int getX() { return x; }
    // получить координату y
    private int getY() { return y; }
    // установить тип части и направление
    private void setPartDir(int _part, int _dir) {
        // если второй аргумент 0, то
        // направление остается неизменным
        part = _part;
        if(_dir != 0) dir = _dir;
    }
}
```

Все элементы змей будет хранить поле

```
private Vector snake;
```

Также для организации графики нам потребуются следующие поля:

```
private int width:    // ширина экрана
private int height:   // высота экрана
private int xHead:    // координата x головы змейки
private int yHead:    // координата y головы змейки
```

После всех необходимых приготовлений и организации структуры класса можно переходить к конструированию нового объекта класса Snake:

```
// конструктор класса Snake
public Snake() {
    // конструктор родительского класса
    super();
    // создать массив для картинок-частей змейки
    images = new Image[20];
    // создать объекты картинок для каждой части
    try {
        // голова во всех направлениях
        images[0] = Image.createImage("/HeadUp.png");
        images[1] = Image.createImage("/HeadLeft.png");
        images[2] = Image.createImage("/HeadRight.png");
        images[3] = Image.createImage("/HeadDown.png");
        // хвост во всех направлениях
        images[4] = Image.createImage("/TailUp.png");
        images[5] = Image.createImage("/TailLeft.png");
        images[6] = Image.createImage("/TailRight.png");
        images[7] = Image.createImage("/TailDown.png");
        // тело
        images[8] = Image.createImage("/BodyUp.png");
        images[9] = Image.createImage("/BodyLeft.png");
        images[10] = images[9];
        images[11] = images[8];
        // сгиб тела против часовой стрелки
        images[12] = Image.createImage("/TurnDownLeft.png");
        images[13] = Image.createImage("/TurnUpLeft.png");
        images[14] = Image.createImage("/TurnDownRight.png");
        images[15] = Image.createImage("/TurnUpRight.png");
        // сгиб тела по часовой стрелке
        images[16] = images[14];
        images[17] = images[12];
        images[18] = images[15];
        images[19] = images[13];
    }
    catch (IOException ioe) {}
    // получить размер одного элемента змеи
    imageSize = images[0].getWidth();
    // создать вектор, хранящий элементы змейки
```

```

snake = new Vector();
// получить ширину экрана
width = getWidth();
// получить высоту экрана
height = getHeight();
// установить координаты головы в центре экрана
// для транспортабельности координаты кратны imageSize
xHead = ((width/2)/imageSize)*imageSize;
yHead = ((height/2)/imageSize)*imageSize;
// создать элемент головы змейки
SnakePart partS = new SnakePart(xHead,yHead,HEAD,UP);
// добавить элемент в вектор
snake.addElement(partS);
// создать элемент тела змейки
partS = new SnakePart(xHead,yHead+imageSize,BODY,UP);
snake.addElement(partS);
// создать элемент хвоста змейки
partS = new SnakePart(xHead,yHead+2*imageSize,TAIL,UP);
snake.addElement(partS);
// установить текущее направление движения
direction = UP;
// сбросить флаг окончания игры
gameOverFlag = false;
)

```

Как описывалось в прошлой главе, для любых классов, расширяющих класс Canvas, обязателен к реализации метод `paint()`, отвечающий за перерисовку экрана. Пройдем по всем элементам вектора и выведем на экран соответствующие части змейки согласно их координатам:

```

public void paint(Graphics g) {
    // очистить экран
    g.setColor(0xFFFFFF);
    g.fillRect(0,0,width,height);
    g.setColor(0x000000);
    // нарисовать рамку
    g.drawRect(1,1,width-3,height-3);
    // получить длину змейки
    int snakeLen = snake.size();
    // если поднят флаг конца игры
    if ( gameOverFlag ) {
        // вывести сообщение
        g.drawString("GAME OVER",width/2,height/2-10,g.HCENTER|g.TOP);
        // вывести счет, равный длине змейки
        String score = new String("YOUR SCORE: "+snakeLen);
        g.drawString(score,width/2,height/2+10,g.HCENTER|g.TOP);
        return;
    }
}

```

```

    } else {
        // для каждого элемента змейки
        for ( int i=0; i<snakeLen; i++){
            // вывести картинку, индекс которой вычисляется
            // с помощью комбинации кода части тела и кода направления
            g.drawImage(images[4*((SnakePart)snake.elementAt(i)).getPart()+
                ((SnakePart)snake.elementAt(i)).getDir()],
                ((SnakePart)snake.elementAt(i)).getX(),
                ((SnakePart)snake.elementAt(i)).getY(),
                g.HCENTER | g.VCENTER);
        }
    }
}

```

Теперь добавим управление нашей змейкой. Для этого реализуем метод `keyPressed(int keyCode)`. Перед тем как осуществлять передвижение, будем проверять, можно ли ползти в заданном направлении. Если на пути стена или собственный хвост, то поднимаем флаг конца игры.

```

public void keyPressed(int keyCode) {
    // если флаг конца игры не поднят
    if ( !gameOverFlag ){
        switch (keyCode) {
            // проверить, можно ли передвигаться в заданном направлении.
            // и вызвать соответствующую функцию передвижения
            case KEY_NUM2:
                checkMove(xHead, yHead-imageSize): moveUp(); break;
            case KEY_NUM4:
                checkMove(xHead- imageSize, yHead): moveLeft(): break;
            case KEY_NUM6:
                checkMove(xHead+ imageSize, yHead): moveRight(): break;
            case KEY_NUM8:
                checkMove(xHead, yHead+ imageSize): moveDown(): break;
        }
        // вызвать перерисовку экрана
        repaint();
    }
}

// проверить возможность передвижения
// в точку с координатами (xH,yH)
public void checkMove(int xH, int yH) {
    // для каждого элемента змеи
    for ( int i=3; i<snake.size()-1; i++ ) {
        // если проверяемая точка совпадает с координатами элемента
        if ( xH == ((SnakePart)(snake.elementAt(i))).getX() &&
            yH == ((SnakePart)(snake.elementAt(i))).getY() )
            // поднять флаг конца игры

```

```
gameOverFlag=true;
}
}
// передвижение влево
private void moveLeft() {
    // если текущее направление не направо
    if ( direction!=RIGHT) {
        // сдвинуть координату головы влево
        xHead-=imageSize;
        // проверить выход за левую границу
        if ( xHead < 4) { gameOverFlag=true; return; }
        // получить первый элемент змеи
        SnakePart head = (SnakePart)(snake.firstElement());
        // если текущее направление вверх
        if ( direction==UP )
            // заменить голову на сгиб против часовой стрелки
            head.setPartDir(ACLOCKWISE_TURN.LEFT);
        else {
            // если текущее направление вниз
            if ( direction==DOWN )
                // заменить голову на сгиб по часовой стрелке
                head.setPartDir(CLOCKWISE_TURN.LEFT);
            else // если текущее направление влево
                // заменить голову на тело
                head.setPartDir(BODY.LEFT);
        }
        // создать новый элемент для головы
        SnakePart sPart = new SnakePart(xHead,yHead,HEAD.LEFT);
        // добавить голову первым элементом вектора
        snake.insertElementAt(sPart,0);
        // продвинуть хвост
        // удалить последний элемент
        snake.removeElement(snake.lastElement());
        // назначить хвостом предпоследний элемент
        ((SnakePart)snake.lastElement()).setPartDir(TAIL,-1);
        // установить текущее направление
        direction = LEFT;
    }
}
```

Передвижение в остальных направлениях реализуйте самостоятельно в функциях `moveUp()`, `moveRight()`, `moveDown()` по образу и подобию приведенной выше функции `moveLeft()`.

Итак, основа игры написана. Приложение уже можно компилировать и запускать. На экране мы увидим вот такого милого червячка (рис. 5.2).

Управление змейкой уже реализовано, поэтому можно смело делать первые шаги, точнее, первые ползки. Покрутимся в разные стороны и убедимся, что все повороты реализованы без ошибок. При попытке уползти за пределы экрана мы должны получить сообщение о завершении игры (рис. 5.3).

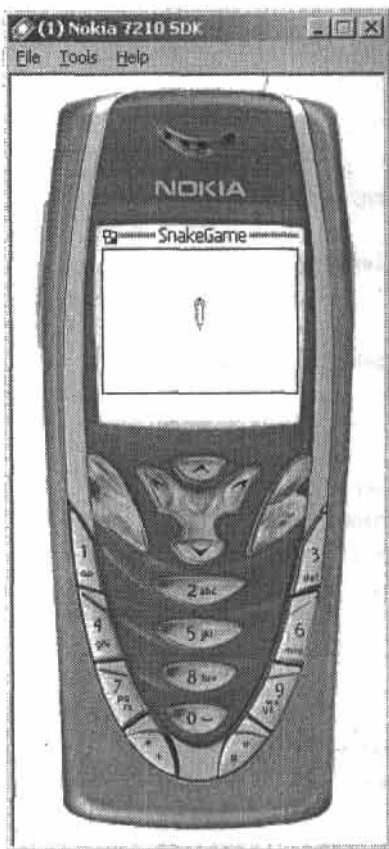


Рис. 5.2. Змейка в начале игры



Рис. 5.3. «Game over!»

Итак, все работает отменно, кроме главного момента — змея не растет. Как мы помним из аналогов, для роста змее требуется вкусная и здоровая пища, которая появляется на экране случайным образом. Пища в нашей игре будет представлена сердечком, картинку для которого сохраним в файле `heart.png`, находящемся в папке `/res` вместе с остальными картинками элементов змейки. Для реализации правильного питания в класс `Snake` добавим несколько полей:

```
private Image heart;      // картинка сердца
private int xHeart;       // координата x сердца
private int yHeart;       // координата y сердца
private boolean eatFlag;  // флаг удлинения змеи
```

Также нам понадобится метод вычисления координат для случайного размещения сердечка на экране. Здесь нам потребуется генератор случайных чисел,

представленный в языке J2ME классом `Random`. Очередное случайное число генерируется с помощью метода `nextInt()`. Заметим, что число может быть отрицательным, поэтому требуется получить его абсолютное значение, используя метод класса `Math` `abs(int a)`.

```
// получить новые координаты для сердца
public void setNewHeart() {
    // получить количество возможных позиций по координате x
    int xPartMaxNum = (width-5)/imageSize;
    // получить количество возможных позиций по координате y
    int yPartMaxNum = (height-5)/imageSize;
    // получить длину змеи
    int len = snake.size();
    // создать объект генератора случайных чисел
    Random rnd = new Random();
    // флаг корректной генерации
    boolean setFlag = false;
    // повторяем, пока не сгенерируем корректные координаты сердца
    while ( !setFlag ) {
        // поднять флаг корректной генерации
        setFlag = true;
        // сгенерировать случайную координату x, кратную imageSize
        xHeart = imageSize*(Math.abs(rnd.nextInt())%xPartMaxNum+1);
        // сгенерировать случайную координату y, кратную imageSize
        yHeart = imageSize*(Math.abs(rnd.nextInt())%yPartMaxNum+1);
        // проверить совпадение координат сердца
        // с координатами элементов змеи
        for ( int i=0; i<len-1; i++ )
            if ( xHeart == ((SnakePart)(snake.elementAt(i))).getX() &&
                yHeart == ((SnakePart)(snake.elementAt(i))).getY() )
                // опустить флаг корректной генерации
                setFlag = false;
    }
}
```

В конструкторе класса `Snake` добавим создание объекта картинки для сердечка `heart = Image.createImage("/heart.png")` и в самом конце вызовем метод вычисления координат для этой картинки `setNewHeart()`. В методе `paint()` вызовем прорисовку картинки: `g.drawImage(heart, xHeart, yHeart, g.HCENTER | g.VCENTER)`.

Осталось реализовать логику удлинения змеи при встрече с сердечком. В методе `checkMove()` при попытке передвижения проверяем, совпадают ли координаты предполагаемого передвижения с координатами сердечка, и в случае равенства поднимаем флаг удлинения змеи:

```
if ( xH == xHeart && yH == yHeart )
    eatFlag = true;
else
    eatFlag = false;
```

В методах передвижения `moveUp()`, `moveLeft()`, `moveRight()`, `moveDown()`, перед тем как продвинуть хвост, удаляя последний элемент вектора, нужно проверить, не поднят ли флаг удлинения змеи. В случае удлинения вызовем метод генерации новых координат для сердечка `setNewHeart()`, а продвижение хвоста производить не будем:

```
// если флаг удлинения змеи поднят
if (eatFlag)
    // получить новые координаты для сердца
    setNewHeart();
else {
    // продвинуть хвост
    // удалить последний элемент
    snake.removeElement(snake.lastElement());
    // назначить хвостом предпоследний элемент
    ((SnakePart)snake.lastElement()).setPartDir(TAIL, -1);
}
```

Вот и все, самое страшное уже позади. Теперь подключаем все необходимые пакеты:

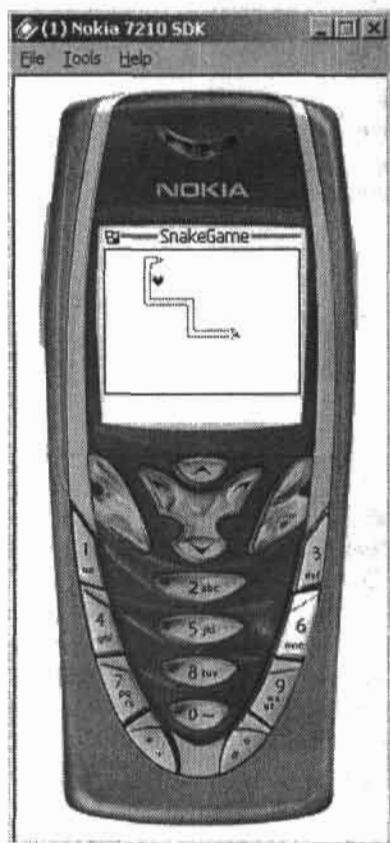


Рис. 5.4. Змейка в действии

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Display;
import java.util.Vector;
import java.util.Random;
import java.io.IOException;
```

компилируем получившееся приложение и исправляем ошибки, без которых не обойтись, особенно на первых этапах освоения нового языка. Запускаем приложение и посвящаем час-другой превращению червяка в анаконду (рис. 5.4).

Заметим, что мы написали только основу для игры, ведь змейка пока что не ползает самостоятельно, так что играть в такую игру будет интересно только детям дошкольного возраста, и то лишь первые десять минут. Также хотелось бы расширить игровое поле на весь экран, реализовать систему хранения таблицы рекордов и многое другое, чем мы и займемся в следующих главах.

Класс Stack

В заключении главы рассмотрим еще один класс-контейнер, реализующий структуру данных *стек*.

Данная структура характерна особым подходом к добавлению и удалению элементов. Принцип хранения данных LIFO (Last In, First Out) означает, что извлечь из структуры можно только тот элемент, который был добавлен последним. Для ясности проведем аналогию с узким стаканом, в который можно складывать различные предметы, но достать можно лишь тот предмет, который находится выше всех.

В языке J2ME класс Stack реализован на основе класса Vector, расширяя его функциональность следующими методами:

- `Object push(Object item)` — добавляет элемент на вершину стека;
- `Object pop()` — извлекает элемент с вершины стека;
- `Object peek()` — возвращает элемент с вершины стека без его извлечения;
- `boolean empty()` — возвращает значение true, если стек пуст.
- `int search(Object o)` — возвращает позицию искомого объекта относительно вершины стека. Если объект содержится в стеке несколько раз, то возвращается позиция самого верхнего; если элемент не содержится в стеке, то возвращается значение -1.

* * *

Итак, мы рассмотрели в теории и применили на практике основные типы и структуры данных языка J2ME. Корректное обращение с данными и преобразование типов данных предотвратит массу возможных ошибок, поэтому к этому моменту стоит отнестись особо внимательно. Перед разработкой любого приложения следует хорошо обдумать, каким образом будут храниться данные приложения, какова область применения тех или иных данных, и в зависимости от этого уже проектировать структуру классов и самого мидлета.

Глава 6

Организация многопоточных приложений

В прошлой главе мы реализовали основную часть игры «Змейка», придерживаясь классических принципов игры. Используя стандартное управление, можно ползать по всему экрану и собирать сердечки, удлиняя змею. При столкновении змеи со стеной или с собственным телом игра заканчивается. Пока что главное отличие от классической игры заключается в том, что змейка стоит на месте, если не нажимать управляющие клавиши. Таким образом, эта увлекательная и развивающая игра теряет всякий смысл.

Основная проблема в реализации самостоятельного продвижения змеи такова, что если программно зациклить какое-то действие, то приложение не будет иметь возможности выйти из цикла и обработать нажатие функциональных клавиш. На помощь в решении этой проблемы приходит многозадачность мобильной операционной системы.

Многозадачность

В современных операционных системах, в том числе и мобильных, реализовано понятие многозадачности, которое заключается в одновременном выполнении нескольких программ. Слово «одновременно» здесь надо понимать так: каждой из программ поочередно отводится определенное количество процессорного времени, в течение которого программа выполняется, а затем снимается с процессора, уступая свое место другой программе.

Сама программа, в нашем случае — мидлет, также может создавать несколько параллельных процессов, которые будут выполняться одновременно. Такие процессы в оригинале языка Java называются *thread*. Понятие *thread* в русской литературе переводят по-разному. В буквальном переводе термин обозначает «нить», но мы все же не швей-мотористки, а какие-никакие программисты. Часто *thread* переводят как «поток», но, к сожалению, этот термин уже занят потоками ввода-вывода (*streams*), на которых мы остановимся позже. Так что в данной главе мы будем пользоваться транслитерацией термина — «тред».

В любом мидлете существует, по крайней мере, хотя бы один тред — главный тред выполнения мидлета, из которого можно создать другие треды для одновременного выполнения. В языке Java треды представлены объектами класса *Thread*.

Класс Thread

Класс Thread реализует интерфейс Runnable, который содержит объявление лишь одного метода: `void run()`. В методе `run()` должны быть реализованы все действия, предусмотренные для выполнения в конкретном треде. Сам класс Thread содержит пустую реализацию метода `run()`, поэтому задать действия для создаваемого треда можно следующими двумя способами:

- расширить класс Thread и переопределить метод `run()`;
- реализовать интерфейс Runnable.

Мы будем пользоваться вторым способом, поскольку в наших примерах класс, создающий тред, уже наследован от какого-то другого класса, а множественное наследование в языке Java не поддерживается. Для каждого способа класс Thread содержит отдельный конструктор:

- `Thread()` — конструктор, не принимающий аргументов; вызывается из конструктора класса-потомка, расширяющего класс Thread;
- `Thread(Runnable target)` — конструктор, принимающий в качестве аргумента объект, реализующий интерфейс Runnable;

После того как объект треда создан, предписанные в методе `run()` действия еще не начинают выполняться. Для управления тредом существует несколько методов класса Thread:

- `void start()` — начать выполнение треда. Виртуальная машина вызывает метод `run()`, который никогда не вызывается программистом явно. После вызова метода `start()` начинается параллельное выполнение программы в двух направлениях: тред идет своей дорогой, а поезд идет своей;
- `void sleep(long millis)` — приостанавливает выполнение треда на `millis` миллисекунд.

В этом месте мы остановимся и попробуем применить изученную теорию на практике. Вернемся к программе фотоальбома, который мы реализовали во второй главе. Усовершенствуем эту программу таким образом: добавим показ слайд-шоу, то есть автоматическую смену картинок на экране.

Организуем управление автоматической сменой картинок в отдельном треде. Для этого в основном классе нашего фотоальбома `SlideShow` реализуем интерфейс Runnable. Теперь декларация класса будет выглядеть следующим образом:

```
public class SlideShow extends MIDlet implements CommandListener, Runnable
```

Следует также добавить в класс `SlideShow` новое поле для объекта треда:

```
private Thread thread;
```

Как уже было сказано ранее, все действия, выполняемые тредом, следует реализовать в методе `run()`:

```
public void run() {
    // вечный цикл
    while (true) {
        try {
            // удалить из формы текущую картинку
```



```

        form.delete(0);
        // добавить в форму новую картинку
        // увеличиваем текущий номер картинки и берем
        // остаток от деления на общее число картинок
        setImage("/" + (slideNum++) % maxSlideNum + 1 + ".png");
        // остановить выполнение треда на 3 секунды
        thread.sleep(3000);
        // обработать исключительную ситуацию
    } catch (InterruptedException e) {
    }
}
}

```

В методе `startApp()` осталось создать объект треда и дать команду для начала его выполнения:

```

// создать объект треда
thread = new Thread(this);
// начать выполнение треда
thread.start();

```

Заметим, что после запуска слайд-шоу остановить смену картинок будет очень проблематично, поэтому стоит все же воздержаться от бесконечных циклов и завести флаг, сбрасывая который, можно организовать завершение работы треда.

Приоритеты тредов

Как уже было отмечено, одновременное выполнение нескольких тредов достаточно условно, поскольку физически в каждый момент времени выполняется только один тред. Переключение тредов и количество выделенного процессорного времени зависит от операционной системы, однако программист тоже может влиять на этот процесс. Иногда некоторым тредам требуется выделить большее или меньшее количество процессорного времени. Технология мультитрединга позволяет определить приоритеты выполнения для каждого треда.

Приоритеты тредов представлены целыми числами в диапазоне от 1 до 10. Чем выше значение приоритета, тем больше времени выделяется треду операционной системой. Также в классе `Thread` определены константы `MIN_PRIORITY`, `NORM_PRIORITY` и `MAX_PRIORITY`, представляющие значения минимального, нормального и максимального приоритетов соответственно.

Работа с приоритетами осуществляется с помощью следующих методов класса `Thread`:

- `void setPriority(int newPriority)` — установить новый приоритет треду. При попытке выставить приоритет, выходящий за допустимый диапазон, будет сформировано исключение `IllegalArgumentException`;
- `int getPriority()` — возвращает текущий приоритет треда.

Продemonстрируем на небольшом примере многозадачность системы. Запустим параллельно два треда, каждый из которых в цикле будет выводить сообщения

в системную область (рис. 6.1). Главный тред будет выполнять основной код мидлета в методе `startApp()`, а для второго тред создадим отдельный класс `SecondThread`:

```
public void startApp() {  
    // создать второй тред  
    SecondThread thread = new SecondThread();  
    // запустить второй тред  
    thread.start();  
    // в цикле выводить сообщения в системную область  
    for(int i=1; i<=5; i++)  
        System.out.println("Main Thread: "+i);  
}  
  
private class SecondThread extends Thread {  
    // в цикле выводить сообщения в системную область  
    public void run() {  
        for(int i=1; i<=5; i++)  
            System.out.println("Second Thread: "+i);  
    }  
}
```

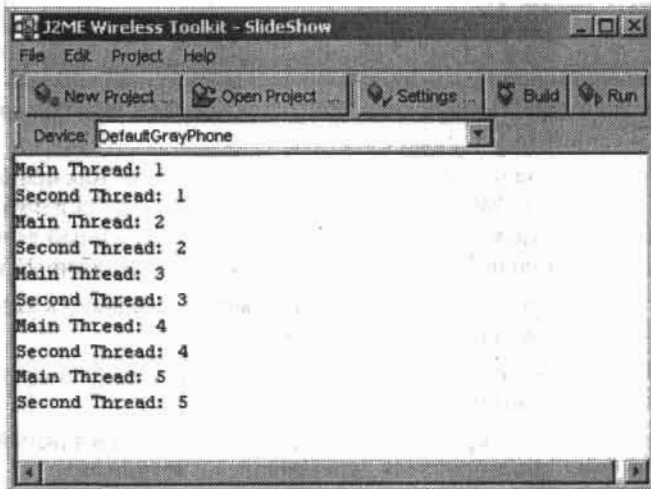


Рис. 6.1. Результаты параллельной работы двух тредов с одинаковым приоритетом

На рисунке мы видим, что переключение тредов работает, как швейцарские часы, и сообщения от каждого тред выдаются по очереди, несмотря на то, что вывод сообщений в программе представлен единым непрерывным циклом.

Попробуем теперь перед запуском второго тред выставить ему повышенный приоритет `thread.setPriority(Thread.MAX_PRIORITY)` и посмотрим, что же из этого получится (рис. 6.2).

Видим, что картина существенно поменялась: главный тред не начал выполняться, пока тред с повышенным приоритетом не закончил работу. Не исключена си-

туация, что при наличии множества тредов с высоким приоритетом какой-то низкоприоритетный тред вообще не получит управления. Такую ситуацию можно предотвратить с помощью следующих методов класса Thread:

- `void yield()` — принудительное переключение системы на выполнение следующего треда;
- `void join()` — ожидание завершения работы треда; останавливает работу программы, из которой был вызван данный тред.

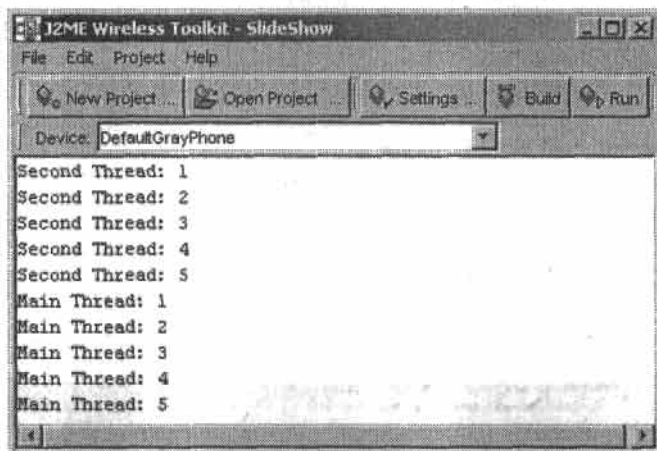


Рис. 6.2. Результаты параллельной работы двух тредов с разным приоритетом

При создании тред получает приоритет создавшего его треда. Как правило, все треды работают с нормальным (`NORM_PRIORITY`) приоритетом. Треды, ожидающие какого-то события, например нажатия кнопки, могут получить повышенный приоритет, а треды, выполняющие длительную, ресурсоемкую работу — пониженный приоритет.

Отследить состояние системы с точки зрения выполняющихся в конкретный момент тредов можно с помощью следующих методов:

- `boolean isAlive()` — возвращает значение `true`, если тред был запущен и до сих пор не прекратил выполнение;
- `Thread currentThread()` — возвращает тред, выполняемый в данный момент времени;
- `int activeCount()` — возвращает количество тредов, выполняемых системой в данный момент времени.

Синхронизация тредов

Основная сложность при работе с тредом заключается в том, что разные треды, порожденные одним мидлетом, имеют общее адресное пространство, то есть одни и те же данные могут быть доступны из разных тредов. В чем здесь криминал? Рассмотрим, например, ситуацию, когда два тред работают с одним и тем же вектором. Первый тред получает длину вектора, затем происходит смена выполняемого треда, и второй тред удаляет элементы из этого же вектора. Первый тред возобновляет ра-

боту и с удивлением обнаруживает, что полученная длина вектора не соответствует действительности, формирует исключение неверного индекса и падает с таким вот грохотом (рис. 6.3).

Это далеко не единственный пример. Контекстное переключение тредов может произойти в любой момент. В самой простейшей операции между вычислением и присваиванием результата может вклиниться другой тред и поменять данные. Результаты такой путаницы могут быть совершенно непредсказуемы.

Описанная проблема разрешается в языке Java при помощи технологии синхронизации тредов. Идея заключается в блокировке тредом доступа к объекту на время работы с общими данными. Перед тем как начать работать с данными, тред ставит блок на доступ к объекту, и все остальные треды при необходимости работы с этими данными ожидают снятия блока.

Блокировка осуществляется с помощью оператора `synchronized(Object obj) {...}`. В фигурных скобках размещается критическая секция программы — код приложения, требующий синхронизации. В качестве аргумента `obj` оператору передается объект, который будет заблокирован. Если в момент выполнения оператора `synchronized` синхронизируемый объект уже заблокирован каким-то другим тредом, то выполнение приостановится до тех пор, пока блок не будет снят.

Если какой-то метод требует синхронизации всех своих операторов, то весь метод может быть помечен как `synchronized`, например:

```
public synchronized void someActionWithData() {...}
```

В таком случае блокируется объект `this`, вызвавший данный метод. Блок снимается только после завершения работы метода.

Очень неприятная ситуация может возникнуть при перекрестной блокировке, когда один тред получил управление синхронизированным объектом А, а второй — синхронизированным объектом В. После этого первый тред пытается вызвать синхронизированный метод объекта В, а второй — аналогичный метод объекта А. Оба треды застрянут и будут ждать освобождения объектов, заблокированных друг другом, а программа с такой структурой тредов просто зависнет. В теории такая ситуация называется взаимоблокировкой (deadlock). Так что при разработке приложений следует очень внимательно относиться к вопросам корректной реализации синхронизации.

В языке Java существует способ предотвращения взаимоблокировки. Внутри синхронизированной критической секции можно приостановить работу тред с последующим ее возобновлением. Класс `Object` содержит несколько методов `wait`, которые откладывают выполнение тред. Отличие этих методов от метода `sleep`



Рис. 6.3. Демонстрация ошибки при синхронизации

класса `Thread` заключается в том, что во время приостановки работы треда блокировка с синхронизируемого объекта снимается. Методы класса `Object`, отвечающие за остановку и возобновление тредов:

- `void wait()` — текущий тред снимает блокировку с объекта и приостанавливается до тех пор, пока из какого-либо другого треда не будут вызваны методы `notify()` или `notifyAll()`;
- `void wait(long timeout)` — текущий тред снимает блокировку и откладывает выполнение, пока из какого-либо другого треда не будут вызваны методы `notify()` или `notifyAll()` или не истечет тайм-аут длиной `timeout` миллисекунд;
- `void wait(long timeout, int nanos)` — аналогичен предыдущему методу, с точностью тайм-аута до наносекунд, которая задается аргументом `nanos`. Вызов этих трех методов допускается только из синхронизированных методов или критических секций программы;
- `void notify()` — возобновляет работу одного приостановленного треда, произвольно выбранного системой;
- `void notifyAll()` — возобновляет работу всех приостановленных на данном объекте тредов. Фактически начинает работу только один тред, первым восстановивший блокировку объекта, остальные ждут снятия блока в обычном режиме.

Следует заметить, что при обращении к общим данным синхронизация нужна только в том случае, если один из тредов изменяет данные. В случае, когда все треды просто читают данные, не изменяя их, синхронизация тредов не требуется.

Класс Snake

Теперь у нас достаточно информации, чтобы добавить в игру самостоятельное продвижение змейки. Для этого необходимо реализовать в классе `Snake` интерфейс `Runnable`, а в методе `run()` организовать цикл продвижения змеи в текущем направлении. Декларация класса `Snake` будет выглядеть теперь так:

```
private class Snake extends Canvas implements Runnable
```

В методе `run()` будем проверять текущее направление движения змейки и в зависимости от этого вызывать соответствующий метод продвижения. Обратим внимание, что метод `run()` и метод `keyPressed(int keyCode)` будут изменять данные одного и того же объекта, поэтому здесь потребуется синхронизация. В объявлении метода `keyPressed` добавим ключевое слово `synchronized`, а в методе `run()` выделим критическую секцию кода:

```
public void run() {
    // пока не поднят флаг конца игры
    while (!gameOverFlag) {
        // синхронизировать с управлением
        synchronized (this) {
            // продвинуть змейку в текущем направлении
            if(direction==UP) {checkMove(xHead, yHead-7); moveUp();}
            if(direction==LEFT) {checkMove(xHead-7, yHead); moveLeft();}
            if(direction==RIGHT) {checkMove(xHead+7, yHead); moveRight();}
            if(direction==DOWN) {checkMove(xHead, yHead+7); moveDown();}
```

```

    }
    // вызвать перерисовку экрана
    repaint();
    try {
        // приостановить тред на speed миллисекунд
        Thread.sleep(level*speed);
    } catch (InterruptedException e) {
    }
}
}

```

Заметим, что в класс Snake мы ввели два новых поля, `private byte level` и `private int speed`, которые будут отвечать за скорость передвижения нашей змейки. Поле `level` содержит уровень сложности игры, а поле `speed` — коэффициент ускорения для вычисления времени, на которое «засыпает» тред перед очередным продвижением змейки. Чем меньше этот параметр, тем быстрее будет двигаться наша змейка. В конструкторе установим `speed = 100`, `level = 5`, а в методе `checkMove` после каждого «съеденного» сердечка будем увеличивать скорость передвижения змейки:

```

// проверить совпадение координат передвижения
// с координатами сердца
if ( xH == xHeart && yH == yHeart ) {
    eatFlag = true;
    // увеличить скорость
    speed--;
} else
    eatFlag = false;

```

Все, что осталось сделать, это создать в методе `startApp()` новый тред и запустить его:

```

// создать объект треда автоматического передвижения
Thread moveThread = new Thread(curSnake);
// начать выполнение треда
moveThread.start();

```

Компилируем, запускаем и наслаждаемся процессом! Играть стало интересней, играть стало веселее, а корпорация Nokia кусает локти от зависти. На достигнутом мы, конечно же, останавливаться не будем и в следующих главах еще усовершенствуем нашу игру. Добавим картинок-заставок, таблицу рекордов, возможность выбора уровня игры и прочие прелести. Оставайтесь с нами, не переключайте канал, дальше будет еще интереснее!

* * *

В этой главе мы опробовали многозадачность виртуальной машины Java на нехитрых примерах. Иногда без создания нового треда просто не обойтись, как это было в случае со змейкой. При создании большой полноценной программы следует задуматься, какие из частей программы могут выполняться одновременно. Зачастую оправдано создание отдельных тредов для каждой подзадачи. После создания тредов следует подумать над распределением приоритетов. Особое внимание следует уделить синхронизации тредов, если они обращаются к одним и тем же данным программы.

Глава 7

Сохранение данных и параметров приложения

Как показывает практика, люди делятся на две категории: на тех, кто прошел Doom от начала до конца, и на тех, у кого не хватило терпения, реакции или секретных кодов, чтобы выпустить кишки всем злобным монстрам, которыми щедро напичкован каждый новый уровень. Поколение игры Doom хорошо знает основной принцип 3D-стрелялок: «Главное — не забыть сохраниться!» В этой главе мы рассмотрим, как организовать долговременное хранение данных в мобильном телефоне. Хранение данных организовано в J2ME с помощью системы управления записями (Record Management System), реализованной в пакете `javax.microedition.rms`. Данные представлены записями (массив байтов типа `byte[]`), которые помещаются в хранилище записей (Record Store) и не стираются при закрытии приложения, перезагрузке телефона или замене батареи. Размер и физическое местоположение хранилища в памяти телефона зависят от конкретной модели аппарата. Хранилище жестко привязано к мидлету, и когда вы удаляете приложение из своего телефона, то все данные, относящиеся к нему, удаляются вместе с ним. По сути, хранилище записей — это база данных, которая может быть представлена следующей схемой (рис. 7.1).

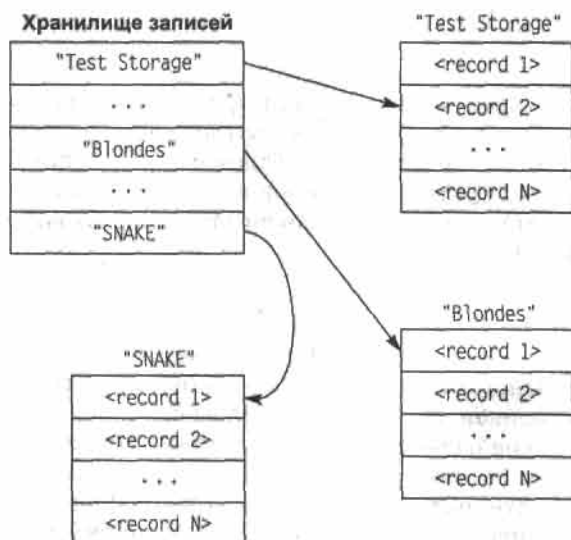


Рис. 7.1. Схема хранилища записей

Применения для RMS могут быть разные, от сохранения параметров и пользовательских настроек приложения до поддержки полноценной адресной книги с исчерпывающей личной информацией и гибким поиском. В игре, которую мы пишем, мы будем использовать систему управления записями для хранения уровня сложности игры, а также таблицы рекордов.

RecordStore — хранилище записей

Итак, хранилище записей представлено объектом класса `RecordStore`, работа с которым начинается с метода открытия хранилища для дальнейшей работы `static RecordStore openRecordStore(String recordStoreName, boolean createIfNecessary)`. Первый аргумент метода — уникальное имя хранилища. Второй аргумент указывает, должно ли быть создано новое хранилище с таким именем, если оно еще не существует. Имя хранилища чувствительно к регистру, длина имени не должна превышать 32 символа.

Добавление записи в хранилище осуществляется с помощью метода `int addRecord(byte[] data, int offset, int numBytes)`. Сохраняемые данные должны быть представлены последовательностью байтов, содержащихся в массиве `data`. Переменная `offset` указывает, с какой позиции в массиве начинаются данные, предназначенные для записи, а `numBytes` определяет размер сохраняемых данных. Метод возвращает номер новой записи, присвоенный ей при сохранении (`recordID`).

Рассмотрим вкратце еще некоторые методы хранилища записей `RecordStore`:

- `void deleteRecord(int recordId)` — удалить из хранилища запись с номером `recordId`;
- `static void deleteRecordStore(String recordStoreName)` — удалить хранилище записей с именем `recordStoreName`;
- `long getLastModified()` — возвращает время последнего изменения данных в хранилище записей в виде количества миллисекунд, прошедших с 1 января 1970 года до момента последнего изменения (Именно так!.. А кому сейчас легко?);
- `String getName()` — получить имя хранилища данных;
- `int getNextRecordID()` — возвращает номер, который будет присвоен следующей добавленной записи;
- `int getNumRecords()` — возвращает количество записей, содержащихся на данный момент в хранилище;
- `int getRecordSize(int recordId)` — возвращает размер (в байтах) конкретной записи хранилища с номером `recordId`;
- `int getSize()` — возвращает объем памяти (в байтах), который занимают все записи хранилища;
- `int getSizeAvailable()` — возвращает объем памяти (в байтах), доступный для расширения данного хранилища;
- `int getVersion()` — получить номер версии хранилища. При каждой операции с хранилищем, добавлении, удалении или изменении любой его записи, поднимается номер версии хранилища;

- `static String[] listRecordStores()` — возвращает массив имен всех хранилищ мидлета;
- `void setRecord(int recordId, byte[] newData, int offset, int numBytes)` — изменить данные, содержащиеся в записи с номером `recordId`; остальные параметры идентичны параметрам метода `addRecord`.

Доступ к данным хранилища предоставляет метод `byte[] getRecord(int recordID)`, где `recordID` — уникальный номер необходимой записи. То есть, чтобы получить запись, нужно точно знать ее номер. На этом этапе возникает небольшая проблема: как запоминать и где хранить эти номера. Вариант сохранения номеров записей в самом хранилище отпадает — по той же самой причине. На выручку приходит нумератор — интерфейс `RecordEnumeration`, осуществляющий перечисление всех записей в хранилище.

RecordEnumeration — нумератор списка записей

Используя метод `enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean keepUpdated)`, можно получить список записей конкретного хранилища. Навигация по списку номеров осуществляется с помощью методов нумератора `nextRecordId()` и `previousRecordId()`, которые возвращают по порядку все номера записей хранилища, или с помощью методов `nextRecord()` и `previousRecord()`, которые возвращают непосредственно записи в формате `byte[]`.

Для наглядности приведем небольшой пример кода программы, которая записывает в хранилище пронумерованные строки, а затем считывает их:

```
import javax.microedition.rms.RecordStore;
import javax.microedition.rms.RecordEnumeration;
import javax.microedition.rms.RecordStoreException;
try {
    String str;
    // открыть хранилище записей с именем "Test Storage"
    RecordStore recordStore = RecordStore.openRecordStore(
        "Test Storage", true);
    // получить нумератор списка записей
    RecordEnumeration re = recordStore.enumerateRecords(null, null, false);
    // добавить в хранилище 5 записей, содержащих пронумерованные строки
    for(int i=1;i<=5;i++) {
        str = "String no " + i;
        int id = recordStore.addRecord(str.getBytes(),0,str.length());
        System.out.println(str+" was stored with recordID = "+id);
    }
    // перестроить список
    re.rebuild();

    // получить все записи хранилища по их номерам
    while (re.hasNextElement()) {
```

```
int id = re.nextRecordId();
str = new String(recordStore.getRecord(id));
System.out.println("RecordID = "+id+": "+str);
}

// вернуть нумератор в исходное положение
re.reset();

// получить все записи хранилища
for (int i=1; i<=re.numRecords(); i++) {
    str = new String(re.previousRecord());
    System.out.println("Record: "+str);
}

// освободить ресурсы нумератора
re.destroy();

// закрыть хранилище записей
recordStore.closeRecordStore();

} catch (RecordStoreException rse) {
    System.out.println(rse.getMessage());
}
}
```

Результат работы примера выглядит так (рис. 7.2).

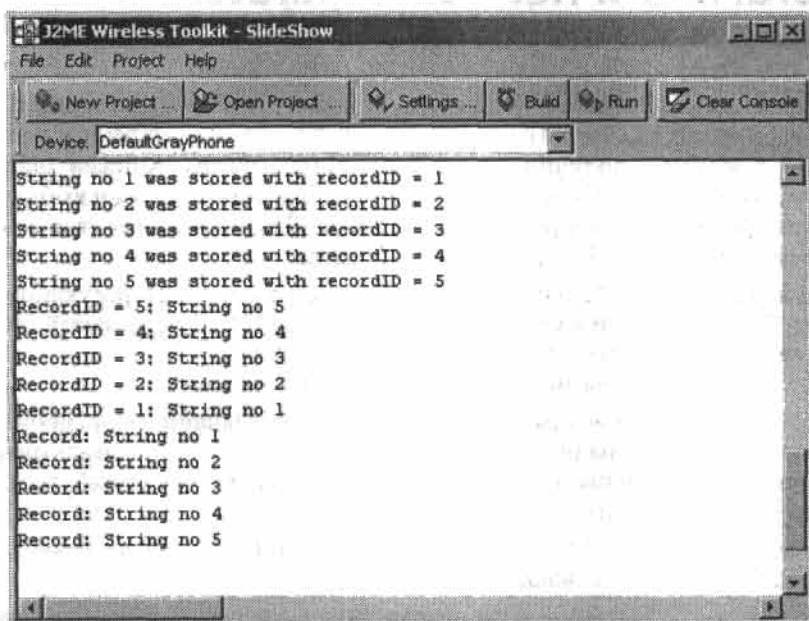


Рис. 7.2. Пример операций с записями хранилища

Рассмотрим методы нумератора, использованные в примере:

- `byte[] nextRecord()` — возвращает копию следующей записи в списке. Любые действия с полученным массивом не повлияют на запись, содержащуюся в хранилище. После вызова функции происходит продвижение по списку, то есть следующий вызов вернет уже другую запись. Первый вызов функции возвращает первую в списке запись;
- `byte[] previousRecord()` — возвращает копию предыдущей записи в списке. Первый вызов функции возвращает последнюю в списке запись;
- `int nextRecordId()` — возвращает номер следующей записи в списке по тому же принципу, что и метод `nextRecord`;
- `int previousRecordId()` — возвращает номер предыдущей записи в списке;
- `boolean hasNextElement()` — возвращает значение `true`, если список содержит следующую запись, то есть возможно продвижение с использованием `nextRecord`;
- `boolean hasPreviousElement()` — возвращает значение `true`, если список содержит предыдущую запись, то есть возможно продвижение с использованием `previousRecord`;
- `int numRecords()` — получить количество записей, содержащихся в списке;
- `void reset()` — возвращает нумератор в исходное положение;
- `void rebuild()` — перестроить список после того, как записи были удалены или добавлены в хранилище;
- `void destroy()` — освобождает все ресурсы, задействованные нумератором.

RecordFilter и RecordComparator — фильтр и компаратор

Вернемся к вышеприведенному примеру. При получении списка записей в методе `enumerateRecords` в качестве параметров `RecordFilter` и `RecordComparator` мы передали `null`, тем самым получив список всех без исключения записей хранилища. На самом деле, можно отделить зерна на этапе получения списка и выбрать только необходимые записи в определенном порядке. Для достижения этих целей служат два интерфейса RMS — фильтр и компаратор.

Начнем с фильтра. Чтобы выбрать из хранилища только интересующие нас записи, необходимо реализовать интерфейс `RecordFilter` и переписать его метод `boolean matches(byte[] candidate)`, который должен проверить запись на соответствие установленному нами критерию и вернуть `true` в положительном случае.

Компаратор — интерфейс сравнения записей, позволяющий отсортировать список в заданном нами порядке. Для этих целей необходимо реализовать интерфейс `RecordComparator` и переписать его метод `int compare(byte[] rec1, byte[] rec2)`, который возвращает константы:

- `RecordComparator.PRECEDES` — если первая запись (`rec1`) предшествует второй записи (`rec2`) согласно нашей логике;
- `RecordComparator.FOLLOWS` — если первая запись следует за второй согласно установленному нами критерию;

■ RecordComparator.EQUIVALENT — если с точки зрения упорядочивания записи равны. Объекты классов фильтра и компаратора передаются в качестве аргументов в метод хранилища записей `enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean keepUpdated)`. Обратим внимание на третий аргумент. Разработчики утверждают, что если передать значение `true`, то список будет автоматически перестраиваться во время работы с хранилищем после добавления или удаления записей. Предупреждение гласит, что использование этой опции может существенно повлиять на производительность приложения. В приведенном примере для целей синхронизации списка и хранилища был использован метод `rebuild()`. Эксперименты с параметром можете провести сами.

Для примера использования фильтра и компаратора допустим, что хранилище содержит картотеку с информацией о высоких незамужних блондинках, а первый байт каждой записи представляет их возраст. Ставим перед собой задачу: выбрать всех представительниц моложе тридцати лет в порядке возрастания. Для этого реализуем два класса, `MyRecordFilter` и `MyRecordComparator`:

```
// класс фильтра
private class MyRecordFilter implements RecordFilter {

    public boolean matches(byte [] candidate)
    {
        // проверяем первый байт записи
        if(candidate[0]<30) return true; else return false;
    }
}

// класс компаратора
private class MyRecordComparator implements RecordComparator {

    public int compare(byte[] rec1, byte[] rec2)
    {
        // устанавливаем порядок записей по первому байту
        if(rec1[0]<rec2[0]) return RecordComparator.PRECEDES;
        if(rec1[0]>rec2[0]) return RecordComparator.FOLLOWS;
        return RecordComparator.EQUIVALENT;
    }
}

// открыть хранилище записей с именем "Blondes"
RecordStore recordStore = RecordStore.openRecordStore("Blondes", true);
// создать объект фильтра
MyRecordFilter filter = new MyRecordFilter();
// создать объект компаратора
MyRecordComparator comparator = new MyRecordComparator();
// получить список записей согласно критерию в порядке возрастания
RecordEnumeration re = recordStore.enumerateRecords
    (filter, comparator, false);
...

```


Таким образом, мы получили список интересующих нас кандидатур, с которыми теперь мы можем производить любые действия, как было показано в первом примере.

RecordListener — лови момент

Для контроля над хранилищем записей используют блок прослушивания записей. Применение блока прослушивания более актуально для организации многопоточных приложений, на которых мы остановимся более подробно в следующей главе. Смысл блока в том, чтобы отследить момент удаления, добавления или изменения записи в хранилище. При возникновении одного из этих событий автоматически вызовется соответствующая функция.

Для того чтобы создать блок прослушивания, нужно реализовать интерфейс `RecordListener`, в котором объявлены следующие методы:

- `recordAdded(RecordStore recordStore, int recordId)` — вызывается сразу после добавления в хранилище `recordStore` новой записи с номером `recordId`;
- `recordChanged(RecordStore recordStore, int recordId)` — вызывается сразу после того, как запись с номером `recordId` была изменена;
- `recordDeleted(RecordStore recordStore, int recordId)` — вызывается после того, как запись с номером `recordId` была удалена. Функция вызывается после удаления, поэтому не стоит пытаться получить в ней запись с номером `recordId` — в этом случае будет сформировано исключение `InvalidRecordIDException`.

Процедура взаимосвязи блока прослушивания и конкретного хранилища записей осуществляется с помощью следующих методов хранилища: `addRecordListener(RecordListener listener)`, `removeRecordListener(RecordListener listener)`. Один блок прослушивания может обслуживать одновременно несколько хранилищ.

RecordStoreException — возможные проблемы

Остановимся на обработке исключительных ситуаций, которые могут возникнуть при работе с системой управления записями. Все виды исключений RMS наследованы от одного класса `RecordStoreException`, поэтому в try-блоке можно отслеживать только объект этого класса, как и было сделано в примере.

Если же мы хотим получить более подробную информацию о возникшей ошибке или предпринять особые действия для устранения возникшей проблемы, то следует добавить обработку следующих исключений:

- `InvalidRecordIDException` — формируется в случае невозможности выполнения операции из-за некорректного номера записи;
- `RecordStoreFullException` — операция не может быть завершена, так как хранилище переполнено, то есть памяти, выделенной конкретной моделью, оказалось недостаточно;

- `RecordStoreNotFoundException` — формируется при попытке открыть хранилище с несуществующим именем;
- `RecordStoreNotOpenException` — возникает при попытке работы с неоткрытым хранилищем.

Таким образом, большинства из возможных проблем можно избежать, грамотно проектируя приложение. Следует внимательно следить за своевременным открытием и закрытием хранилища, актуальностью номеров записей. Распространенная ошибка — работа со списком записей, полученным до удаления или добавления некоторых записей, поэтому не забываем про метод `rebuild()`.

High Score — таблица рекордов

Вернемся к нашим змеям и попробуем усовершенствовать игру, применив вышеизложенное на практике. В долговременном хранилище записей будем хранить запись, состоящую из двух байт, первый из которых будет содержать текущий рекорд, а второй — уровень игры, от которого будет зависеть скорость передвижения змейки. Для реализации наших идей нам потребуется импортировать пакеты поддержки хранилища данных, а также завести новые поля в классе `Snake`:

```
import javax.microedition.rms.RecordStore;
import javax.microedition.rms.RecordEnumeration;
import javax.microedition.rms.RecordStoreException;
...
private RecordStore recordStore; // хранилище данных
private int recordID; // ID записи параметров
private byte level; // уровень сложности игры
private byte speed; // коэффициент вычисления задержки треда
private byte highScore; // текущий рекорд
```

В конструкторе класса `Snake` добавим считывание параметров из хранилища данных или создание новой записи для первого запуска игры:

```
// параметры для долговременного хранения
// первый байт - текущий рекорд
// второй байт - уровень сложности игры
byte buff[] = {0, 5};
try {
    // открыть хранилище записей с именем "SNAKE"
    recordStore = RecordStore.openRecordStore("SNAKE", true);
    // получить список записей хранилища
    RecordEnumeration re = recordStore.enumerateRecords(null, null, false);
    // если хранилище пусто
    if (re.numRecords() == 0)
        // добавить новую запись параметров игры
        recordID = recordStore.addRecord(buff, 0, 2);
    else
        // получить id записи параметров игры
```

```

        recordID = re.nextRecordId();
        // считать запись параметров игры
        buff = recordStore.getRecord(recordID);
        // первый байт - текущий рекорд
        highScore = buff[0];
        // второй байт - уровень сложности игры
        level = buff[1];
        // установить коэффициент вычисления задержки
        speed = 100;
    } catch (RecordStoreException rse) {
    }

```

Теперь при завершении игры будем сравнивать набранный результат с текущим рекордом. Сообщение о новом рекорде будем выводить красным цветом, а результат записывать в хранилище данных (рис. 7.3).

Добавим эти изменения в метод paint():

```

...
// если поднят флаг конца игры
if ( gameOverFlag ) {
    // вывести сообщение конца игры
    g.drawString("GAME OVER",width/2,height/2-10,g.HCENTER|g.TOP);
    // строка сообщения счета игры
    String scoreStr;
    // массив параметров для долговременного хранения
    byte buff[] = new byte[2];
    // если текущий рекорд меньше нового результата
    if(highScore < snakeLen) {
        highScore = (byte)snakeLen;
        // инициализировать массив с новыми параметрами
        buff[0] = highScore;
        buff[1] = level;
        try {
            // записать новый рекорд в хранилище
            recordStore.setRecord(recordID,buff,0,2);
        } catch (RecordStoreException rse) {
        }
        // установить цвет текста красным
        g.setColor(0xff0000);
        // сформировать строку с новым рекордом
        scoreStr = new String("HIGH SCORE: "+snakeLen);
    } else
        // сформировать строку со счетом
        scoreStr = new String("YOUR SCORE: "+snakeLen);
    // вывести счет
    g.drawString(scoreStr,width/2,height/2+10,g.HCENTER|g.TOP);
} else

```

Готово, можно уже запускать! Правда, играть стало гораздо интереснее? Теперь можно ставить новые рекорды. Именно этим параметром увлекают многие и многие игрушки, как компьютерные, так и мобильные. Вспомните, как в детстве, раз за разом, вы запускали игрушку, даже такую тупую, как «Ну, погоди!» или «Тайны океана», и старались изо всех сил вписать свое имя в таблицу славы, побив чужой рекорд. Самые хитрые, естественно, ничего не проходили, а просто вскрывали программу и приписывали себе любые результаты. Как это делается, мы рассмотрим в одной из заключительных глав.

Заметим, что для хранения текущего рекорда у нас отведен всего один байт, то есть максимальный рекорд, который можно поставить, ограничивается длиной змеи, равной 255 элементам. Хотя с такой реализацией увеличения скорости после каждого «съеденного» сердечка нам это не грозит. При изменении этой логики, возможно, следует добавить памяти для хранения текущего рекорда. При проектировании приложения всегда стоит задуматься, отвечает ли заведенный тип данных характеру хранимых данных. Как мы уже видели, тип `byte` может принимать значения от 0 до 255.

В данный момент мы реализовали хранение только одного рекорда, а хотелось бы, конечно, иметь таблицу лучшей десятки с их именами.

Всему свое время: пока что нам не хватает знаний о реализации ввода текста и работе со строками, но и до этого мы обязательно доберемся.

* * *

В этой главе мы достаточно подробно рассмотрели систему управления записями RMS, которая представляет собой простую абстракцию базы данных (RecordStore), связанную с записями. Записи не имеют определенного типа, а хранятся как массив байтов. Записи можно получать по уникальному идентификатору записи (ID), если он известен, или же получить список всех записей. Фильтр записей (Record Filter) реализует механизм запросов, с помощью которого можно выбирать только соответствующие каким-то критериям записи. Компаратор записей (Record Comparator) определяет порядок извлечения записей из списка и реализует сортировку записей.

Стоит отметить, что производительность RMS даже в современных моделях достаточно низка, поэтому стоит использовать RMS только в тех случаях, когда



Рис. 7.3. Сообщение о новом рекорде

это действительно необходимо. Есть мнение, что производительность может увеличиваться, если не обновлять некоторые элементы, а переписать полностью все содержимое хранилища.

Чтобы оценить производительность написанной программы, лучше воспользоваться реальным аппаратом, поскольку эмулятор в этом случае может не отобразить реальной картины. Вот, собственно, и все. Мы еще вернемся к системе управления записями в одном из следующих примеров.

Глава 8

Стандартные средства пользовательского интерфейса

Если вы смогли оторваться от игры, побив все мыслимые рекорды по откармливанию змей, то продолжим уроки мобильного программирования. Посмотрим критическим взглядом на нашу игру и подумаем, чего в ней еще не хватает. Уважающий себя театр должен начинаться с гардероба, а приложение для мобильного телефона, как и ресторан, начинается с меню. Меню мы изобретать не будем, а воспользуемся готовым пользовательским интерфейсом высокого уровня.

Класс List

Простейший вариант организации меню — использование готового интерфейса высокого уровня, предоставляющего все необходимые функции навигации и выбора одного из пунктов. Меню представлено классом `List`, который наследован от класса `Screen`, то есть является объектом, готовым к демонстрации через менеджер дисплея. Создается объект меню с помощью одного из конструкторов:

```
List(String title, int listType)
```

или

```
List(String title, int listType, String[] stringElements,  
      Image[] imageElements)
```

Первый конструктор создает пустой объект, в который можно будет добавить необходимые пункты, второй же принимает пункты меню в виде массива строк `stringElements`. Параметр `imageElements` определяет массив изображений, которые можно поставить в соответствие каждому пункту меню. Массив `stringElements`, как и каждый его элемент, не должен быть пустым. Длина массива определяет количество пунктов меню. Массив `imageElements` может иметь значение `null` или содержать такое же количество элементов, как и массив `stringElements`.

Параметр `title` задает заголовок меню, а `listType` определяет один из трех типов меню, представленных следующими константами класса `List`:

- **IMPLICIT** — каждое перемещение по пунктам меню вызывает немедленное оповещение через блок прослушивания команд. Класс `List` содержит особую команду `Command SELECT_COMMAND`, которая формируется при выборе нового пункта и передается в качестве аргумента в метод блока прослушивания `commandAction(Command, Displayable)`;

- **EXCLUSIVE** — позволяет выбор единственного пункта меню; оповещения приложения не генерируются;
- **MULTIPLE** — позволяет выбор нескольких пунктов меню в любом сочетании; оповещения не генерируются.

Каждый из пунктов меню может находиться в двух состояниях — выбранный или невыбранный. В режимах работы **IMPLICIT** и **EXCLUSIVE** выбранным может быть только один пункт меню, в режиме **MULTIPLE** выбранными могут быть несколько пунктов меню одновременно.

Класс **List** расширяет класс **Screen** и реализует интерфейс **Choice**. Все его методы определяются реализуемым интерфейсом.

Interface Choice

Интерфейс **Choice** определяет набор методов для элементов пользовательского интерфейса верхнего уровня, предоставляющих возможность выбора из нескольких предложенных вариантов (класс **List** и класс **ChoiceGroup**). Каждый из вариантов представляется текстовой строкой и необязательным графическим изображением — пиктограммой.

Для организации работы с меню интерфейс **Choice** определяет следующие методы, реализованные в классе **List**:

- **int size()** — возвращает количество пунктов меню. Все пункты меню индексируются целыми числами от 0 до **size()-1**;
- **int append(String stringPart, Image imagePart)** — добавляет новый пункт меню, представленный строкой **stringPart** и картинкой **imagePart**, в конец списка и возвращает присвоенный элементу индекс;
- **void insert(int elementNum, String stringPart, Image imagePart)** — аналогично предыдущему методу, вставляет новый пункт меню в позицию списка, заданную параметром **elementNum**;
- **void delete(int elementNum)** — удаляет пункт меню с индексом **elementNum**;
- **void set(int elementNum, String stringPart, Image imagePart)** — изменяет пункт меню с индексом **elementNum** в соответствии с параметрами **stringPart** и **imagePart**;
- **String getString(int elementNum)** — возвращает строку, представляющую пункт меню с индексом **elementNum**;
- **Image getImage(int elementNum)** — возвращает картинку, связанную с пунктом меню, имеющим индекс **elementNum**;
- **boolean isSelected(int elementNum)** — возвращает значение **true**, если пункт меню с индексом **elementNum** является выделенным;
- **int getSelectedIndex()** — возвращает индекс выделенного пункта меню. В режиме **MULTIPLE** всегда возвращает значение **-1**;
- **void setSelectedIndex(int elementNum, boolean selected)** — в режиме **MULTIPLE** устанавливает пункт меню с индексом **elementNum** в состояние, соответствующее параметру **selected**. В остальных режимах параметр **selected** должен иметь значение **true**, при этом соответствующий пункт меню переходит в выделен-

ное состояние. Пункт, бывший до этого в выделенном состоянии, переходит в невыделенное состояние. В режиме IMPLICIT никаких неявных вызовов не происходит;

- `int getSelectedFlags(boolean[] selectedArray_return)` — в параметр `selectedArray_return` возвращается массив флагов, соответствующих пунктам меню. Элемент массива содержит значение `true`, если выделен соответствующий пункт меню. В режиме `MULTIPLE` значение `true` может содержать произвольное число элементов массива, в остальных режимах — только один элемент. Сам же метод возвращает количество выделенных элементов;
- `void setSelectedFlags(boolean[] selectedArray)` — устанавливает пункты меню в выделенное состояние в соответствии с массивом флагов `selectedArray`. В режимах `EXCLUSIVE` и `IMPLICIT` только один элемент массива `selectedArray` должен иметь значение `true`. Если таких элементов нет, то выделенным становится первый пункт меню. Если элементов больше, чем один, то первый соответствующий пункт принимает выделенное состояние, остальные игнорируются.

Класс SnakeGame

Добавим в игру стартовое меню, которое будет иметь несколько команд: начать новую игру, установить уровень сложности игры и посмотреть текущий рекорд. При запуске игры сначала будем демонстрировать меню, а по окончании игры возвращаться туда же, а не зависать намертво, как мы делали до этого. Как мы уже выяснили, для воплощения наших идей нам потребуется создать объект класса `List`, который будет представлять стартовое меню игры, а также реализовать интерфейс `CommandListener` для прослушивания команд.

Для начала нам потребуется подключить несколько пакетов, реализующих необходимые классы:

```
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.List;
```

Интерфейс `CommandListener` мы реализуем в основном классе мидлета `SnakeGame`. Его декларация теперь будет выглядеть следующим образом:

```
public class SnakeGame extends MIDlet implements CommandListener
```

Также нам потребуются два новых члена класса, представляющих собственно объект меню, а также команда для выбора пункта меню:

```
private List menu;           // стартовое меню
private Command ok;          // команда выбора пункта меню
```

Дальнейшие изменения произведем в методе `startApp()`, из которого уберем создание объекта змейки и старт нового треда, а добавим создание меню и отображение его на экране. Теперь стартовый метод мидлета будет выглядеть так:

```
public void startApp() {
    // массив строк с названиями пунктов меню
    String menuOptions[] = {"New Game", "Set Level", "High Score"};
```

```

// создать новое меню
menu = new List("", List.IMPLICIT, menuOptions, null);
// создать команду выбора пункта меню
ok = new Command("Ok", Command.OK, 1);
// добавить команду в меню
menu.addCommand(ok);
// установить блок прослушивания команд для меню
menu.setCommandListener(this);
// получить ссылку на менеджер дисплея
display = Display.getDisplay(this);
// отобразить меню на экране
display.setCurrent(menu);
}

```

Чтобы откомпилировать и посмотреть, что у нас получилось, нужно реализовать метод `commandAction`, представляющий блок прослушивания команд. Этот метод и будет предписывать те или иные действия в зависимости от выбранного пункта меню. Пока что можно оставить его пустым, откомпилировать приложение и посмотреть, как выглядит меню на экране телефона (рис. 8.1).



Рис. 8.1. Теперь игра начинается с меню

Итак, мы видим на экране аппарата строки, представляющие пункты меню, заданные нами в массиве `menuOptions`. Один из пунктов подсвечен синей полоской. Одна из функциональных клавиш связана с командой `Ok`, при нажатии на которую, правда, пока что ничего не происходит.

Перед тем как продвигаться дальше, стоит тщательно обдумать структуру приложения. Это относится к любому программированию вообще, поскольку просчеты в проектировании программы могут привести к тому, что придется начинать с чистого листа и все переписывать заново. Так что, как говорится, лучше час потерять, а потом за пять минут долететь.

Обратить внимание следует на логику смены экранов. Если для начала новой игры нам просто потребуется создать объект змейки и отобразить его на экране, то для демонстрации текущего рекорда и установки нового уровня сложности потребуется новый отображаемый объект, который будет нести необходимую информацию, а также обладающий возможностью возврата к стартовому меню программы. В качестве отображаемого объекта выберем уже знакомую нам форму. Для удобства напомним новый метод, который будет создавать форму, принимая необходимые параметры, и отображать ее на экране.

```
private void showNewScreen(String title, Command c, Item item) {  
    // создать форму для нового экрана  
    Form newScreenForm = new Form(title);  
    // добавить команду для возврата из формы  
    newScreenForm.addCommand(c);  
    // добавить элемент в форму  
    newScreenForm.append(item);  
    // установить блок прослушивания команд  
    newScreenForm.setCommandListener(this);  
    // отобразить форму на экране  
    display.setCurrent(newScreenForm);  
}
```

Обратим внимание, что и стартовое меню программы, и новая форма пользуются одним и тем же блоком прослушивания команд, реализованном в основном классе мидлета SnakeGame.

Кроме того, текущий рекорд и уровень сложности игры связаны с параметрами, находящимися в хранилище данных, поэтому напомним еще один метод, который будет читать необходимую запись хранилища и возвращать нужный параметр:

```
private byte getParameter(int index) {  
    // массив параметров игры  
    byte buff[] = {0,0};  
    try {  
        // открыть хранилище записей с именем "SNAKE"  
        RecordStore recordStore = RecordStore.openRecordStore  
            ("SNAKE", true);  
        // получить список записей хранилища  
        RecordEnumeration re = recordStore.enumerateRecords  
            (null, null, false);  
        // если хранилище не пусто  
        if (re.numRecords() != 0) {  
            // получить id и считать запись параметров игры  
            buff = recordStore.getRecord(re.nextRecordId());  
        }  
    } catch (RecordStoreException rse) {  
    }  
    // вернуть параметр  
    return buff[index];  
}
```

Перейдем к блоку прослушивания команд, который, собственно, и будет дирижировать нашей программой и предписывать действия в зависимости от выбранного пункта меню. В обязательном к реализации методе `commandAction` есть два входных параметра, которые определяют вызванную команду и отображаемый экран, из которого была вызвана команда. В зависимости от этих парамет-

ров мы и будем определять необходимые действия. В первую очередь обработаем команду Ok:

```
public void commandAction(Command c, Displayable d) {
    // если была команда Ok из главного меню игры
    if (c == ok && d == menu) {
        // получить выбранный пункт меню
        int selIndex = menu.getSelectedIndex();
        switch(selIndex) {
            // первый пункт меню – новая игра
            case 0 :
                // создать объект змейки
                curSnake = new Snake();
                // создать объект треда автоматического передвижения
                Thread moveThread = new Thread(curSnake);
                // начать выполнение треда
                moveThread.start();
                // отобразить змейку на экране
                display.setCurrent(curSnake);
                break;
            // второй пункт меню – установить уровень сложности
            case 1 :
                // массив строк с названиями пунктов меню
                String levelOptions[] = {"Level 1", "Level 2",
                                         "Level 3", "Level 4"};
                // создать меню выбора уровня игры
                LevelChoice =
                    new ChoiceGroup("", List.EXCLUSIVE, levelOptions, null);
                // выделить пункт меню
                // соответствующий текущему уровню сложности
                LevelChoice.setSelectedIndex(5 - getParameter(1).true);
                // отобразить новую форму
                showNewScreen("Set Level", set, levelChoice);
                break;
            // третий пункт меню – посмотреть текущий рекорд
            case 2 :
                // получить строку с текущим рекордом
                String strScore = (new Integer(getParameter(0))).toString();
                // создать объект элемента-строки
                StringItem highScoreItem = new StringItem("", strScore);
                // отобразить новую форму
                showNewScreen("HIGH SCORE", ok, highScoreItem);
                break;
        }
    }
    // если была команда Ok, вызванная из формы
    if (c == ok && d != menu)
```

```
// отобразить на экране стартовое меню
display.setCurrent(menu);
```

Итак, как мы видим из кода, меню установки уровня сложности представлено объектом класса `ChoiceGroup`, который принадлежит к иерархии класса `Item`, поэтому он может быть отображен с помощью формы. На этом примере мы наглядно видим использование полиморфизма объектно-ориентированного языка: метод `showNewScreen` принимает в качестве одного и того же аргумента различные объекты классов `ChoiceGroup` и `StringItem`, порожденных одним классом `Item`.

Из собственного опыта у меня сформировался следующий подход к программированию: при добавлении любой новой функциональности стоит запустить программу и убедиться, что все работает корректно. Такой подход позволяет вовремя отследить допущенные просчеты и ошибки. Именно поэтому рекомендую не пытаться написать сразу всю программу целиком, чтобы потом ломать голову, где же в этой куче кода была допущена ошибка, а продвигаться от главы к главе постепенно. Это не детектив, чтобы заглядывать в конец книжки: кто же убийца? Полный листинг программы в конце приведен не для бездумного переписывания, а для дополнительного контроля и помощи в поиске ошибок.

Для того чтобы откомпилировать и запустить программу на данном этапе, требуется несколько дополнительных штрихов. Первое: требуется импортировать необходимые пакеты, классы которых мы будем использовать:

```
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Item;
import javax.microedition.lcdui.ChoiceGroup;
import javax.microedition.lcdui.StringItem;
```

Второе: в классе `SnakeGame` мы завели два новых поля для реализации меню выбора уровня игры:

```
private ChoiceGroup levelChoice; // меню выбора уровня сложности игры
private Command set; // команда выбора уровня сложности игры
```

В стартовом методе мидлета `startApp()` создадим объект команды выбора уровня сложности игры:

```
set = new Command("Set", Command.BACK, 1);
```

Вот теперь можно запустить игру. Обратите внимание, что обработка команды `set` в блоке прослушивания команд еще не реализована, поэтому пока что удовлетворимся просмотром текущего рекорда (рис. 8.2).

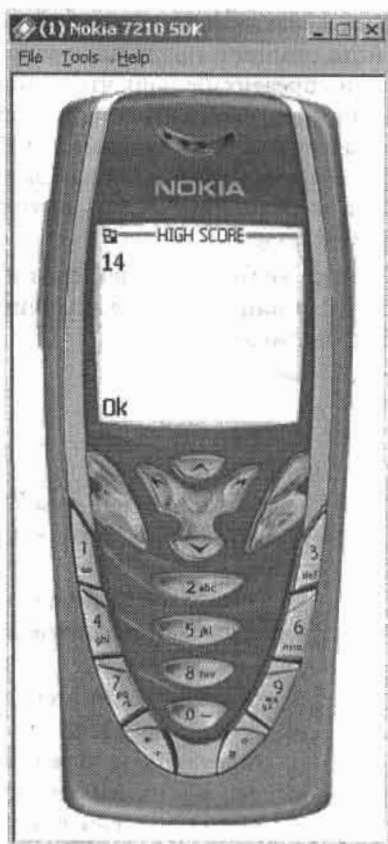


Рис. 8.2. Демонстрация наилучшего результата

При нажатии функциональной клавиши формируется команда `Ok`, и приложение снова демонстрирует на экране стартовое меню. Выглядит это все, конечно, не очень презентабельно, что обусловлено ограниченностью возможностей формы. Например, наложить текст на картинку мы не сможем при всем желании. Так что в качестве самостоятельной работы можете попробовать использовать вместо формы класс, порожденный от класса `Canvas`. В методе `paint` можно будет нарисовать все, что душе угодно, а возврат в основное меню отработать с помощью метода `keyPressed`.

Однако не будем отвлекаться и допишем в блоке прослушивания команд обработку команды `set`, где получим выбранный уровень сложности и запишем его в хранилище записей:

```
if(c == set) {
    try {
        // массив параметров игры
        byte buff[] = {0,5};
        // открыть хранилище записей с именем "SNAKE"
        RecordStore recordStore = RecordStore.openRecordStore
            ("SNAKE", true);
        // получить список записей хранилища
        RecordEnumeration re = recordStore.enumerateRecords
            (null, null, false);
        // если хранилище не пусто
        if (re.numRecords()!=0) {
            // получить id записи параметров игры
            int id = re.nextRecordId();
            // считать запись с параметрами игры
            buff = recordStore.getRecord(id);
            // вычислить текущий уровень игры
            buff[1] = (byte)(5 - levelChoice.getSelectedIndex());
            // записать параметры игры
            recordStore.setRecord(id,buff,0,2);
        } else {
            // добавить новую запись параметров игры
            recordStore.addRecord(buff, 0, 2);
        }
    } catch(RecordStoreException rse) {
    }
    // отобразить на экране стартовое меню
    display.setCurrent(menu);
}
```

Теперь все пункты стартового меню полностью соответствуют заявленным действиям. Попробуем, как работает меню выбора уровня сложности игры (рис. 8.3).

В данном случае мы использовали режим `EXCLUSIVE`, поэтому меню отличается от стартового внешним видом. Команда `Select` появилась автоматически и отвечает за выделение одного из пунктов меню точкой, так как перемещение подсвеченной

синей подоски по пунктам не приводит к выделению одного из них. Реализованная нами команда Set вызывает запись нового уровня сложности, который учитывается при запуске новой игры. Теперь каждый может задать свой собственный темп, что только добавит увлекательности нашей игре.

Класс Gauge

Благодаря гибкой реализации и полиморфизму, можно легко поменять реализацию выбора сложности игры, используя, например, не меню выбора ChoiceGroup, а шкалу Gauge. Шкала также принадлежит иерархии класса Item, поэтому метод showNewScreen изменять не потребуется. Внешний вид шкалы, как и всех объектов пользовательского интерфейса высокого уровня, зависит от конкретной модели телефона. Например, в стандартном цветном эмуляторе шкала будет выглядеть так (рис. 8.4).

Здесь мы видим привычные глазу прямоугольнички, меняющие цвет в зависимости от выбора. У разработчиков корпорации Nokia свое видение шкалы, поэтому то же самое на эмуляторе Nokia 7210 будет выглядеть следующим образом (рис. 8.5).

Изменить внешний вид этого элемента у нас нет никакой возможности, поэтому придется довольствоваться имеющимися методами:

- `Gauge(String label, boolean interactive, int maxValue, int initialValue)` — конструктор класса Gauge. Создает шкалу с заголовком `label` и возможным диапазоном значений от 0 до `maxValue`. Значение по умолчанию выставляется равным параметру `initialValue`. Если параметр `interactive` имеет значение `true`, то значение шкалы можно изменять, иначе, как в стрип-клубе: смотреть — можно, трогать — нельзя;
- `int getMaxValue()` — возвращает максимально возможное значение шкалы;
- `int getValue()` — возвращает текущее значение шкалы;
- `boolean isInteractive()` — возвращает значение `true`, если пользователь может изменять значение шкалы;
- `void setLabel(String label)` — устанавливает новый заголовок шкалы;
- `void setMaxValue(int maxValue)` — устанавливает новое максимально возможное значение шкалы;
- `void setValue(int value)` — возвращает выбранное пользователем значение шкалы.

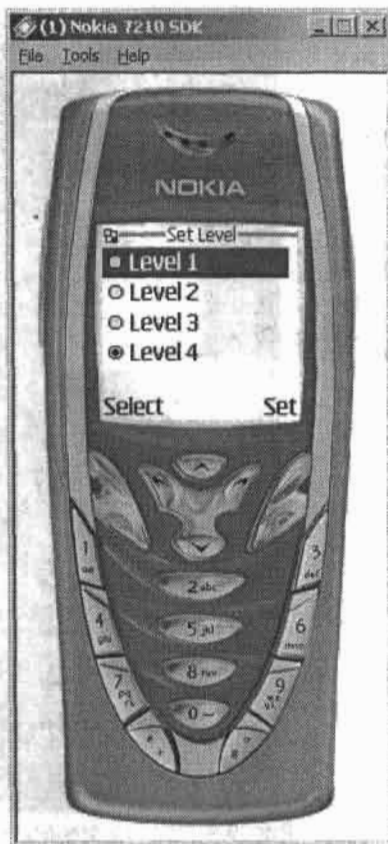


Рис. 8.3. Меню выбора уровня сложности



Рис. 8.4. Шкала в стандартном эмуляторе

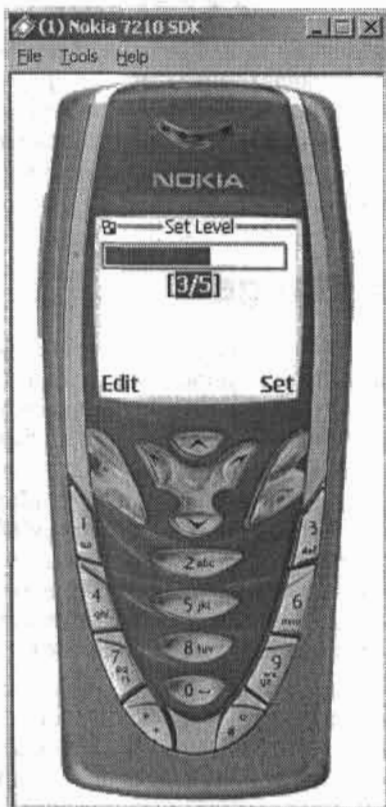


Рис. 8.5. Шкала на эмуляторе Nokia 7210

Последний штрих для полного счастья — заставка, которая будет выводиться перед запуском игры. Сохраним картинку для заставки в файле `title.png` и добавим в стартовом методе приложения `startApp()` следующий код:

```
Image title = null;
try {
    // создать объект заставки
    title = Image.createImage("/title.png");
} catch (IOException ioe) {}
// создать новый элемент формы
ImageItem item = new ImageItem("", title, ImageItem.LAYOUT_CENTER, "");
// отобразить заставку
showNewScreen("SNAKE".ok, item);
```

Для отображения заставки мы воспользовались все тем же универсальным методом `showNewScreen`, который в качестве аргумента принимает объект класса `ImageItem`, содержащий картинку (рис. 8.6).

Вот теперь точно — все!

В этой главе мы разобрались с организацией меню с помощью пользовательского интерфейса высокого уровня. На этом наше знакомство с компонентами интерфейса высокого уровня не заканчивается. Мы еще вернемся к некоторым из них в других примерах. Лучше всего такие вещи усваиваются на практике: почитал — сразу реализовал — и почувствовал силу в руках.

Основной принцип высокоуровневого интерфейса в том, что все объекты иерархии класса Displayable можно отобразить на экране телефона, а все объекты иерархии класса Item можно включить в отображаемую форму. Всегда требуется следить за переходами по пунктам меню: доступен ли необходимый отображаемый объект в конкретный момент времени. В нашем случае основной класс приложения содержит стартовое меню в качестве поля, а форма второго уровня создается каждый раз заново при вызове соответствующего пункта меню с необходимыми параметрами. Немного практики, и у вас все получится.

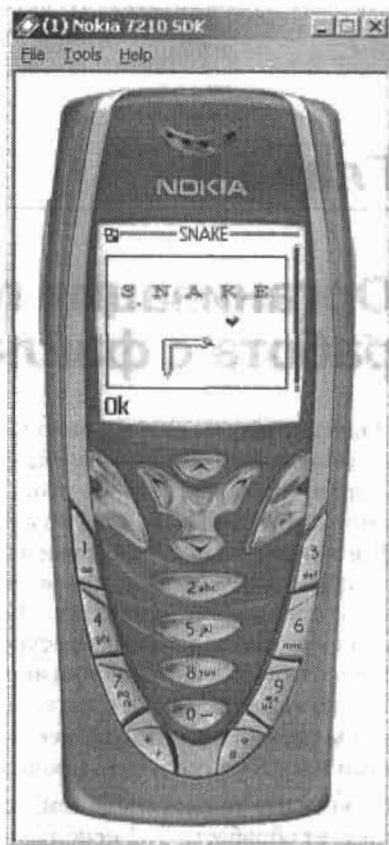


Рис. 8.6. Теперь у игры есть даже заставка

Глава 9

Организация потоков, работа с файлами

Надеюсь, что вы старательно выполняете самостоятельные работы и к моменту чтения этой главы уже смогли похвастаться перед друзьями своими руками сотворенной «Змейкой». В этой игре мы выводили на экран, главным образом, картинки, графику и прочую визуализацию, которую приложение считывало из файлов-ресурсов. Ресурсами в этом случае являлись картинки в определенном, воспринимаемом телефоном, формате. На самом же деле ресурсами могут являться не только картинки. Любая информация в самом разнообразном виде может храниться в файле-ресурсе. Каким образом информация из файла `title.png` превратилась в картинку на экране мобильного, осталось для нас за кадром: мы просто вызвали метод `createImage` и получили готовый объект `Image`. В этой главе мы рассмотрим подробнее, что такое потоки, как работать с файлами-ресурсами и как организовать процедуру чтения из файла.

На этот раз только мы знаем, какого типа информацию содержит ресурс, как ее следует обрабатывать, использовать или отображать. Черновую работу формирования объекта из файла, как это было в случае с картинкой, выполнять теперь за нас некому.

Класс `InputStream`

Связь между источником данных и приложением осуществляется с помощью объекта класса `InputStream`, который можно сравнить с трубой, пропускающей через себя поток данных. Класс `InputStream` является базовым для всех классов ввода и предоставляет основные методы работы с потоком ввода:

- `int read()` — прочитать следующий байт из потока данных. Эта функция является абстрактной, то есть в любом классе-потомке обязательна ее реализация;
- `int read(byte[] b)` — считать данные из потока в буфер. Функция читает все доступные данные, ограничиваясь размером буфера, и возвращает количество прочитанных байт;
- `int read(byte[] b, int off, int len)` — считать данные длины `len` из потока данных и разместить их в буфере, начиная с позиции `off`;
- `int available()` — возвращает количество байт, доступных для чтения в данном потоке;

- `void mark(int readlimit)` — помечает текущую позицию в потоке данных с целью предоставления возможности повторного чтения с этой позиции. Аргумент указывает, сколько байт может быть прочитано, прежде чем метка станет недействительной;
- `boolean markSupported()` — проверка возможности повторного чтения данных с помеченной позиции. Возвращает `true`, если позиция была задана и число прочитанных после этого данных не превысило лимита;
- `void reset()` — возвращает поток данных в состояние, сохраненное при последнем вызове функции `mark`;
- `long skip(long n)` — пропустить `n` байтов данных потока. Возвращает реальное количество пропущенных байт;
- `void close()` — закрывает поток и освобождает все используемые им ресурсы.

Следует отметить, что в базовом классе `InputStream` такие функции, как `available`, `mark`, `reset`, работать не будут. Они объявлены для последующей реализации в классах-потомках, а пока что не выполняют никаких действий и возвращают нулевое значение.

Рассмотрим использование класса `InputStream` на простом примере вывода текста из файла на экран телефона. Поместим текстовый файл `pushkin.txt` в папку ресурсов приложения `/res`. Чтобы связать ресурс с потоком, воспользуемся методом `getResourceAsStream(String name)` класса `Class`. Затем с помощью метода потока `read` считаем содержимое файла и поместим его на форму, связанную с экраном. Пример кода будет выглядеть так:

```
...
import java.io.InputStream;
import java.io.IOException;
...
Form form = new Form();
Display display = Display.getDisplay(this);
// связать файл с потоком is
InputStream is = getClass().getResourceAsStream("/pushkin.txt");
byte[] bArr = new byte[150];
try {
    is.read(bArr); // считать содержимое потока в массив байт
    catch(IOException e) { }
    // сформировать строку из массива байт
    String str = new String(bArr);
    form.append(str); // поместить строку в форму
    display.setCurrent(form); // отобразить форму на экране
...

```

Запускаем приложение в стандартном эмуляторе из WTK и получаем вот такой замечательный результат (рис. 9.1).

Эмулятор справился с заданием на твердую пятерку: навигацию нам реализовала форма, стрелочки перематывают текст вниз и вверх, на экране не появляется ника-

ких лишних символов, связанных с тем, что размер запрошенного нами массива (`new byte[150]`) превышает объем текста. Красота!

Но, как показала практика, радоваться было рано. Запускаем наш пример на эмуляторе от Nokia 7210, что более близко соответствует запуску на реальном аппарате, и видим печальную картину (рис. 9.2).



Рис. 9.1. Вы знали, что на телефоне можно читать книги?



Рис. 9.2. Осталось только научиться это читать...

Русский текст превратился в шифровку агентов МОСАДа, а снизу прилипли квадратики неиспользованных байтов массива. Отсюда делаем вывод, что кодировка `win1251` не совпала с той, что запрограммирована в аппарате Nokia. Единственное правильное решение в данной ситуации — пользоваться универсальной кодировкой, которая подходила бы ко всем моделям. Благо, такая кодировка существует. Ее название говорит само за себя — Unicode.

Встает вопрос: как теперь перевести текст с привычной глазу кириллицы на Unicode? Если текст находится в коде программы и является, например, заголовком формы или текстовым сообщением, то можно воспользоваться утилитой `native2ascii` из Java SDK, который мы ставили в самую первую очередь.

Вызывается эта утилита из командной строки следующим образом: `.. \bin \ native2ascii -encoding cp1251 file1.java > file2.java`, где параметр `encoding` указывает

исходную кодировку, `file1` — исходный файл, `file2` — конвертированный файл. Обратный перевод из Unicode на родной язык осуществляется с использованием параметра `-reverse`.

Например, после преобразования исходная строка

```
form.append("Здравствуй. мир!");
```

будет выглядеть так:

```
form.append("\u0417\u0443\u0434\u0440\u0432\u0441\u0442\u0443\u0432\u0443\u0439.\n\u043c\u0438\u0440!");
```

Такой вариант действительно решает проблему корректного отображения русского текста. Однако в нашем случае текст находится в файле-ресурсе приложения, а значит конверсия исходного кода программы нам не поможет. Предлагаемое решение основывается на поочередном преобразовании каждого символа при считывании его из потока.

Класс `DataInputStream`

На этот раз мы воспользуемся классом `DataInputStream`, который является наследником класса `InputStream` и реализует чтение основных типов данных. То есть, кроме основного метода `read()`, этот класс содержит методы `readBoolean()`, `readByte()`, `readChar()`, `readInt()`, `readLong()`, `readShort()`, `readUnsignedByte()`, `readUnsignedShort()`. Преобразование будем выполнять с помощью кодовой строки, которая каждому символу кириллицы ставит в соответствие его код в универсальной кодировке. Таким образом, наша программа приобретает следующий вид:

```
import java.io.DataInputStream;

...

Form form = new Form();
Display display = Display.getDisplay(this);
InputStream is = getClass().getResourceAsStream("/pushkin.txt");
DataInputStream dis = new DataInputStream(is);
int i = 0;
char[] bArr = new char[150];
try { // преобразовать символы в Unicode и записать в массив
    while(true) bArr[i++] = (char)convert(dis.readUnsignedByte());
}
catch(IOException ioe) { }
String str = new String(bArr, 0, i-1);
form.append(str);

...

char convert(int ch) {
    return (ch < 128) ? (char)ch : WIN1251_TO_UNICODE.charAt(ch-128);
}

// кодовая строка для преобразования символа из win1251 в unicode
private String WIN1251_TO_UNICODE =
    "\u0402\u0403\u0410\u0453\u041E\u0426\u0420\u0421\u040a\u0409\u040a\u040c\u040b\u040f\u0452\u0418\u0419\u041c\u041d\u0422\u0423\u0424\u0444\u0422\u0459\u043a\u045a\u045c\u0457\u0458\u0459\u045a\u045b\u045c\u045d\u045e\u045f\u0460\u0461\u0462\u0463\u0464\u0465\u0466\u0467\u0468\u0469\u046a\u046b\u046c\u046d\u046e\u046f\u0470\u0471\u0472\u0473\u0474\u0475\u0476\u0477\u0478\u0479\u047a\u047b\u047c\u047d\u047e\u047f\u0480\u0481\u0482\u0483\u0484\u0485\u0486\u0487\u0488\u0489\u048a\u048b\u048c\u048d\u048e\u048f\u0490\u0491\u0492\u0493\u0494\u0495\u0496\u0497\u0498\u0499\u049a\u049b\u049c\u049d\u049e\u049f\u04a0\u04a1\u04a2\u04a3\u04a4\u04a5\u04a6\u04a7\u04a8\u04a9\u04aa\u04ab\u04ac\u04ad\u04ae\u04af\u04b0\u04b1\u04b2\u04b3\u04b4\u04b5\u04b6\u04b7\u04b8\u04b9\u04ba\u04bb\u04bc\u04bd\u04be\u04bf\u04c0\u04c1\u04c2\u04c3\u04c4\u04c5\u04c6\u04c7\u04c8\u04c9\u04ca\u04cb\u04cc\u04cd\u04ce\u04cf\u04d0\u04d1\u04d2\u04d3\u04d4\u04d5\u04d6\u04d7\u04d8\u04d9\u04da\u04db\u04dc\u04dd\u04de\u04df\u04e0\u04e1\u04e2\u04e3\u04e4\u04e5\u04e6\u04e7\u04e8\u04e9\u04ea\u04eb\u04ec\u04ed\u04ee\u04ef\u04f0\u04f1\u04f2\u04f3\u04f4\u04f5\u04f6\u04f7\u04f8\u04f9\u04fa\u04fb\u04fc\u04fd\u04fe\u04ff\u0500\u0501\u0502\u0503\u0504\u0505\u0506\u0507\u0508\u0509\u050a\u050b\u050c\u050d\u050e\u050f\u0510\u0511\u0512\u0513\u0514\u0515\u0516\u0517\u0518\u0519\u051a\u051b\u051c\u051d\u051e\u051f\u0520\u0521\u0522\u0523\u0524\u0525\u0526\u0527\u0528\u0529\u052a\u052b\u052c\u052d\u052e\u052f\u0530\u0531\u0532\u0533\u0534\u0535\u0536\u0537\u0538\u0539\u053a\u053b\u053c\u053d\u053e\u053f\u0540\u0541\u0542\u0543\u0544\u0545\u0546\u0547\u0548\u0549\u054a\u054b\u054c\u054d\u054e\u054f\u0550\u0551\u0552\u0553\u0554\u0555\u0556\u0557\u0558\u0559\u055a\u055b\u055c\u055d\u055e\u055f\u0560\u0561\u0562\u0563\u0564\u0565\u0566\u0567\u0568\u0569\u056a\u056b\u056c\u056d\u056e\u056f\u0570\u0571\u0572\u0573\u0574\u0575\u0576\u0577\u0578\u0579\u057a\u057b\u057c\u057d\u057e\u057f\u0580\u0581\u0582\u0583\u0584\u0585\u0586\u0587\u0588\u0589\u058a\u058b\u058c\u058d\u058e\u058f\u0590\u0591\u0592\u0593\u0594\u0595\u0596\u0597\u0598\u0599\u059a\u059b\u059c\u059d\u059e\u059f\u05a0\u05a1\u05a2\u05a3\u05a4\u05a5\u05a6\u05a7\u05a8\u05a9\u05aa\u05ab\u05ac\u05ad\u05ae\u05af\u05b0\u05b1\u05b2\u05b3\u05b4\u05b5\u05b6\u05b7\u05b8\u05b9\u05ba\u05bb\u05bc\u05bd\u05be\u05bf\u05c0\u05c1\u05c2\u05c3\u05c4\u05c5\u05c6\u05c7\u05c8\u05c9\u05ca\u05cb\u05cc\u05cd\u05ce\u05cf\u05d0\u05d1\u05d2\u05d3\u05d4\u05d5\u05d6\u05d7\u05d8\u05d9\u05da\u05db\u05dc\u05dd\u05de\u05df\u05e0\u05e1\u05e2\u05e3\u05e4\u05e5\u05e6\u05e7\u05e8\u05e9\u05ea\u05eb\u05ec\u05ed\u05ee\u05ef\u05f0\u05f1\u05f2\u05f3\u05f4\u05f5\u05f6\u05f7\u05f8\u05f9\u05fa\u05fb\u05fc\u05fd\u05fe\u05ff\u0600\u0601\u0602\u0603\u0604\u0605\u0606\u0607\u0608\u0609\u060a\u060b\u060c\u060d\u060e\u060f\u0610\u0611\u0612\u0613\u0614\u0615\u0616\u0617\u0618\u0619\u061a\u061b\u061c\u061d\u061e\u061f\u0620\u0621\u0622\u0623\u0624\u0625\u0626\u0627\u0628\u0629\u062a\u062b\u062c\u062d\u062e\u062f\u0630\u0631\u0632\u0633\u0634\u0635\u0636\u0637\u0638\u0639\u063a\u063b\u063c\u063d\u063e\u063f\u0640\u0641\u0642\u0643\u0644\u0645\u0646\u0647\u0648\u0649\u064a\u064b\u064c\u064d\u064e\u064f\u0650\u0651\u0652\u0653\u0654\u0655\u0656\u0657\u0658\u0659\u065a\u065b\u065c\u065d\u065e\u065f\u0660\u0661\u0662\u0663\u0664\u0665\u0666\u0667\u0668\u0669\u066a\u066b\u066c\u066d\u066e\u066f\u0670\u0671\u0672\u0673\u0674\u0675\u0676\u0677\u0678\u0679\u067a\u067b\u067c\u067d\u067e\u067f\u0680\u0681\u0682\u0683\u0684\u0685\u0686\u0687\u0688\u0689\u068a\u068b\u068c\u068d\u068e\u068f\u0690\u0691\u0692\u0693\u0694\u0695\u0696\u0697\u0698\u0699\u069a\u069b\u069c\u069d\u069e\u069f\u06a0\u06a1\u06a2\u06a3\u06a4\u06a5\u06a6\u06a7\u06a8\u06a9\u06aa\u06ab\u06ac\u06ad\u06ae\u06af\u06b0\u06b1\u06b2\u06b3\u06b4\u06b5\u06b6\u06b7\u06b8\u06b9\u06ba\u06bb\u06bc\u06bd\u06be\u06bf\u06c0\u06c1\u06c2\u06c3\u06c4\u06c5\u06c6\u06c7\u06c8\u06c9\u06ca\u06cb\u06cc\u06cd\u06ce\u06cf\u06d0\u06d1\u06d2\u06d3\u06d4\u06d5\u06d6\u06d7\u06d8\u06d9\u06da\u06db\u06dc\u06dd\u06de\u06df\u06e0\u06e1\u06e2\u06e3\u06e4\u06e5\u06e6\u06e7\u06e8\u06e9\u06ea\u06eb\u06ec\u06ed\u06ee\u06ef\u06f0\u06f1\u06f2\u06f3\u06f4\u06f5\u06f6\u06f7\u06f8\u06f9\u06fa\u06fb\u06fc\u06fd\u06fe\u06ff\u0700\u0701\u0702\u0703\u0704\u0705\u0706\u0707\u0708\u0709\u070a\u070b\u070c\u070d\u070e\u070f\u0710\u0711\u0712\u0713\u0714\u0715\u0716\u0717\u0718\u0719\u071a\u071b\u071c\u071d\u071e\u071f\u0720\u0721\u0722\u0723\u0724\u0725\u0726\u0727\u0728\u0729\u072a\u072b\u072c\u072d\u072e\u072f\u0730\u0731\u0732\u0733\u0734\u0735\u0736\u0737\u0738\u0739\u073a\u073b\u073c\u073d\u073e\u073f\u0740\u0741\u0742\u0743\u0744\u0745\u0746\u0747\u0748\u0749\u074a\u074b\u074c\u074d\u074e\u074f\u0750\u0751\u0752\u0753\u0754\u0755\u0756\u0757\u0758\u0759\u075a\u075b\u075c\u075d\u075e\u075f\u0760\u0761\u0762\u0763\u0764\u0765\u0766\u0767\u0768\u0769\u076a\u076b\u076c\u076d\u076e\u076f\u0770\u0771\u0772\u0773\u0774\u0775\u0776\u0777\u0778\u0779\u077a\u077b\u077c\u077d\u077e\u077f\u0780\u0781\u0782\u0783\u0784\u0785\u0786\u0787\u0788\u0789\u078a\u078b\u078c\u078d\u078e\u078f\u0790\u0791\u0792\u0793\u0794\u0795\u0796\u0797\u0798\u0799\u079a\u079b\u079c\u079d\u079e\u079f\u07a0\u07a1\u07a2\u07a3\u07a4\u07a5\u07a6\u07a7\u07a8\u07a9\u07aa\u07ab\u07ac\u07ad\u07ae\u07af\u07b0\u07b1\u07b2\u07b3\u07b4\u07b5\u07b6\u07b7\u07b8\u07b9\u07ba\u07bb\u07bc\u07bd\u07be\u07bf\u07c0\u07c1\u07c2\u07c3\u07c4\u07c5\u07c6\u07c7\u07c8\u07c9\u07ca\u07cb\u07cc\u07cd\u07ce\u07cf\u07d0\u07d1\u07d2\u07d3\u07d4\u07d5\u07d6\u07d7\u07d8\u07d9\u07da\u07db\u07dc\u07dd\u07de\u07df\u07e0\u07e1\u07e2\u07e3\u07e4\u07e5\u07e6\u07e7\u07e8\u07e9\u07ea\u07eb\u07ec\u07ed\u07ee\u07ef\u07f0\u07f1\u07f2\u07f3\u07f4\u07f5\u07f6\u07f7\u07f8\u07f9\u07fa\u07fb\u07fc\u07fd\u07fe\u07ff\u0800\u0801\u0802\u0803\u0804\u0805\u0806\u0807\u0808\u0809\u080a\u080b\u080c\u080d\u080e\u080f\u0810\u0811\u0812\u0813\u0814\u0815\u0816\u0817\u0818\u0819\u081a\u081b\u081c\u081d\u081e\u081f\u0820\u0821\u0822\u0823\u0824\u0825\u0826\u0827\u0828\u0829\u082a\u082b\u082c\u082d\u082e\u082f\u0830\u0831\u0832\u0833\u0834\u0835\u0836\u0837\u0838\u0839\u083a\u083b\u083c\u083d\u083e\u083f\u0840\u0841\u0842\u0843\u0844\u0845\u0846\u0847\u0848\u0849\u084a\u084b\u084c\u084d\u084e\u084f\u0850\u0851\u0852\u0853\u0854\u0855\u0856\u0857\u0858\u0859\u085a\u085b\u085c\u085d\u085e\u085f\u0860\u0861\u0862\u0863\u0864\u0865\u0866\u0867\u0868\u0869\u086a\u086b\u086c\u086d\u086e\u086f\u0870\u0871\u0872\u0873\u0874\u0875\u0876\u0877\u0878\u0879\u087a\u087b\u087c\u087d\u087e\u087f\u0880\u0881\u0882\u0883\u0884\u0885\u0886\u0887\u0888\u0889\u088a\u088b\u088c\u088d\u088e\u088f\u0890\u0891\u0892\u0893\u0894\u0895\u0896\u0897\u0898\u0899\u089a\u089b\u089c\u089d\u089e\u089f\u08a0\u08a1\u08a2\u08a3\u08a4\u08a5\u08a6\u08a7\u08a8\u08a9\u08aa\u08ab\u08ac\u08ad\u08ae\u08af\u08b0\u08b1\u08b2\u08b3\u08b4\u08b5\u08b6\u08b7\u08b8\u08b9\u08ba\u08bb\u08bc\u08bd\u08be\u08bf\u08c0\u08c1\u08c2\u08c3\u08c4\u08c5\u08c6\u08c7\u08c8\u08c9\u08ca\u08cb\u08cc\u08cd\u08ce\u08cf\u08d0\u08d1\u08d2\u08d3\u08d4\u08d5\u08d6\u08d7\u08d8\u08d9\u08da\u08db\u08dc\u08dd\u08de\u08df\u08e0\u08e1\u08e2\u08e3\u08e4\u08e5\u08e6\u08e7\u08e8\u08e9\u08ea\u08eb\u08ec\u08ed\u08ee\u08ef\u08f0\u08f1\u08f2\u08f3\u08f4\u08f5\u08f6\u08f7\u08f8\u08f9\u08fa\u08fb\u08fc\u08fd\u08fe\u08ff\u0900\u0901\u0902\u0903\u0904\u0905\u0906\u0907\u0908\u0909\u090a\u090b\u090c\u090d\u090e\u090f\u0910\u0911\u0912\u0913\u0914\u0915\u0916\u0917\u0918\u0919\u091a\u091b\u091c\u091d\u091e\u091f\u0920\u0921\u0922\u0923\u0924\u0925\u0926\u0927\u0928\u0929\u092a\u092b\u092c\u092d\u092e\u092f\u0930\u0931\u0932\u0933\u0934\u0935\u0936\u0937\u0938\u0939\u093a\u093b\u093c\u093d\u093e\u093f\u0940\u0941\u0942\u0943\u0944\u0945\u0946\u0947\u0948\u0949\u094a\u094b\u094c\u094d\u094e\u094f\u0950\u0951\u0952\u0953\u0954\u0955\u0956\u0957\u0958\u0959\u095a\u095b\u095c\u095d\u095e\u095f\u0960\u0961\u0962\u0963\u0964\u0965\u0966\u0967\u0968\u0969\u096a\u096b\u096c\u096d\u096e\u096f\u0970\u0971\u0972\u0973\u0974\u0975\u0976\u0977\u0978\u0979\u097a\u097b\u097c\u097d\u097e\u097f\u0980\u0981\u0982\u0983\u0984\u0985\u0986\u0987\u0988\u0989\u098a\u098b\u098c\u098d\u098e\u098f\u0990\u0991\u0992\u0993\u0994\u0995\u0996\u0997\u0998\u0999\u099a\u099b\u099c\u099d\u099e\u099f\u09a0\u09a1\u09a2\u09a3\u09a4\u09a5\u09a6\u09a7\u09a8\u09a9\u09aa\u09ab\u09ac\u09ad\u09ae\u09af\u09b0\u09b1\u09b2\u09b3\u09b4\u09b5\u09b6\u09b7\u09b8\u09b9\u09ba\u09bb\u09bc\u09bd\u09be\u09bf\u09c0\u09c1\u09c2\u09c3\u09c4\u09c5\u09c6\u09c7\u09c8\u09c9\u09ca\u09cb\u09cc\u09cd\u09ce\u09cf\u09d0\u09d1\u09d2\u09d3\u09d4\u09d5\u09d6\u09d7\u09d8\u09d9\u09da\u09db\u09dc\u09dd\u09de\u09df\u09e0\u09e1\u09e2\u09e3\u09e4\u09e5\u09e6\u09e7\u09e8\u09e9\u09ea\u09eb\u09ec\u09ed\u09ee\u09ef\u09f0\u09f1\u09f2\u09f3\u09f4\u09f5\u09f6\u09f7\u09f8\u09f9\u09fa\u09fb\u09fc\u09fd\u09fe\u09ff\u0a00\u0a01\u0a02\u0a03\u0a04\u0a05\u0a06\u0a07\u0a08\u0a09\u0a0a\u0a0b\u0a0c\u0a0d\u0a0e\u0a0f\u0a10\u0a11\u0a12\u0a13\u0a14\u0a15\u0a16\u0a17\u0a18\u0a19\u0a1a\u0a1b\u0a1c\u0a1d\u0a1e\u0a1f\u0a20\u0a21\u0a22\u0a23\u0a24\u0a25\u0a26\u0a27\u0a28\u0a29\u0a2a\u0a2b\u0a2c\u0a2d\u0a2e\u0a2f\u0a30\u0a31\u0a32\u0a33\u0a34\u0a35\u0a36\u0a37\u0a38\u0a39\u0a3a\u0a3b\u0a3c\u0a3d\u0a3e\u0a3f\u0a40\u0a41\u0a42\u0a43\u0a44\u0a45\u0a46\u0a47\u0a48\u0a49\u0a4a\u0a4b\u0a4c\u0a4d\u0a4e\u0a4f\u0a50\u0a51\u0a52\u0a53\u0a54\u0a55\u0a56\u0a57\u0a58\u0a59\u0a5a\u0a5b\u0a5c\u0a5d\u0a5e\u0a5f\u0a60\u0a61\u0a62\u0a63\u0a64\u0a65\u0a66\u0a67\u0a68\u0a69\u0a6a\u0a6b\u0a6c\u0a6d\u0a6e\u0a6f\u0a70\u0a71\u0a72\u0a73\u0a74\u0a75\u0a76\u0a77\u0a78\u0a79\u0a7a\u0a7b\u0a7c\u0a7d\u0a7e\u0a7f\u0a80\u0a81\u0a82\u0a83\u0a84\u0a85\u0a86\u0a87\u0a88\u0a89\u0a8a\u0a8b\u0a8c\u0a8d\u0a8e\u0a8f\u0a90\u0a91\u0a92\u0a93\u0a94\u0a95\u0a96\u0a97\u0a98\u0a99\u0a9a\u0a9b\u0a9c\u0a9d\u0a9e\u0a9f\u0aa0\u0aa1\u0aa2\u0aa3\u0aa4\u0aa5\u0aa6\u0aa7\u0aa8\u0aa9\u0aaa\u0aab\u0aac\u0aad\u0aae\u0aaf\u0ab0\u0ab1\u0ab2\u0ab3\u0ab4\u0ab5\u0ab6\u0ab7\u0ab8\u0ab9\u0aba\u0abb\u0abc\u0abd\u0abe\u0abf\u0ac0\u0ac1\u0ac2\u0ac3\u0ac4\u0ac5\u0ac6\u0ac7\u0ac8\u0ac9\u0aca\u0acb\u0acc\u0acd\u0ace\u0acf\u0ad0\u0ad1\u0ad2\u0ad3\u0ad4\u0ad5\u0ad6\u0ad7\u0ad8\u0ad9\u0ada\u0adb\u0adc\u0add\u0ade\u0adf\u0ae0\u0ae1\u0ae2\u0ae3\u0ae4\u0ae5\u0ae6\u0ae7\u0ae8\u0ae9\u0aea\u0aeb\u0aec\u0aed\u0aee\u0aef\u0af0\u0af1\u0af2\u0af3\u0af4\u0af5\u0af6\u0af7\u0af8\u0af9\u0afa\u0afb\u0afc\u0afd\u0afe\u0aff\u0b00\u0b01\u0b02\u0b03\u0b04\u0b05\u0b06\u0b07\u0b08\u0b09\u0b0a\u0b0b\u0b0c\u0b0d\u0b0e\u0b0f\u0b10\u0b11\u0b12\u0b13\u0b14\u0b15\u0b16\u0b17\u0b18\u0b19\u0b1a\u0b1b\u0b1c\u0b1d\u0b1e\u0b1f\u0b20\u0b21\u0b22\u0b23\u0b24\u0b25\u0b26\u0b27\u0b28\u0b29\u0b2a\u0b2b\u0b2c\u0b2d\u0b2e\u0b2f\u0b30\u0b31\u0b32\u0b33\u0b34\u0b35\u0b36\u0b37\u0b38\u0b39\u0b3a\u0b3b\u0b3c\u0b3d\u0b3e\u0b3f\u0b40\u0b41\u0b42\u0b43\u0b44\u0b45\u0b46\u0b47\u0b48\u0b49\u0b4a\u0b4b\u0b4c\u0b4d\u0b4e\u0b4f\u0b50\u0b51\u0b52\u0b53\u0b54\u0b55\u0b56\u0b57\u0b58\u0b59\u0b5a\u0b5b\u0b5c\u0b5d\u0b5e\u0b5f\u0b60\u0b61\u0b62\u0b63\u0b64\u0b65\u0b66\u0b67\u0b68\u0b69\u0b6a\u0b6b\u0b6c\u0b6d\u0b6e\u0b6f\u0b70\u0b71\u0b72\u0b73\u0b74\u0b75\u0b76\u0b77\u0b78\u0b79\u0b7a\u0b7b\u0b7c\u0b7d\u0b7e\u0b7f\u0b80\u0b81\u0b82\u0b83\u0b84\u0b85\u0b86\u0b87\u0b88\u0b89\u0b8a\u0b8b\u0b8c\u0b8d\u0b8e\u0b8f\u0b90\u0b91\u0b92\u0b93\u0b94\u0b95\u0b96\u0b97\u0b98\u0b99\u0b9a\u0b9b\u0b9c\u0b9d\u0b9e\u0b9f\u0ba0\u0ba1\u0ba2\u0ba3\u0ba4\u0ba5\u0ba6\u0ba7\u0ba8\u0ba9\u0bba\u0bbb\u0bbc\u0bbd\u0bbe\u0bbf\u0bc0\u0bc1\u0bc2\u0bc3\u0bc4\u0bc5\u0bc6\u0bc7\u0bc8\u0bc9\u0bca\u0acb\u0bcc\u0bcd\u0bce\u0bcf\u0bd0\u0bd1\u0bd2\u0bd3\u0bd4\u0bd5\u0bd6\u0bd7\u0bd8\u0bd9\u0bda\u0bdb\u0bdc\u0bdd\u0bde\u0bdf\u0be0\u0be1\u0be2\u0be3\u0be4\u0be5\u0be6\u0be7\u0be8\u0be9\u0bea\u0beb\u0bec\u0bed\u0bee\u0bef\u0bf0\u0bf1\u0bf2\u0bf3\u0bf4\u0bf5\u0bf6\u0bf7\u0bf8\u0bf9\u0bfa\u0bfb\u0bfc\u0bfd\u0bfe\u0bff\u0c00\u0c01\u0c02\u0c03\u0c04\u0c05\u0c06\u0c07\u0c08\u0c09\u0c0a\u0c0b\u0c0c\u0c0d\u0c0e\u0c0f\u0c10\u0c11\u0c12\u0c13\u0c14\u0c15\u0c16\u0c17\u0c18\u0c19\u0c1a\u0c1b\u0c1c\u0c1d\u0c1e\u0c1f\u0c20\u0c21\u0c22\u0c23\u0c24\u0c25\u0c26\u0c27\u0c28\u0c29\u0c2a\u0c2b\u0c2c\u0c2d\u0c2e\u0c2f\u0c30\u0c31\u0c32\u0c33\u0c34\u0c35\u0c36\u0c37\u0c38\u0c39\u0c3a\u0c3b\u0c3c\u0c3d\u0c3e\u0c3f\u0c40\u0c41\u0c42\u0c43\u0c44\u0c45\u0c46\u0c47\u0c48\u0c49\u0c4a\u0c4b\u0c4c\u0c4d\u0c4e\u0c4f\u0c50\u0c51\u0c52\u0c53\u0c54\u0c55\u0c56\u0c57\u0c58\u0c59\u0c5a\u0c5b\u0c5c\u0c5d\u0c5e\u0c5f\u0c60\u0c61\u0c62\u0c63\u0c64\u0c65\u0c66\u0c67\u0c68\u0c69\u0c6a\u0c6b\u0c6c\u0c6d\u0c6e\u0c6f\u0c70\u0c71\u0c72\u0c73\u0c74\u0c75\u0c76\u0c77\u0c78\u0c79\u0c7a\u0c7b\u0c7c\u0c7d\u0c7e\u0c7f\u0c80\u0c81\u0c82\u0c83\u0c84\u0c85\u0c86\u0c87\u0c88\u0c89\u0c8a\u0c8b\u0c8c\u0c8d\u0c8e\u0c8f\u0c90\u0c91\u0c92\u0c93\u0c94\u0c95\u0c96\u0c97\u0c98\u0c99\u0c9a\u0c9b\u0c9c\u0c9d\u0c9e\u0c9f\u0ca0\u0ca1\u0ca2\u0ca3\u0ca4\u0ca5\u0ca6\u0ca7\u0ca8\u0ca9\u0caa\u0cab\u0cac\u0cad\u0cae\u0caf\u0cb0\u0cb1\u0cb2\u0cb3\u0cb4\u0cb5\u0cb6\u0cb7\u0cb8\u0cb9\u0cba\u0cbb\u0cbc\u0cbd\u0cbe\u0cbf\u0cc0\u0cc1\u0cc2\u0cc3\u0cc4\u0cc5\u0cc6\u0cc7\u0cc8\u0cc9\u0cca\u0ccb\u0ccc\u0ccd\u0cce\u0ccf\u0cd0\u0cd1\u0cd2\u0cd3\u0cd4\u0cd5\u0cd6\u0cd7\u0cd8\u0cd9\u0cda\u0cdb\u0cdc\u0cdd\u0cde\u0cdf\u0ce0\u0ce1\u0ce2\u0ce3\u0ce4\u0ce5\u0ce6\u0ce7\u0ce8\u0ce9\u0cea\u0ceb\u0cec\u0ced\u0cee\u0cef\u0cf0\u0cf1\u0cf2\u0cf3\u0cf4\u0cf5\u0cf6\u0cf7\u0cf8\u0cf9\u0cfa\u0cfb\u0cfc\u0cfd\u0cfe\u0cff\u0d00\u0d01\u0d02\u0d03\u0d04\u0d05\u0d06\u0d07\u0d08\u0d09\u0d0a\u0d0b\u0d0c\u0d0d\u0d0e\u0d0f\u0d10\u0d11\u0d12\u0d13\u0d14\u0d15\u0d16\u0d17\u0d18\u0d19\u0d1a\u0d1b\u0d1c\u0d1d\u0d1e\u0d1f\u0d20\u0d21\u0d22\u0d23\u0d24\u0d25\u0d26\u0d27\u0d28\u0d29\u0d2a\u0d2b\u0d2c\u0d2d\u0d2e\u0d2f\u0d30\u0d31\u0d32\u0d33\u0d34\u0d35\u0d36\u0d37\u0d38\u0d39\u0d3a\u0d3b\u0d3c\u0d3d\u0d3e\u0d3f\u0d40\u0d41\u0d42\u0d43\u0d44\u0d45\u0d46\u0d47\u0d48\u0d49\u0d4a\u0d4b\u0d4c\u0d4d\u0d4e\u0d4f\u0d50\u0d51\u0d52\u0d53\u0d54\u0d55\u0d56\u0d57\u0d58\u0d59\u0d5a\u0d5b\u0d5c\u0d5d\u0d5e\u0d5f\u0d60\u0d61\u0d62\u0d63\u0d64\u0d65\u0d66\u0d67\u0d68\u0d69\u0d6a\u0d6b\u0d6c\u0d6d\u0d6e\u0d6f\u0d70\u0d71\u0d72\u0d73\u0d74\u0d75\u0d76\u0d77\u0d78\u0d79\u0d7a\u0d7b\u0d7c\u0d7d\u0d7e\u0d7f\u0d80\u0d81\u0d82\u0d83\u0d84\u0d85\u0d86\u0d87\u0d88\u0d89\u0d8a\u0d8b\u0d8c\u0d8d\u0d8e\u0d8f\u0d90\u0d91\u0d92\u0d93\u0d94\u0d95\u0d96\u0d97\u0d98\u0d99\u0d9a\u0d9b\u0d9c\u0d9d\u0d9e\u0d9f\u0da0\u0da1\u0da2\u0da3\u0da4\u0da5\u0da6\u0da7\u0
```


щаем в памяти программы и захватываем область памяти, равную по объему всему содержимому ресурса. Такой подход сводит на нет все преимущества потока, способного выдавать данные мелкими порциями по мере надобности. В качестве эксперимента помещаем в ресурс небольшой рассказик объемом 20 килобайтов и пробуем, что из этого получится. Стандартный эмулятор после двухминутных раздумий все же справился с задачей (поверьте на слово или проверьте, если хотите), а эмулятор от Nokia опять сошел с дистанции, выдав вот такое милое сообщение (рис. 9.4).

Класс BookReader

Итак, пишем приложение для удобного чтения литературы с экрана телефона. Для начала хотелось бы уменьшить шрифт, выводимый на экране. Смену шрифта поддерживает класс Graphics, содержащий метод `setFont(Font font)`. В примерах мы пользовались объектом класса `Font`, который не поддерживает прорисовку объектов класса `Graphics`. В нашем случае воспользуемся отображаемым классом `Canvas`, который мы уже рассматривали в одной из первых глав.

Остановимся на классе `Font`, который и представляет отображаемый шрифт. Получить новый шрифт можно с помощью метода `Font.getFont(int face, int style, int size)`, параметры которого определяют, как будут выглядеть символы:

- `face` — вид шрифта определяется константами `FACE_SYSTEM`, `FACE_MONOSPACE` или `FACE_PROPORTIONAL`;
- `style` — стиль шрифта задается константой `STYLE_PLAIN` или комбинацией констант `STYLE_BOLD`, `STYLE_ITALIC` и `STYLE_UNDERLINED`;
- `size` — размер шрифта задается одной из констант `SIZE_SMALL`, `SIZE_MEDIUM` или `SIZE_LARGE`.

Если в классе `Form` форматирование текста по размеру экрана выполнялось автоматически, то сейчас вся ответственность за корректное позиционирование текста переходит к нам. Класс `Font` содержит ряд методов для определения размера текста в пикселах:

- `int getHeight()` — возвращает высоту строки шрифта в пикселах;
- `int charWidth(char ch)` — возвращает ширину символа `ch` в пикселах;
- `int charsWidth(char[] ch, int offset, int length)` — возвращает ширину набора символов, содержащихся в массиве `ch`, начиная с позиции `offset` длины `length`;
- `int stringWidth(String str)` — возвращает ширину строки `str` в пикселах;
- `int substringWidth(String str, int offset, int len)` — возвращает ширину фрагмента строки `str`, начинающегося с позиции `offset` и имеющего длину `len`.

В классе `BookCanvas` реализуем следующую логику отображения текста. В стеке `PageIndex` будем хранить смещение от начала текста для каждой страницы, чтобы организовать возможность обратной прокрутки текста. Функция `paint` читает из потока данных по одному слову и отображает его, если на экране достаточно пространства. Клавиша 2 осуществляет переход к следующей странице

текста, клавиша 1 — к предыдущей странице, а клавиша 0 возвращает к началу текста:

```
import java.io.InputStream;
import java.io.EOFException;
import java.io.IOException;
import java.util.Stack;

public class BookCanvas extends Canvas {

    private Stack PageIndex; // стек смещений страниц от начала текста

    public BookCanvas() {
        PageIndex = new Stack();
    }

    protected void paint(Graphics g) {
        // получить ширину и высоту рабочей области экрана
        int gw = g.getClipWidth();
        int gh = g.getClipHeight();

        // создать шрифт
        Font font = Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_PLAIN, Font.SIZE_SMALL);
        // установить шрифт
        g.setFont(font);

        // инициализировать поток
        InputStream is = getClass().getResourceAsStream("/story.txt");
        MyDataInputStream mdis = new MyDataInputStream(is);

        int x=0, y=0; // текущее положение вывода на экране
        int offset=0; // смещение по исходному тексту

        // очистка экрана
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, gw, gh);
        g.setColor(0, 0, 0);

        // получить смещение для очередной страницы
        if(!PageIndex.empty())
            offset=((Integer)PageIndex.peek()).intValue();

        try{
            // сместиться по тексту до необходимой страницы
```

```

    mdis.skipBytes(offset);
} catch(IOException ioe) {}

String sWord;
do {
    // прочитать очередное слово из потока
    sWord = mdis.readWord();
    // продвинуть смещение по файлу на одно слово

    offset+=sWord.length();
    // если текущая позиция и пиксельная ширина слова
    // не выходят за экран
    if(x+font.stringWidth(sWord)<=gw) {
        // отобразить слово, начиная с текущей позиции
        g.drawString(sWord, x.y, g.TOP|g.LEFT);
        // сместить текущую позицию на пиксельную ширину слова
        x+=font.stringWidth(sWord);
        // если слово – последнее в строке, перевести текущую позицию
        // в начало новой строки на дисплее
        if (mdis.b_endLine) { y+=font.getHeight(); x=0; }
    }
    // слово не входит по ширине с текущей позиции
    else {
        // перейти на новую строку
        y+=font.getHeight();
        // если на экране есть место для новой строки,
        // отобразить слово с начала строки
        if(y+font.getHeight()<=gh) g.drawString
            (sWord, 0.y, g.TOP|g.LEFT);
        // перевести текущую позицию на ширину слова
        x=font.stringWidth(sWord);
    }
    // повторять до тех пор, пока есть место для новой строки
    // или не кончится файл
} while(y+font.getHeight()<=gh && !mdis.b_endFile);

int index;    // смещение начала страницы
// если цикл закончился чтением слова,
// которое не было отображено на экране.
// сместить начало новой страницы назад на одно слово
if(mdis.b_endLine) index=offset; else index=offset-sWord.length();
// поместить в стек начало следующей страницы
if(!mdis.b_endFile) PageIndex.push(new Integer(index));
// закрыть поток
try{
    mdis.close();
}

```



```

    } catch(IOException ioe) {}
}

public void keyPressed(int keyCode){
    // обработка нажатий клавиш
    switch(keyCode) {
        case KEY_NUM1:
            // удалить из стека начала следующей и текущей страницы
            PageIndex.pop();
            if(!PageIndex.empty()) PageIndex.pop();
            repaint();
            break;
        case KEY_NUM2:
            // перерисовать экран (начало следующей страницы уже в стеке)
            repaint();
            break;
        case KEY_NUM0:
            // очистить стек и перерисовать экран
            PageIndex.removeAllElements();
            repaint();
            break;
    }
}
}

```

Поток в данном случае представлен классом, наследованным из `DataInputStream` и реализующим функцию чтения слова `readWord()`. При чтении очередного слова функция выставляет флаги конца строки и конца файла. Сам класс выглядит следующим образом:

```

import java.io.InputStream;
import java.io.DataInputStream;
import java.io.EOFException;
import java.io.IOException;

public class MyDataInputStream extends DataInputStream {

    private String WIN1251_TO_UNICODE = "..."; // кодовая строка
                                                // преобразования символов

    public boolean b_endLine = false;           // флаг конца строки
    public boolean b_endFile = false;           // флаг конца файла

    public MyDataInputStream(InputStream is) {
        super(is);
    }

    public String readWord() {
        int i=0; // счетчик символов в текущем слове
    }
}

```

```

b_endFile = false;
char[] word = new char[100]; // массив для чтения слова
try {
    // читать слово посимвольно до пробела или перевода строки
    do word[i++]=(char)convert(readUnsignedByte());
    while (word[i-1]!='32' && word[i-1]!='10');
} catch (EOFException ioe) {i--; b_endFile = true;}
catch (IOException ioe)

if (word[i-1]=='10') b_endLine=true; else b_endLine=false;
return (new String(word,0,i));
}

char convert(int ch) {
    return (ch < 128) ? (char)ch : WIN1251_TO_UNICODE.charAt(ch-128);
}
}

```

При чтении данных потока могут быть сформированы следующие исключения:

- EOFException — формируется при достижении конца файла;
- IOException — базовое исключение для всех функций ввода-вывода.

В программе мидлета осталось лишь создать объект класса BookCanvas и установить его в качестве текущего дисплея, как мы уже неоднократно делали. Запускаем получившееся приложение на эмуляторе и получаем в пользование вот такую замечательную читалку (рис. 9.5).

Совершенству, как известно, предела нет, поэтому некоторые моменты остались на самостоятельную доработку. Непроработана ситуация, когда длина слова превышает размер экрана — здесь нужна логика деления слова на части и переноса на следующую строку. При чтении последней страницы очередное смещение в стек не заносится, поэтому при прокрутке текста назад с конечной позиции одна страница будет пропущена. Кроме этого, текст не должен содержать в конце пробелов или переводов строки: это может стать причиной формирования исключения при просмотре последней страницы. Еще можно поработать над более удобным, интуитивным перемещением по тексту, задействовав не цифры, а навигационные клавиши аппарата, помеченные стрелками. Для разных моделей телефонов эти клавиши формируют разные коды,



Рис. 9.5. «Алло! Мистер Шухарт?»

поэтому лучше пользоваться API-константами конкретной модели. На API-функциях некоторых моделей мы остановимся позже.

Класс `ByteArrayInputStream`

На примерах мы рассмотрели работу с файлом-ресурсом, который предоставлял приложению все необходимые данные. На самом деле, поток может быть сформирован из любых данных и представлен в виде последовательности байтов. Еще один класс, который порожден из базового класса `InputStream` и представляет поток в виде открытого массива байтов, — это `ByteArrayInputStream`.

Данный класс предоставляет все объявленные, но не реализованные функции класса `InputStream`, позволяя получить заранее число доступных байтов потока, пометить позицию возврата для повторного чтения данных.

Внутренняя структура потока открыта для нас. Программисту доступны следующие поля:

- `byte[] buf` — массив байтов для хранения данных, указанный при создании потока;
- `int count` — число доступных байтов потока;
- `int mark` — индекс текущей помеченной позиции для повторного чтения;
- `int pos` — индекс следующего байта потока для чтения.

Использование объектов класса `ByteArrayInputStream` мы рассмотрим в следующей главе.

Класс `OutputStream`

До сих пор мы работали с потоком только одного направления — от ресурса к приложению. Все рассмотренные классы, так или иначе, связаны со словом `Input` (ввод), а методы чтения данных — со словом `read`. Вся система ввода зеркально отображается на работу с потоком обратного направления. Поток вывода данных сохраняет всю иерархию классов потока ввода, с той лишь разницей, что в названиях классов вместо слова `Input` фигурирует слово `Output`: `OutputStream`, `ByteArrayOutputStream`, `DataOutputStream`. Каждому методу чтения, содержащему в названии слово `read`, соответствует аналогичный метод записи, только со словом `write`.

Функции потоков чтения `skip`, `available`, `mark` и `reset` для потоков вывода не актуальны, вместо них потоки вывода реализуют лишь одну функцию `flush()`, которая очищает все данные выходного потока.

* * *

Это были простейшие примеры использования потоков данных. Но кроме ресурсов приложения существуют также и сетевые ресурсы, ведь мобильный телефон это все-таки средство связи. Организация сетей, поддержка работы с сетевыми ресурсами, коммуникация между мидлетами и создание межсетевых приложений реализуются с помощью потоков данных. На некоторых примерах использования потоков для сетевых данных мы остановимся в заключительных главах, поэтому оставайтесь с нами: впереди еще много интересного.

Глава 10

Ввод текста, работа со строками

В этой главе мы рассмотрим стандартные средства связи с пользователем, позволяющие не только выбрать один из предложенных пунктов меню, но и вводить с клавиатуры телефона произвольный текст, и реализуем долговременное хранение введенных данных, используя технологии, рассмотренные нами в прошлых главах, такие как потоки данных и RMS.

В наш век научно-технического прогресса сотовый телефон практически всегда под рукой, в отличие от обычной авторучки. Часто ли вы сталкивались с ситуацией, когда нужно записать какую-то важную информацию: почтовый адрес, e-mail, названия лекарств, маркировку деталей, да мало ли что? Стандартная записная книжка мобильного телефона рассчитана лишь на запись телефонного номера и не очень заковыристых имени и фамилии его обладателя. Можно, конечно, использовать не по назначению ремайндер, если он есть, или писать текст в теле сообщения SMS, но в таком случае потерять информацию так же легко, как и случайный клочок бумаги. Мы же попробуем сейчас написать свою адресную книгу, расширив количество хранимой информации согласно нашим требованиям.

Стандартные средства предлагают нам два варианта для реализации полей ввода текста: `TextField` и `TextBox`. Рассмотрим подробно каждый из них.

Класс `TextBox`

Класс `TextBox` является потомком класса `Screen` и представляет собой экран, позволяющий вводить и редактировать текст с клавиатуры аппарата. При создании объекта в конструкторе нужно указать заголовок поля ввода, начальное содержимое поля, максимально допустимое количество вводимых символов и ограничитель для вводимой информации:

```
TextBox(String title, String text, int maxSize, int constraints)
```

Ограничители позволяют заранее задать тип вводимого текста. Так, например, если информация содержит только цифры, то буквы не будут появляться в этом поле при нажатии на клавиши. Для полей ввода доступны следующие ограничители, которые определяются константами класса `TextField`:

- `ANY` — допускается ввод любых символов без ограничений;
- `EMAILADDR` — используется для ввода адреса электронной почты;
- `NUMERIC` — применяется для ввода целых чисел. Клавиша `*` позволяет вводить отрицательные числа и добавляет знак `-` перед числом;

- **PHONENUMBER** — ввод телефонного номера. Поле ввода такого типа привязано к стандартному вводу телефонного номера аппарата. Это значит, что в меню автоматически появится команда поиска номера в записной книжке **Search**;
- **PASSWORD** — используется в комбинации с другими константами для скрытия вводимой информации. Вводимые символы отображаются звездочками, как при вводе PIN-кода. Комбинации констант задаются с помощью оператора **|** (или), например: **NUMERIC | PASSWORD**;
- **URL** — ввод сетевого адреса интернет-страницы в формате **URL**;
- **CONSTRAINT_MASK** — маска ограничителя; константа, используемая для получения текущего ограничителя поля из значения, возвращенного методом **GetConstraints()**. Маска накладывается с помощью оператора **&** (и) на возвращенное значение и отбрасывает флаги модификаторов ограничителя, например, такие как **PASSWORD**.

После того как объект поля ввода создан, он может быть отображен на экране телефона так же, как и все отображаемые объекты, наследованные из класса **Displayable**, с помощью метода класса **Display** **setCurrent(TextBox)**. Если вводимый текст превышает размер экрана, система сама обеспечит прокрутку без каких-либо усилий с нашей стороны. Дальнейшая работа с созданным объектом обеспечивается следующими методами класса **TextBox**:

- **void delete(int offset, int length)** — удаляет из поля ввода **length** символов, начиная с позиции **offset**;
- **int getCaretPosition()** — возвращает текущее положение позиции ввода (0 — начало текста);
- **int getChars(char[] data)** — копирует весь текст из поля ввода в символьный массив **data**, возвращает число скопированных символов. Текст, отображаемый в поле ввода, не изменяется;
- **int getConstraints()** — возвращает текущий ограничитель поля ввода;
- **int getMaxSize()** — возвращает максимально допустимое количество вводимых символов;
- **String getString()** — возвращает содержимое поля ввода в виде строки;
- **void insert(String src, int position)** — вставляет строку **src** в поле ввода, начиная с позиции **position**. Если указанная позиция меньше или равна нулю, то строка вставляется в начало поля, если позиция больше количества уже введенных символов, то строка вставляется сразу за последним символом;
- **void insert(char[] data, int offset, int length, int position)** — вставляет в поле ввода с позиции, заданной переменной **position**, **length** символов из массива **data**, начиная с символа с индексом **offset**. Логика вставки, как в предыдущем методе;
- **void setChars(char[] data, int offset, int length)** — очищает поле ввода и вставляет **length** символов из массива **data**, начиная с позиции массива **offset**;
- **void setConstraints(int constraints)** — устанавливает ограничители поля ввода;
- **int setMaxSize(int maxSize)** — устанавливает максимально допустимое количество вводимых символов;
- **void setString(String text)** — заменяет текущее содержимое поля ввода на строку, заданную параметром **text**;
- **int size()** — возвращает количество символов, содержащееся в поле ввода.

Класс TextField

Объект класса TextField представляет собой редактируемую строку, отображаемую на экране телефона и позволяющую пользователю редактировать и вводить текстовые символы с клавиатуры аппарата.

Разница между объектами TextBox и TextField такая же, как и между рассмотренными ранее объектами List и ChoiceGroup. Они реализуют один и тот же набор методов, а различаются только тем, что первый является самостоятельным отображаемым объектом иерархии класса Displayable, а второй для демонстрации должен быть включен в форму.

Класс TextField расширяет рассмотренный нами в программе слайд-шоу класс Item, а это значит, что объект класса TextField можно включить в обычную форму (class Form) с помощью стандартного метода формы append.

Для примера рассмотрим программу калькулятора, выполняющего простые арифметические действия. В калькуляторе будут использоваться три поля ввода: два для аргументов и одно для символа арифметического действия:

```
...
import javax.microedition.lcdui.TextField;
...
private Display display;
private Form form;
private Command equal;
private MyCommandListener cl = new MyCommandListener();
private TextField arg1,arg2,action;    // поля ввода
...
public void startApp() {

    // создать форму калькулятора
    display = Display.getDisplay(this);
    form = new Form("Calculator");
    form.setCommandListener(cl);

    // команда, запускающая вычисления
    equal = new Command("Equal", Command.OK, 1);
    form.addCommand(equal);

    // поле ввода первого аргумента
    arg1 = new TextField("Argument1","",5,TextField.NUMERIC);
    form.append(arg1);

    // поле ввода символа операции
    action = new TextField("Action","",1,TextField.ANY);
    form.append(action);

    // поле ввода второго аргумента
    arg2 = new TextField("Argument2","",5,TextField.NUMERIC);
```



```

    form.append(arg2);
    display.setCurrent(form);
}

private class MyCommandListener implements CommandListener
{
    public void commandAction(Command c, Displayable d)
    {
        if(c==equal) { // запустить вычисление
            // форма для экрана результата операции
            Form resForm = new Form("Result");
            resForm.setCommandListener(c);
            Command back = new Command("Back",Command.BACK,1);
            resForm.addCommand(back);

            // проверить заполнение всех полей ввода
            if(arg1.size()==0 || arg2.size()==0 || action.size()==0) {
                // ошибка, не все поля ввода заполнены
                resForm.append("Missing argument");
                // отобразить сообщение
                display.setCurrent(resForm);
                // выйти
                return;
            }

            // получить символ операции
            char[] act = new char[1];
            action.getChars(act);

            // преобразование аргументов
            int first = Integer.parseInt(arg1.getString());
            int second = Integer.parseInt(arg2.getString());
            int res = 0;
            switch(act[0]) {
                // выполнить действие в соответствии
                // с символом операции
                case '+':
                    res = first + second;
                    break;
                case '-':
                    res = first - second;
                    break;
                case '*':
                    res = first * second;
                    break;
            }
        }
    }
}

```

```
case '/':
    res = first / second;
    break;
default:
    // ошибка, некорректный символ операции
    resForm.append("Illegal Operation");
    // отобразить сообщение
    display.setCurrent(resForm);
    // выйти
    return;
}

// преобразовать результат в строку и добавить в форму
resForm.append((new Integer(res)).toString());
// отобразить результат
display.setCurrent(resForm);
} else
    // команда Back формы результата приводит сюда
    // отобразить первоначальную форму
    display.setCurrent(form);
}
```

Вид и размещение полей ввода на экране зависит от конкретной модели аппарата. На стандартном эмуляторе наш калькулятор будет выглядеть следующим образом (рис. 10.1).

Казалось бы, можно радоваться, поскольку мы получили бесплатный калькулятор, который отсутствует во многих моделях. Главная проблема состоит в том, что виртуальная машина CDLC не поддерживает вычислений с плавающей точкой. Другими словами, наш калькулятор сможет работать только с целыми числами, отбрасывая при делении дробную часть. К сожалению, такой калькулятор сгодится лишь для подсчета сдачи в магазине.

Класс AddressBook

В прошлых главах мы рассмотрели методы хранения информации в памяти телефона, создание меню, а также потоковую передачу данных с использованием потоков ввода и вывода. Сейчас нам понадобятся все эти знания, поскольку тот пример, который мы собираемся реализовать, будет хранить данные с помощью RMS и передавать их с помощью потоков.

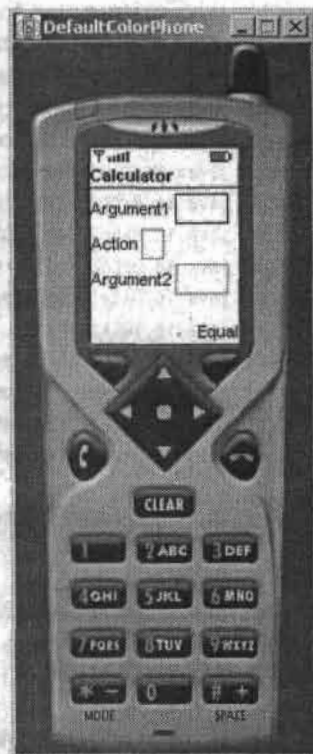


Рис. 10.1. Так выглядит простейший калькулятор

Порой мы недовольны стандартной записной книжкой телефона, которая, конечно же, очень классная, только в ней для полного счастья не хватает какой-то очень важной функции. Выход есть, ведь можно реализовать свою записную книжку, где учесть все свои прихоти, а при надобности добавить еще и чужие. Этим мы сейчас и займемся: реализуем основу для записной книжки, которая будет хранить записи, содержащие имя, номер телефона и адрес электронной почты.

Для начала продумаем сценарий смены экранов. Приложение стартует с экрана со списком всех имен, уже добавленных в записную книжку. Команда Ok демонстрирует форму, содержащую все параметры записи, соответствующей выбранному имени, а команда Add начинает выполнение сценария добавления новой записи. Сценарий добавления записи предоставляет по очереди несколько экранов для ввода каждого из параметров, команда Next выполняет переход к следующему экрану, а команда Back — возврат к предыдущему.

Рассмотрим основной класс мидлета AddressBook, который в качестве полей содержит экран списка имен, экраны сценария добавления записи, а также команды перехода между экранами. Стартовый метод startApp() инициализирует экраны и команды и создает список имен из уже имеющихся записей:

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.TextField;
import javax.microedition.lcdui.TextBox;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.IOException;
import javax.microedition.rms.RecordStore;
import javax.microedition.rms.RecordEnumeration;
import javax.microedition.rms.RecordComparator;
import javax.microedition.rms.RecordStoreException;

public class AddressBook extends MIDlet implements CommandListener
{
    private Display display;           // менеджер дисплея
    private RecordStore recordStore;   // хранилище записей
    private List nameList;             // список имен
    private int recIndexes[];          // массив ID записей.
                                      // соответствующий списку имен
    private Command add,ok,next,back;  // команды переходов между экранами
    private TextBox tbName,tbPhone,tbEMail; // экраны ввода параметров записи
```

```

public void startApp() {
    try {
        // открыть хранилище записей с именем "Address-Book"
        recordStore = RecordStore.openRecordStore("Address-Book", true);
    } catch (RecordStoreException rse) {}

    // получить ссылку на менеджер дисплея
    display = Display.getDisplay(this);

    // создание объектов команд переходов
    ok = new Command("Ok", Command.OK, 1);
    add = new Command("Add", Command.BACK, 1);
    next = new Command("Next", Command.OK, 1);
    back = new Command("Back", Command.BACK, 1);

    // поле ввода имени
    tbName = new TextBox("Name:", "", 15, TextField.ANY);
    tbName.addCommand(next);
    tbName.addCommand(back);
    tbName.setCommandListener(this);

    // поле ввода номера телефона
    tbPhone = new TextBox("Number:", "", 15, TextField.PHONENUMBER);
    tbPhone.addCommand(next);
    tbPhone.addCommand(back);
    tbPhone.setCommandListener(this);

    // поле ввода электронной почты
    tbEMail = new TextBox("E-Mail:", "", 35, TextField.EMAILADDR);
    tbEMail.addCommand(next);
    tbEMail.addCommand(back);
    tbEMail.setCommandListener(this);

    // создать список имен
    BuildNameList();
    // отобразить список имен на экране
    display.setCurrent(nameList);
}

```

Обратим внимание, что если в предыдущей программе для ввода информации мы использовали несколько объектов класса `TextField`, которые добавляли в одну форму, то теперь каждое поле ввода представляет собой отдельный экран класса `TextBox`. Каждый экран содержит команды `Next` и `Back`, которые осуществляют переходы от одного экрана ввода к другому.

Создание списка имен реализовано в методе `BuildNameList()`, который создает и инициализирует новый объект списка, затем получает все записи хранилища, отсортированные по именам в алфавитном порядке. Имена добавляются в список

имен, а ID соответствующих записей — в массив `recIndexes`, чтобы при необходимости можно было легко получить все остальные параметры необходимой записи:

// метод создает список имен из адресной книги

```
private void BuildNameList() {
    // создать объект списка
    nameList = new List("Address-Book", List.IMPLICIT);
    nameList.SetCommandListener(this);
    // команда добавления записи
    nameList.AddCommand(add);
    // команда получения параметров записи
    nameList.AddCommand(ok);
    try {
        // получить количество записей
        int size = recordStore.getNumRecords();
        // создать массив для хранения ID записей
        recIndexes = new int[size];
        // создать объект компаратора в алфавитном порядке
        AlphabeticalOrdering comparator = new AlphabeticalOrdering();
        // получить список записей хранилища
        RecordEnumeration re = recordStore.enumerateRecords(
            (null, comparator, false);
        // индекс массива хранения ID записей
        int i=0;
        // бесконечный цикл: выход из цикла происходит
        // по формированию исключения получения ID следующей записи.
        // после того как было получено ID последней записи
        while(true) {
            // получить ID следующей записи
            int id = re.nextRecordId();
            // записать ID в массив
            recIndexes[i++] = id;
            // получить запись по ID
            byte[] record = recordStore.getRecord(id);
            // преобразовать запись в байтовый поток
            ByteArrayInputStream bais = new ByteArrayInputStream(record);
            // создать поток, поддерживающий чтение по типу
            DataInputStream dis = new DataInputStream(bais);
            // считать из потока первую строку и добавить в список
            nameList.append(dis.readUTF(), null);
        }
    }
    catch (RecordStoreException rse) {}
    catch (IOException ioe) {}
}
```

Алфавитный порядок добавления записей в список имен реализован с помощью компаратора, представленного классом `AlphabeticalOrdering`:

// класс компаратора записей по алфавиту

```
private class AlphabeticalOrdering implements RecordComparator {
```

```
    // метод сравнения записей
```

```
    public int compare(byte[] rec1, byte[] rec2) {
```

```
        // преобразовать записи в байтовый поток
```

```
        ByteArrayInputStream bais1 = new ByteArrayInputStream(rec1);
```

```
        ByteArrayInputStream bais2 = new ByteArrayInputStream(rec2);
```

```
        // создать потоки, поддерживающие чтение по типу
```

```
        DataInputStream dis1 = new DataInputStream(bais1);
```

```
        DataInputStream dis2 = new DataInputStream(bais2);
```

```
        // строки имен
```

```
        String name1 = null;
```

```
        String name2 = null;
```

```
        try {
```

```
            // считать строки с именами
```

```
            name1 = dis1.readUTF ();
```

```
            name2 = dis2.readUTF ();
```

```
        }
```

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

```
        // лексикографическое сравнение строк
```

```
        int result = name1.compareTo(name2);
```

```
        if (result < 0)
```

```
            // первая запись предшествует второй
```

```
            return RecordComparator.PRECEDES;
```

```
        else
```

```
            if (result == 0)
```

```
                // записи содержат идентичные имена
```

```
                return RecordComparator.EQUIVALENT;
```

```
            else
```

```
                // вторая запись предшествует первой
```

```
                return RecordComparator.FOLLOWS;
```

```
    }
```

Таким образом, остается реализовать только блок прослушивания команд, который определяет действия для осуществления переходов между экранами. Если же создать пустой метод `commandAction`, то приложение уже можно компилировать. Запустив приложение, при наличии записей, мы могли бы увидеть список имен, по которому можно перемещаться с помощью навигационных клавиш (рис. 10.2).

Осталось сделать последний шаг и написать блок прослушивания команд, который будет контролировать

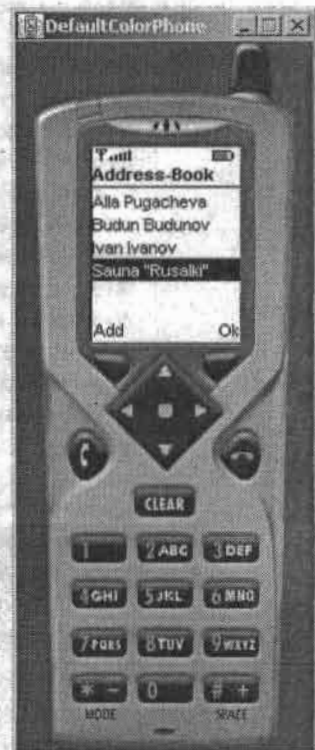


Рис. 10.2. Если встроенная записная книжка вас не устраивает, можно создать свою

порядок отображения экранов в зависимости от выбранной команды, а также реализует сценарий добавления новой записи:

```
// блок прослушивания команд
public void commandAction(Command c, Displayable d) {

    // команда Add отображает экран ввода имени
    if(c==add) display.setCurrent(tbName);

    // команда Ok
    if(c==ok) {
        // если команда вызвана из экрана со списком имен
        if(d==nameList) {
            try {
                // получить в массиве recIndexes ID необходимой записи
                int id = recIndexes[nameList.getSelectedIndex()];
                // считать необходимую запись
                byte[] record = recordStore.getRecord(id);
                // преобразовать запись в байтовый поток
                ByteArrayInputStream bais =
                    new ByteArrayInputStream(record);
                // создать поток, поддерживающий чтение по типу
                DataInputStream dis = new DataInputStream(bais);
                // создать форму для отображения параметров
                Form infoForm = new Form("");
                // считать строки с параметрами и добавить в форму
                infoForm.append(dis.readUTF()+"\n");
                infoForm.append(dis.readUTF()+"\n");
                infoForm.append(dis.readUTF()+"\n");
                // добавить команду возврата к списку имен
                infoForm.addCommand(ok);
                infoForm.setCommandListener(this);
                // отобразить форму на экране
                display.setCurrent(infoForm);
            }
            catch(RecordStoreException rse) {}
            catch(IOException ioe) {}
        } else { // если команда вызвана из формы параметров записи.
            // вернуться к списку имен
            display.setCurrent(nameList);
        }
    }

    // команда Next: перейти к следующему полю ввода
    if(c==next) {
        // из поля ввода имени к полю ввода номера телефона
        if(d==tbName) display.setCurrent(tbPhone);
    }
}
```



```

// из поля ввода номера телефона к полю ввода e-mail
if(d==tbPhone) display.setCurrent(tbEMail);
// из поля ввода e-mail
if(d==tbEMail) {
    // создать байтовый поток вывода
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    // создать поток вывода, поддерживающий запись по типу
    DataOutputStream dos = new DataOutputStream(baos);
    try {
        // записать введенные параметры в поток вывода
        dos.writeUTF(tbName.getString());
        dos.writeUTF(tbPhone.getString());
        dos.writeUTF(tbEMail.getString());
        // добавить запись в хранилище
        recordStore.addRecord(baos.toByteArray(), 0, baos.size());
    }
    catch(IOException ioe) {}
    catch (RecordStoreException rse) {}
    // создать список имен
    BuildNameList();
    // отобразить список имен на экране
    display.setCurrent(nameList);
}
}

// команда Back: возврат к предыдущему полю ввода
if(c==back) {
    // из поля ввода имени к списку имен
    if(d==tbName) display.setCurrent(nameList);
    // из поля ввода номера телефона к полю ввода имени
    if(d==tbPhone) display.setCurrent(tbName);
    // из поля ввода e-mail к полю ввода номера телефона
    if(d==tbEMail) display.setCurrent(tbPhone);
}
}

```

На этом пока остановимся. Теперь все команды должны работать, можно запускать и пробовать. Экран ввода параметров записи будет выглядеть следующим образом (рис. 10.3).

После того как мы ввели параметры записи и добавили ее в хранилище, введенное имя записи появится в списке. Можно выбрать его, используя навигационные клавиши, и с помощью команды Ok вывести на экран параметры записи (рис. 10.4).

Основное направление движения задано. Дальше вы можете продолжить самостоятельно. Первым делом нужно реализовать поиск записи по имени или его фрагменту с помощью фильтра записей хранилища. Не лишней при добавлении новой записи будет проверка, существует ли уже запись с подобным именем. Такую проверку можно организовать с помощью того же фильтра. Я уверен, что богатая фан-

тазия программиста подскажет вам еще массу приятных и полезных нововведений. Одно из возможных усовершенствований мы подробно рассмотрим в следующей главе. Я думаю, что такого вы еще не видели ни в одной модели телефона.

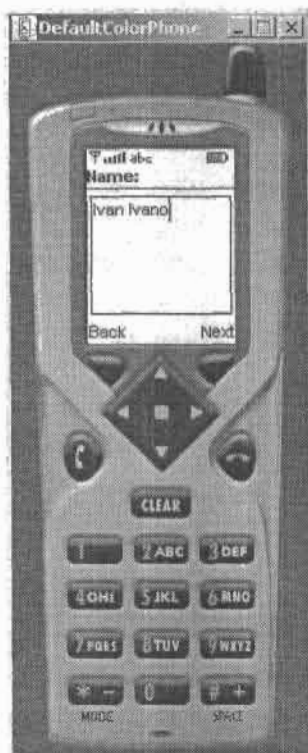


Рис. 10.3. Экран ввода параметров записи на стандартном эмуляторе

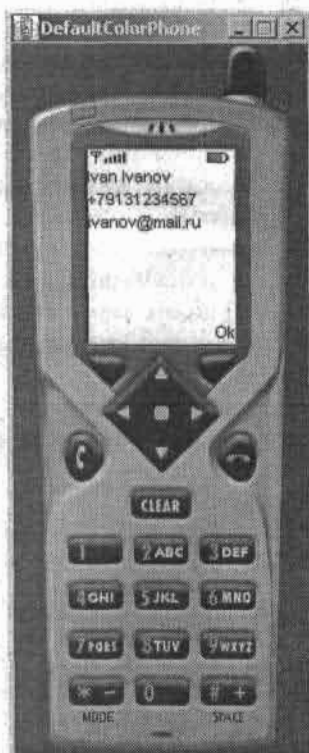


Рис. 10.4. Информация из записной книжки выведена на экране

Один большой минус заключается в том, что, в общем случае, доступа из мидлета к стандартной адресной книге телефона и ее записям нет, то есть невозможно получить программно записи из стандартной адресной книги. Иногда, если повезет, такой доступ предоставляет API производителя конкретной модели телефона.

В этой главе мы рассмотрели на примерах работу с полями ввода, представленными объектами классов `TextBox` и `TextField`. Даже в небольшой программе адресной книги нам понадобились знания почти из всех предыдущих глав. Мы использовали интерфейс высокого уровня для связи с пользователем и реализации логики смены экранов, создали список имен с помощью объекта класса `List`, организовали долговременное хранение записей с использованием RMS, осуществили запись и чтение данных с помощью потоков ввода-вывода. Возможности мобильного языка J2ME очень ограничены, нетрудно разобраться со всеми и попробовать их в действии. Это поможет реально оценивать свои силы при проектировании функциональности программ.

Глава 11

Работа со временем, датой и календарем

Продолжим наше знакомство с программированием для мобильных телефонов. Современный аппарат — это больше, чем средство связи, ведь он напичкан всевозможными, нужными и не очень, дополнениями, которые к самой связи имеют отдаленное отношение. Некоторые из усовершенствований, которые продавцы салонов связи демонстрируют первым делом, покажутся откровенно лишними, а без других уже сложно представить себе телефон. Например, с появлением телефона я совсем перестал пользоваться наручными часами, настольным будильником и карманным калькулятором.

Реализация J2ME предоставляет нам возможность пользоваться внутренним временем и календарем телефона. Стоит подключить фантазию и таким возможностям можно легко найти применение. Например, составление ежедневных индивидуальных гороскопов или организация испытательных тридцатидневных версий приложений. В этой главе мы рассмотрим использование объектов календаря и даты для расширения записной книжки функцией напоминания о днях рождения.

Класс Date

Объект класса `Date` представляет собой текущее время с точностью до миллисекунд. Время представлено в миллисекундах, прошедших с полуночи 1 января 1970 года. Информация, безусловно, нужная, но в таком виде совершенно бесполезная. К сожалению, класс `Date` не содержит интерпретаторов для перевода даты в более понятный для нас формат. Для этого нужно использовать класс `Calendar`, который мы рассмотрим позже.

Класс `Date` имеет всего несколько методов. Все они оперируют с датой и временем лишь в указанном формате. Рассмотрим эти методы:

- `Date()` — конструктор; создает новый объект даты, содержащий текущее время;
- `Date(long date)` — конструктор; создает новый объект даты, содержащий время, заданное в аргументе `date`;
- `boolean equals(Object obj)` — возвращает значение `true` в том случае, если две даты идентичны с точностью до миллисекунды;
- `long getTime()` — возвращает время, представленное объектом даты;
- `void setTime(long time)` — устанавливает объект даты в момент времени, представленный параметром `time`.

Вот, собственно, и все. Разнообразием и гибкостью методов класс `Date` нас не радует. Самое время сформулировать задачу, которую мы хотим реализовать. К программе записной книжки, которую мы разбирали в прошлой главе, добавим функцию напоминания о днях рождения записанных абонентов. Для этого потребуется расширить формат записи, содержащейся в хранилище записей, добавив поле даты рождения. В сценарий ввода данных о новом абоненте потребуется добавить экран для ввода даты рождения. Здесь нас ждет приятный сюрприз, потому что поле ввода даты предоставляет нам пользовательский интерфейс высокого уровня в виде класса `DateField`.

Класс `DateField`

Класс `DateField` является потомком класса `Item`, то есть для его демонстрации на экране нам потребуется еще и форма. Поле может работать в трех режимах ввода: ввод только даты, ввод только времени, ввод даты и времени. При вводе только времени дата по умолчанию будет соответствовать 1 января 1970 года. Для работы с полем ввода класс `DateField` предоставляет следующие методы:

■ `DateField(String label, int mode)` — конструктор; создает поле ввода даты с заголовком `label`. Режим поля ввода определяется аргументом `mode`, который может принимать значение одной из следующих определяющих констант:

- `DATE` — ввод только даты;
- `DATE_TIME` — ввод даты и времени;
- `TIME` — ввод только времени;

- `Date getDate()` — возвращает введенное значение даты и времени в формате класса `Date`. При создании объекта исходное значение даты и времени не определено, поэтому при запросе отсутствующей введенной даты будет возвращено значение `null`;
- `int getInputMode()` — возвращает режим ввода поля в виде определяющей константы;
- `void setDate(Date date)` — устанавливает в поле ввода новое значение, представленное параметром `date`;
- `void setInputMode(int mode)` — устанавливает новый режим ввода, заданный аргументом `mode` в виде определяющей константы;
- `void setLabel(String label)` — устанавливает новый заголовок поля ввода.

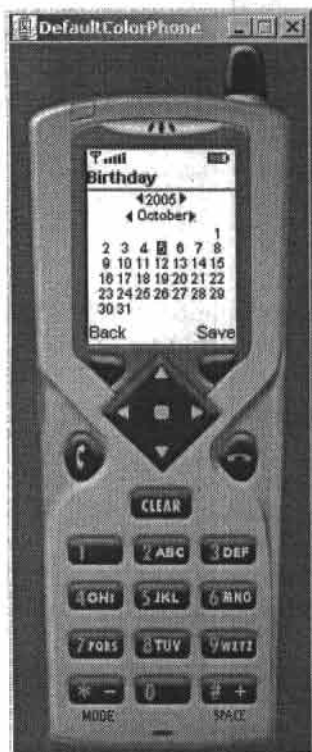


Рис. 11.1. Экран установки даты в стандартном эмуляторе WTK

Поскольку поле ввода даты предоставлено интерфейсом высокого уровня, то внешний вид поля и логика ввода зависят от конкретной модели телефона. Например, в стандартном эмуляторе от WTK поле ввода даты выглядит следующим образом (рис. 11.1).

На данном этапе информации достаточно, чтобы начать дополнение нашей записной книжки. Прежде всего, импортируем необходимые нам библиотеки и добавим в класс AddressBook два новых поля, которые будут предоставлять возможность ввода даты рождения:

```
...
import javax.microedition.lcdui.DateField;
import java.util.Date;
import java.util.Calendar;
...
private Form dateForm;           // форма ввода дня рождения
private DateField dateField;     // поле ввода дня рождения
```

В методе startApp() проинициализируем поле ввода даты аналогично полям ввода остальной личной информации:

```
...
// экран ввода дня рождения
dateField = new DateField("Birthday", DateField.DATE);
dateForm = new Form("");
dateForm.append(dateField);
dateForm.addCommand(next);
dateForm.addCommand(back);
dateForm.setCommandListener(this);
...

```

В блок прослушивания команд добавим логику демонстрации экрана ввода дня рождения, а также дополнительное поле в форме отображения параметров записи. Хранимая запись теперь будет состоять из четырех частей: три текстовых части и одна типа long, содержащая дату рождения в формате класса Date.

Экран информации записи будет теперь формироваться следующим образом:

```
// если команда вызвана из экрана со списком имен
if(d==nameList) {
    try {
        ...
        // создать объект даты рождения
        Date birthday = new Date(dis.readLong());
        // создать объект поля ввода даты
        DateField df = new DateField("", DateField.DATE);
        // установить дату рождения в поле ввода даты
        df.setDate(birthday);
        // добавить даты рождения в форму отображения параметров
        infoForm.append(df);
        ...
        display.setCurrent(infoForm);
    }
}
```

Запись, в свою очередь, будет формироваться так:

```
try {
    // записать введенные параметры в поток вывода
    dos.writeUTF(tbName.getString());
    dos.writeUTF(tbPhone.getString());
    dos.writeUTF(tbEMail.getString());
    dos.writeLong(dateField.getDate().getTime());
    // добавить запись в хранилище
    recordStore.addRecord(baos.toByteArray(), 0, baos.size());
}
```

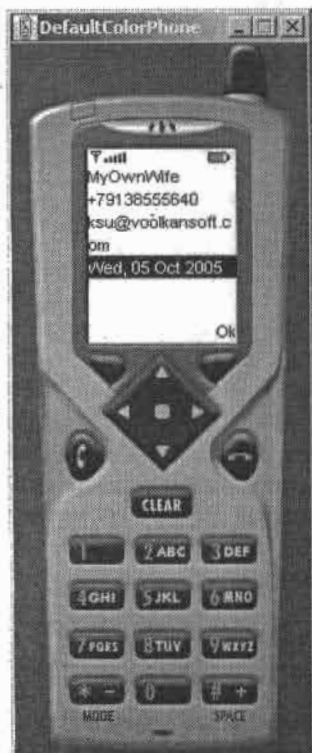


Рис. 11.2. К информации об абоненте добавлена дата рождения

С корректной сменой экранов при обработке команд Next и Back, думаю, вы справитесь самостоятельно. Теперь во время ввода нового контакта появится поле ввода даты рождения, а форма информации об абоненте будет выглядеть как на рис. 11.2.

Заметим, что записи, введенные в программе из предыдущей главы, имеют другой формат, поэтому при попытке считать их в новом формате будет формироваться исключение, а форма с информацией демонстрироваться не будет.

Осталось реализовать самый важный момент: нужно получить текущую дату и сравнить ее с датами рождения, содержащимися во всех записях. Сейчас самое время познакомиться с классом Calendar, без которого пересчет компьютерных миллисекунд в человеческие дни и месяцы изрядно попортит бы нам нервы.

Класс Calendar

Класс Calendar является абстрактным классом для работы с датой и временем, содержащим набор констант, представляющих дни недели и месяцы. Для работы с месяцами определены константы от JANUARY до DECEMBER, а для работы с днями недели константы от MONDAY до SUNDAY. Поскольку класс Calendar является абстрактным, мы не можем просто создать объект этого класса, но мы можем получить его с помощью специального метода:

- `static Calendar getInstance()` — получить объект календаря, представляющий текущие дату и время.

Класс Calendar связан с уже рассмотренным классом Date следующими методами:

- `Date getTime()` — возвращает текущее время, представленное объектом календаря в виде объекта класса Date;
- `void setTime(Date date)` — устанавливает календарю текущее время, представленное объектом класса Date.

Преобразование времени и даты осуществляется с помощью методов `get` и `set`:

- `int get(int field)` — возвращает преобразованное значение одного из параметров календаря. Параметр определяется аргументом `field`. Аргумент `field` может принимать значение одной из следующих констант: `YEAR`, `MONTH`, `DATE`, `DAY_OF_WEEK`, `HOURL_OF_DAY`, `HOUR`, `AM_PM`, `MINUTE`, `SECOND`, `MILLISECOND`. Метод возвращает значение запрошенного параметра в виде константы, определяющей месяц или день недели, либо числовое значение запрошенной единицы времени. Константа `HOURL_OF_DAY` определяет значение часа в 24-часовом формате, а константа `HOUR` — в 12-часовом. Константа `AM_PM` используется в 12-часовом формате для определения значения «до полудня» (`AM`) или «после полудня» (`PM`);
- `void set(int field, int value)` — устанавливает значение `value` в один из параметров даты или времени, который определяется аргументом `field`. Параметр `DAY_OF_WEEK` не может быть установлен.

Таким образом, чтобы из объекта класса `Date` получить дату и время в виде удобных констант, необходимо получить объект календаря, присвоить ему время, представленное объектом `Date`, и лишь затем интерпретировать его, используя методы класса `Calendar`. Так же как и в классе `Date`, календарь может работать со временем, представленным количеством миллисекунд, прошедших от начала 70-х:

- `long getTimeInMillis()` — возвращает время календаря в миллисекундах;
- `void setTimeInMillis(long millis)` — устанавливает время календаря, заданное параметром `millis`.

Класс `Calendar` организует сравнение времени двух календарей с помощью следующих методов:

- `boolean equals(Object obj)` — возвращает значение `true`, если аргумент `obj` не равен `null`, и представляет то же самое время, что и вызывающий объект календаря;
- `boolean before(Object when)` — возвращает значение `true`, если время, представленное вызывающим объектом, ранее времени, представленного аргументом `obj`;
- `boolean after(Object when)` — возвращает значение `true`, если время, представленное вызывающим объектом, позднее времени, представленного аргументом `obj`.

Кроме того, класс `Calendar` поддерживает работу с часовыми поясами:

- `TimeZone getTimeZone()` — получить часовой пояс календаря;
- `void setTimeZone(TimeZone value)` — установить часовой пояс календаря.

Класс `TimeZone`

Для полноты картины предоставляемых реализацией J2ME возможностей, рассмотрим абстрактный класс `TimeZone`, представляющий часовой пояс. Класс `TimeZone` позволяет узнать, какой часовой пояс установлен на аппарате, выполняющем программу, и осуществляется ли переход на летнее время. Класс `TimeZone` имеет следующий набор методов:

- `String[] getAvailableIDs()` — возвращает аббревиатуры всех поддерживаемых аппаратом часовых поясов в виде массива строк;

- `TimeZone getDefault()` — возвращает часовой пояс, используемый аппаратом по умолчанию;
- `String getID()` — возвращает аббревиатуру текущего часового пояса, используемого аппаратом;
- `int getRawOffset()` — возвращает смещение данного часового пояса относительно времени по Гринвичу;
- `TimeZone getTimeZone(String ID)` — получить объект часового пояса по его аббревиатуре;
- `boolean useDaylightTime()` — возвращает значение `true`, если часовой пояс использует переход на летнее время (DST — Daylight Savings Time).

Класс BirthdayFilter

Теперь мы располагаем всеми необходимыми средствами для реализации функции напоминания о дне рождения в нашей записной книжке. Для выбора именинников из хранилища записей реализуем класс фильтра `BirthdayFilter`, который будет сравнивать даты рождения с текущей датой и выбирать лишь интересующие нас записи:

// класс фильтра записей по дням рождения

```
private class BirthdayFilter implements RecordFilter {
    // метод сравнения записей
    public boolean matches(byte[] candidate) {
        // преобразовать запись в байтовый поток
        ByteArrayInputStream bais = new ByteArrayInputStream(candidate);
        // создать поток, поддерживающий чтение по типу
        DataInputStream dis = new DataInputStream(bais);
        // день рождения
        Date birthDate = new Date();
        try {
            // считать строковые параметры записи
            dis.readUTF();
            dis.readUTF();
            dis.readUTF();
            // считать дату рождения
            birthDate.setTime(dis.readLong());
        }
        catch (IOException ioe) { return false; }

        // получить два календаря с текущей датой
        Calendar rightNow = Calendar.getInstance();
        Calendar birthday = Calendar.getInstance();
        // установить дату рождения именинника
        birthday.setTime(birthDate);
        // сравнить день и месяц рождения с текущей датой
        if(rightNow.get(Calendar.DAY_OF_MONTH) == birthday.get(Calendar.DAY_OF_MONTH) &&
```

```

        rightNow.get(Calendar.MONTH) == birthday.get(Calendar.MONTH))
        return true;
    else
        return false;
    }
}

```

Реализуем функцию, которая будет фильтровать хранилище записей по текущей дате и возвращать имя человека, у которого сегодня день рождения, или значение null, если таковых не имеется:

```

// метод SearchBirthday возвращает строку с именем именинника;
// если в этот день именинников нет, то возвращает null
private String SearchBirthday() {
    String name = null;
    try {
        // создать объект фильтра по дню рождения
        BirthdayFilter filter = new BirthdayFilter();
        // получить список записей с подходящим днем рождения
        RecordEnumeration re = recordStore.enumerateRecords(filter, null, false);
        // получить ID записи
        int id = re.nextRecordId();
        // получить запись по ID
        byte[] record = recordStore.getRecord(id);
        // преобразовать запись в байтовый поток
        ByteArrayInputStream bais = new ByteArrayInputStream(record);
        // создать поток, поддерживающий чтение по типу
        DataInputStream dis = new DataInputStream(bais);
        // считать из потока строку с именем
        name = dis.readUTF();
    }
    catch(RecordStoreException rse) {}
    catch(IOException ioe) {}
    // вернуть имя именинника
    return name;
}

```

Все, что нам остается сделать, это вызвать в стартовом методе приложения startApp() функцию поиска именинников и при необходимости вывести соответствующее напоминание:

```

...
// создать список имен
BuildNameList();
// поиск именинника
String name = SearchBirthday();
// если имя найдено
if(name!=null) {
    // создать форму напоминания
}

```

```

Form remindForm = new Form("Reminder");
// добавить напоминание в форму
remindForm.append(name + " has a birthday today!");
// добавить команду возврата
remindForm.addCommand(ok);
remindForm.setCommandListener(this);
// отобразить форму
display.setCurrent(remindForm);
} else
// отобразить список имен на экране
display.setCurrent(nameList);

```

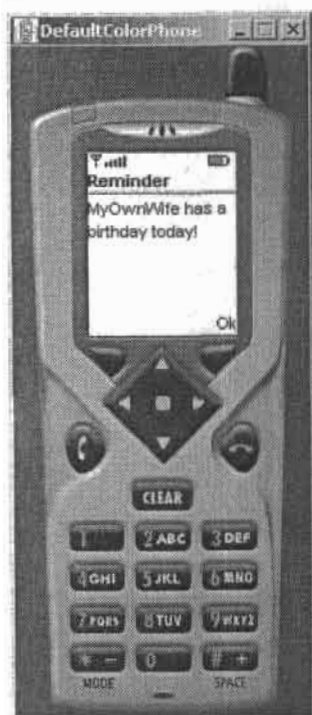


Рис. 11.3. Телефон напомнит вам, кого надо поздравить с днем рождения

Теперь все готово. При первом запуске нужно добавить контакт с текущей датой рождения, а затем заново запустить приложение. Если все запрограммировано верно, то при запуске появится сообщение, показанное на рис. 11.3.

Основная недоработка программы заключается в том, что если у нескольких ваших знакомых день рождения в один день, то напоминание все равно будет только одно. Неплохо было бы также добавить возможность удаления и редактирования контактов. Напоминание можно красочно оформить, а также с легкостью высчитать возраст именинника. Все эти детали, а также любые другие капризы вашей фантазии, остаются вам на самостоятельную доработку.

* * *

Итак, в этой главе мы изучили работу с датой, временем, календарем и часовым поясом. Мы рассмотрели лишь один пример из всего многообразия возможных приложений, имеющих дело со временем и датой. Часовой пояс, например, можно использовать для интернационализации программ и автоматического выбора языка интерфейса. А теперь можете сделать паузу и опробовать изученный материал. В добрый путь!

Глава 12

Работа над ошибками

С обработкой исключительных ситуаций мы уже сталкивались неоднократно во время реализации наших нехитрых примеров. Пришло время подробнее разобраться со структурой и механизмом формирования исключений.

Во время выполнения программы может случиться много разных неприятностей, таких как деление на ноль, выход индекса массива за допустимые пределы, неверное имя загружаемого файла и масса других. В функционально-ориентированных языках программирования, таких как C или Pascal, такие ситуации обрабатывались с помощью многочисленных операторов `if...else`, которые значительно увеличивают как объем, так и время выполнения программы. В объектно-ориентированных языках реализован другой подход к обработке внешних ситуаций.

Класс Exception

При возникновении исключительной ситуации система формирует соответствующий объект-исключение, содержащий сведения об ошибке, и передает его на обработку программе, в которой произошла ошибка. Если программа не предусматривает особую обработку данного исключения, то объект передается системе, которая останавливает выполнение программы и выводит на экран тип сформированного исключения. При делении на ноль, например, это происходит как показано на рис. 12.1.

Остановку выполнения программы можно предотвратить, реализовав в программе обработку возможных исключений. Как мы уже видели на примерах, команды, потенциально формирующие исключения, заключаются в блок `try{...}`, а действия, которые следует выполнить в исключительной ситуации, заключаются в блок `catch(Exception e) {...}`. В качестве аргумента опе-

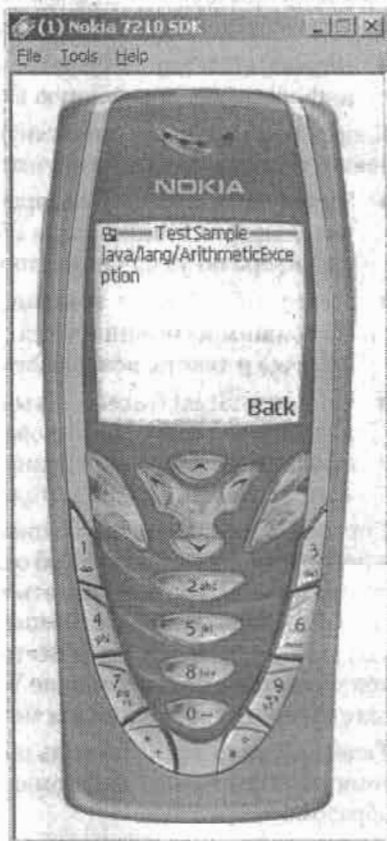


Рис. 12.1. В случае деления на ноль программа прерывается

ратор `catch` принимает объект исключения определенного типа в зависимости от ситуации, которую мы хотим обработать. После одного блока `try` может следовать несколько блоков `catch` с обработкой разных исключений. Управление будет передано первому блоку, аргумент которого будет соответствовать сформированному исключению. Таким образом, синтаксис блока проверки и обработки исключений в общем случае выглядит так:

```
try { ...
}
catch(...) {...}
...
catch(...) {...}
```

Остановимся на объектах исключений, которые формируются системой. Все они наследованы от одного и того же класса `Exception`, который, в свою очередь, порожден от класса `Throwable`. Только объекты, порожденные от класса `Throwable`, могут быть обработаны в блоке `catch` или искусственно сформированы с помощью оператора `throw`, который мы рассмотрим немного позже.

Класс `Throwable` и все его расширения содержат два конструктора:

- `Throwable()` — конструктор по умолчанию;
- `Throwable(String message)` — объект исключения будет содержать текстовую информацию, переданную в строке `message`.

Класс `Throwable` несет дополнительную информацию о сформированном исключении, которую можно получить с помощью следующих методов:

- `String getMessage()` — возвращает строку с текстовой информацией об исключительной ситуации. Если объект был создан пользователем с помощью конструктора по умолчанию, то возвращает значение `null`;
- `String toString()` — возвращает строку с текстовым описанием исключения, состоящим из имени класса сформированного исключения, двоеточия (разделителя) и текста, возвращаемого методом `getMessage`;
- `void printStackTrace()` — выводит в область сообщения об ошибках название класса и метода, сформировавших исключение, со всей предыдущей историей вызовов, включая внутреннюю реализацию. Первая выводимая строка содержит результат работы метода `toString`.

Система определяет два потока вывода сообщений: поток вывода информации и поток вывода сообщений об ошибках. Поток вывода доступен из любой точки программы. Мы можем писать туда любую полезную информацию, включая содержимое переменных, с помощью системных методов: `System.out.println(String text)` и `System.err.println(String text)`. В нашем случае сообщения обоих потоков появятся в основном окне WTK. При запуске приложения на реальном аппарате никаких сообщений мы можем и не увидеть.

Теперь, если деление на ноль поместить в блок `try`, а в блоке `catch` вывести в системную область всю информацию о сформированном исключении следующим образом:

```
int n=0;
try {
```

```

    int k = 5/n;
} catch(Exception exc) {
    System.out.println("Get Message: " + exc.getMessage());
    System.out.println("To String: " + exc.toString());
    System.err.println("Print Stack Trace:");
    exc.printStackTrace();
}
System.out.println("Continue program...");

```

то в области вывода системных сообщений мы увидим такую картину (рис. 12.2).

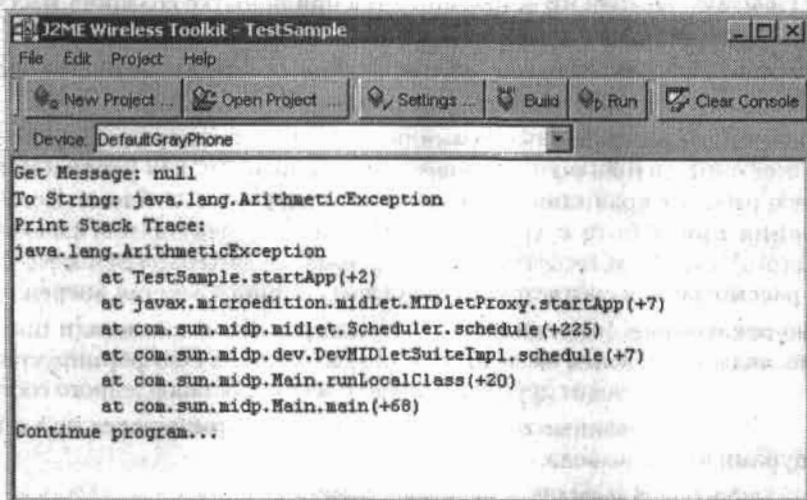


Рис. 12.2. В основное окно WTK может быть выведена подробная информация об ошибке

Заметим, что программа после формирования исключения и вывода сообщений в системную область не зависла, не упала, не вылетела, а спокойно продолжила свое выполнение с команды, следующей за обработкой исключения.

Обработка некоторых исключений, называемых проверяемыми, является обязательной для программиста. Ее отсутствие приведет к ошибке на этапе компиляции. Например, если убрать обработку исключения при открытии хранилища записей, то программа не откомпилируется и выдаст следующую ошибку:

```

AddressBook.java:44: unreported exception javax.microedition.rms.RecordStoreException:
must be caught or declared to be thrown
recordStore = RecordStore.openRecordStore("Address-Book", true);

```

Рассмотрим некоторые наиболее распространенные исключения, формируемые реализацией J2ME и уже использованные нами в примерах. Основная группа исключений, наследованных от класса `RuntimeException`, является непроверяемой, поэтому к ним нужно относиться особо внимательно. В наших примерах обработка исключений была опущена везде, где только возможно, что не есть хорошо.

- `ArithmeticException` — формируется при ошибках в арифметических операциях, например при делении на ноль;

- `ArrayStoreException` — формируется при попытке записать в массив данные неверного типа;
- `EmptyStackException` — формируется во время работы со стеком при попытке получения данных, когда стек пуст;
- `IllegalArgumentException` — возникает при передаче в метод некорректного значения аргумента. Если в качестве аргумента предусмотрена константа, то лучше пользоваться ей в чистом виде во избежание подобных ситуаций;
- `IndexOutOfBoundsException` — указывает на то, что используемый индекс массива, вектора или строки выходит за пределы допустимых значений;
- `NegativeArraySizeException` — формируется при попытке создания массива отрицательной длины;
- `NullPointerException` — формируется при попытке вызвать метод или модифицировать поле объекта, значение которого `null`.

Исключения, сформированные хранилищем записей, наследованы от класса `RecordStoreException` и являются проверяемыми, поэтому для успешной компиляции всю работу с хранилищем мы оформляли в `try...catch` блоки. Возможные исключения при работе с хранилищем записей (`InvalidRecordIDException`, `RecordStoreFullException`, `RecordStoreNotFoundException`, `RecordStoreNotOpenException`) мы уже рассмотрели в соответствующей главе, немного забежав вперед.

Еще одно исключение, `InterruptedException`, которое мы использовали при работе с тредами, является прямым наследником класса `Exception`. Оно формируется в случае, когда один тред выводит другой из спящего или приостановленного состояния.

Исключения, наследованные от класса `IOException`, формируются при ошибках с процедурами ввода-вывода:

- `ConnectionNotFoundException` — цель соединения не найдена;
- `EOFException` — конец файла был достигнут при работе с потоком ввода;
- `InterruptedIOException` — формируется в случае, если операция ввода-вывода была прервана;
- `UnsupportedEncodingException` — данная кодировка не поддерживается;
- `UTFDataFormatException` — формируется при работе с `Unicode`, если символы строки не являются символами универсальной кодировки.

В рамках данной главы мы рассмотрели не все возможные исключения. В документации, без которой вам в любом случае не обойтись, в описании каждого метода содержится список всех возможных исключений, которые могут быть сформированы при выполнении данного метода.

Формирование исключений

Кроме существующих классов исключений можно формировать собственные исключения, реализовав свой класс, наследованный от класса `Exception` или от любого из его многочисленных потомков.

Рассмотрим формирование собственного исключения на примере программы калькулятора, которую мы разбирали в одной из прошлых глав. Оформим проверку

наличия всех аргументов в форме ввода в виде формирования исключения. Для этого реализуем класс исключения `MissingArgException`:

```
class MissingArgException extends Exception {
    // текстовое сообщение исключения
    private String msg;
    // конструктор без параметров
    MissingArgException(){ msg = null;}
    // конструктор, принимающий строку
    MissingArgException(String str) { msg = str;}
    // метод возврата текстового сообщения
    public String toString(){
        return "Exception (" + msg + ")";
    }
}
```

Исключения активизируются с помощью оператора `throw`, который позволяет выбрасывать любые исключения иерархии класса `Throwable` из любого места программы. Команда `throw new ArithmeticException()` приведет к такому же эффекту, как если бы здесь произошло деление на ноль, и обработка этого исключения пойдет своим чередом. Тот факт, что метод формирует исключение, обозначается в заголовке метода ключевым словом `throws` и названием класса исключения. Реализуем метод проверки наличия аргументов в форме:

```
void checkArgs() throws MissingArgException {
    // проверить заполнение всех полей ввода
    // сформировать исключение в случае отсутствия аргумента
    if(arg1.size()==0)
        throw new MissingArgException("Missing First Argument");
    if(arg2.size()==0)
        throw new MissingArgException("Missing Second Argument");
    if(action.size()==0)
        throw new MissingArgException("Missing Action Sign");
}
```

Проверка заполнения формы ввода в методе `commandAction` будет выглядеть теперь следующим образом:

```
// проверить заполнение всех полей ввода
try {
    checkArgs();
} catch(MissingArgException maexc) {
    // ошибка: не все поля ввода заполнены
    resForm.append(maexc.toString());
    // отобразить сообщение
    display.setCurrent(resForm);
    // выйти
    return;
}
```

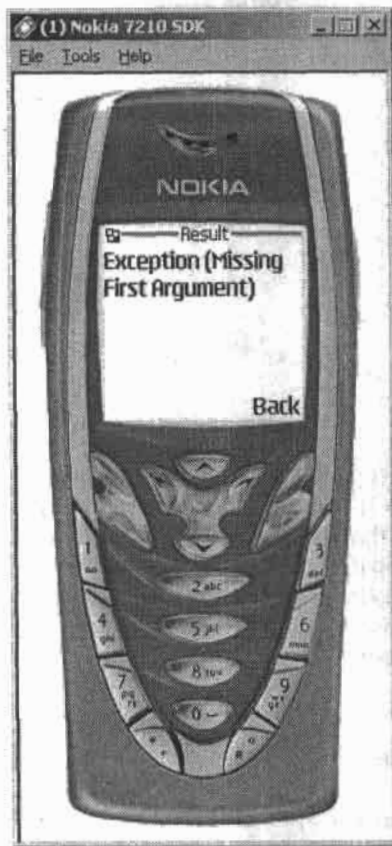


Рис. 12.3. При отсутствии аргументов формируется исключение и выводится сообщение об ошибке

Если запустить программу и вызвать команду вычисления результатов, не задавая аргументов, то сформируется соответствующее исключение, а мы увидим на экране сообщение (рис. 12.3).

Для отображения сообщения об ошибке мы воспользовались обычной формой. На самом деле существует специально предназначенный для этого класс.

Класс Alert

Неважно, пользуется ли программой ваш друг или человек, заплативший за это деньги, — ситуация, когда приложение вылетает, оставляя за собой лишь непонятное сообщение на экране, считается дурным тоном в программировании. Большинство исключений не интересуют конечного пользователя программы, но иногда нужно дать понять, что его действия некорректны.

Поскольку системная область сообщений недоступна пользователю, существует цивилизованный способ вывода подобной информации. Класс `Alert` представляет собой сообщение, состоящее из изображения и текста, которое отображается на экране в течение заданного количества времени, после чего автоматически происходит смена экрана. Класс `Alert` используется для информирования пользователя об ошибках и возникших исключительных ситуациях, а также для кратковременных информационных сообщений о результатах выполнения операций.

Время демонстрации сообщения может быть бесконечно большим. Тогда сообщение будет отображаться на экране до тех пор, пока пользователь вручную не закроет его. Аналогичная ситуация возникает, если большое количество информации требует прокрутки экрана. Остановимся на методах класса `Alert`:

- `Alert(String title)` — конструктор создает пустое сообщение с заголовком `title`;
- `Alert(String title, String alertText, Image alertImage, AlertType alertType)` — конструктор создает сообщение с заголовком `title`, текстом `alertText` и картинкой `alertImage`. Параметр `alertType` задает тип сообщения, представленный классом `AlertType`, который мы еще рассмотрим подробнее. Параметр `alertType` может принимать значение `null` для создания обычного сообщения;
- `void addCommand(Command cmd)` — разумное объяснение наличию этого метода придумать сложно, поскольку класс `Alert` не поддерживает добавление команд. При вызове метода будет сформировано исключение `IllegalStateException`;
- `int getDefaultTimeout()` — возвращает значение времени демонстрации сообщения в миллисекундах, устанавливаемое по умолчанию при создании объекта

класса `Alert`. Может также вернуть значение константы `FOREVER`, если по умолчанию сообщение демонстрируется неограниченное количество времени;

- `Image getImage()` — возвращает объект картинки, установленной для сообщения, или значение `null`, если картинка не установлена;
- `String getString()` — возвращает текст данного сообщения;
- `int getTimeout()` — возвращает время демонстрации данного сообщения в миллисекундах или константу `FOREVER`;
- `AlertType getType()` — возвращает тип данного сообщения в виде объекта класса `AlertType`;
- `void setCommandListener(CommandListener l)` — так же, как и метод `addCommand`, при вызове всегда формирует исключение `IllegalStateException`;
- `void setImage(Image img)` — устанавливает новую картинку для сообщения;
- `void setString(String str)` — устанавливает текст `str` данному сообщению;
- `void setTimeout(int time)` — устанавливает новое время демонстрации сообщения. Время задается аргументом `time` в миллисекундах или константой `FOREVER` класса `Alert` для неограниченного времени демонстрации сообщения;
- `void setType(AlertType type)` — задает новый тип сообщения в виде объекта класса `AlertType`.

Класс `AlertType`

Класс `AlertType` задает тип сообщения класса `Alert`. Тип сообщения определяет шаблон, в котором будет продемонстрировано сообщение. Константы класса `AlertType` определяют следующие шаблоны сообщений:

- `ALARM` — события, привязанные к определенным моментам, например напоминание о назначенной встрече;
- `CONFIRMATION` — подтверждение успешных результатов действий пользователя, например «Сообщение отправлено»;
- `ERROR` — сообщение об ошибке в результате действий пользователя, например «Недостаточно памяти для сохранения данных»;
- `INFO` — отображение информационных сообщений;
- `WARNING` — предупреждение об опасных действиях пользователя, например «Все данные будут уничтожены».

Класс `AlertType` содержит лишь два метода: конструктор, не принимающий аргументов, и метод воспроизведения звукового сигнала, связанного с типом сообщения:

- `boolean playSound(Display display)` — воспроизводит звук, соответствующий данному типу сообщения. В определенных устройствах некоторые звуки могут отсутствовать. Возвращает значение `true`, если звук был воспроизведен, и `false` — в противном случае. Метод `playSound` может быть вызван и без использования сообщений класса `Alert`, например для организации звуковых сигналов во время игры.

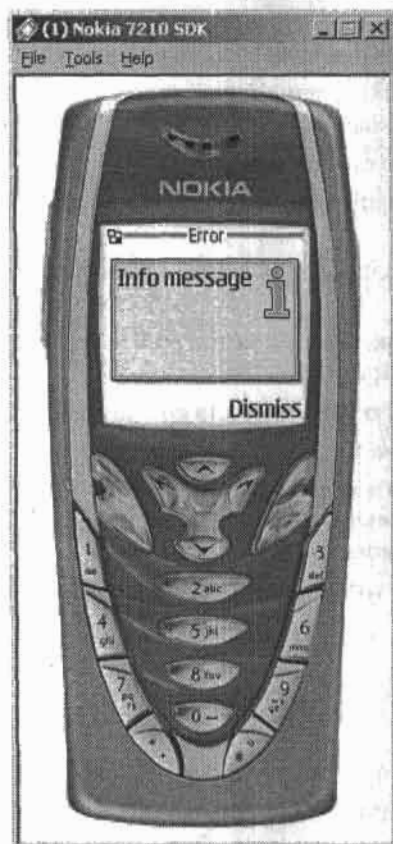


Рис. 12.4. Пример информационного сообщения

Внешний вид и звуковой сигнал сообщений каждого типа зависят от конкретной модели телефона. Например, на телефоне Nokia сообщение информационного типа выглядит, как показано на рис. 12.4.

Класс `Alert` является потомком класса `Screen`, то есть экраном, готовым к демонстрации. Поскольку смена экрана по истечении тайм-аута происходит автоматически, то демонстрация сообщения осуществляется с помощью особого метода менеджера дисплея (объекта класса `Display`), который принимает объект класса `Alert`, а также объект следующего экрана:

- `void setCurrent (Alert alert, Displayable nextDisplayable)` — устанавливает в качестве текущего экрана сообщение, представленное аргументом `alert`, и задает экран `nextDisplayable`, который станет текущим после закрытия сообщения. Объект `nextDisplayable` не может быть объектом класса `Alert` или иметь значение `null`.

Рассмотрим использование класса `Alert` на примере все той же программы калькулятора. Дополним программу проверкой деления на ноль, которую мы проигнорировали, из-за чего приложение в такой ситуации вылетало с ошибкой. В блоке `catch` перехватим исключение ошибки в арифметической операции, создадим объект сообщения об ошибке с неограниченным временем демонстрации и отобразим его, указав основную форму ввода в качестве следующего демонстрируемого экрана:

```
...
import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.AlertType;
...
try {
    switch(act[0]) {
        // выполнить действие в соответствии
        // с символом операции
        case '+':
            res = first + second;
            break;
        case '-':
            res = first - second;
            break;
        case '*':
```

```
res = first * second;
break;
case '/':
res = first / second;
break;
default:
// ошибка: некорректный символ операции
resForm.append("Illegal Operation");
// отобразить сообщение
display.setCurrent(resForm);
// выйти
return;
}
// преобразовать результат в строку и добавить в форму
resForm.append((new Integer(res)).toString());
// отобразить результат
display.setCurrent(resForm);

// блок перехвата арифметического исключения
} catch (ArithmeticException aexc) {
// создать сообщение об ошибке
Alert alert = new Alert("Error", "Divide by zero", null, AlertType.ERROR);
// установить бесконечное время демонстрации
alert.setTimeout(Alert.FOREVER);
// установить сообщение в качестве текущего экрана
display.setCurrent(alert.form);
}
```

Аналогичным образом вы самостоятельно можете обработать ошибку некорректного символа арифметического действия. Попробуем программу в действии и разделим что-нибудь на ноль. Теперь вместо остановки программа продолжает свою работу, отображая на экране сообщение, показанное на (рис. 12.5).

После нажатия клавиши Done программа возвращается к первоначальному экрану ввода. Заметим, что добавление и обработку этой команды мы в программе не предусматривали, она появилась в результате установки объекту класса Alert бесконечного времени демонстрации. В случае установки фиксированного времени демонстрации команда Done не отображается и смена экранов происходит автоматически по истечении заданного тайм-аута.

До сих пор в примерах мы лишь перехватывали возможные исключения, чтобы программа компилировалась и не вылетала при возможных ошибках. Внима-

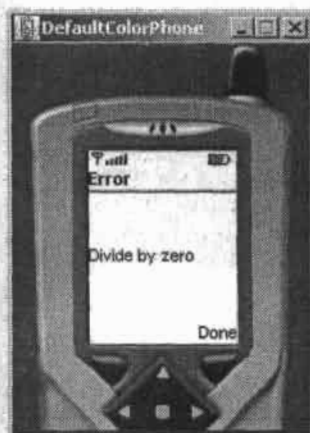


Рис. 12.5. Теперь при делении на ноль программа не падает, а сообщает, что пользователь был не прав

ние на этом не заострялось, чтобы не отвлекаться от основной темы. В действительности же грамотная обработка исключений и вовремя предпринятые действия помогут решить много проблем. Также неверно оставлять блок обработки исключения `catch` пустым, как мы это делали в примерах. Даже простое сообщение в области ошибок принесет немало пользы в процессе отладки программы.

При обработке нескольких типов исключений в одном `try`-блоке следует быть внимательными, не перехватывается ли необходимое исключение в одном из блоков `catch`, расположенных выше. Так, если в первом же блоке `catch` перехватывается исключение самого общего типа `Exception`, то до остальных блоков дело не дойдет никогда. Немного практики и терпения, и вы научитесь лавировать меж подводных камней объектно-ориентированных программ, а главное, помните, что аккуратность — главный залог успеха программиста.

Глава 13

Расширение функциональности с помощью API

На данный момент мы рассмотрели большинство возможностей реализации MIDP 1.0 мобильного языка J2ME. Возможности эти, прямо скажем, небезграничны, а порой просто разочаровывают. Это и есть плата за универсальность платформы и совместимость приложений для многих моделей телефонов. Расширять возможности J2ME можно в двух направлениях: с помощью профайла MIDP 2.0 и с использованием специальных классов от производителей конкретной модели телефона. Оба варианта значительно сокращают ряд моделей, где наша программа будет чувствовать себя комфортно, то есть работать как следует.

Возможно, в ближайшем будущем телефоны с поддержкой профайла MIDP 2.0 не будут дорогостоящей редкостью, что позволит освоить новые горизонты мобильного программирования. А пока что подавляющее большинство моделей телефонов на душу населения заключено в жесткие рамки профайла MIDP 1.0, на который и сделан основной акцент в этой книге.

В этой главе мы остановимся подробнее на втором варианте расширения функциональности, то есть на API-функциях компаний-производителей. API (Application Programming Interface, интерфейс программирования приложений) является набором классов и методов для конкретных моделей определенного производителя.

Мы рассмотрим несколько разных API и начнем с самой распространенной модели, эмулятором которой мы пользовались на протяжении всей книги. Эмулятор Nokia 7210 уже включает в себя все необходимые классы API, и если вы следовали инструкциям и устанавливали все в указанные в первой главе папки, то J2ME Wireless Toolkit обнаружит все, что нужно, самостоятельно. Распирять мы будем нашу любимую «Змейку», в которую добавим несколько приятных моментов.

Nokia API

Посмотрим, чем же нас могут порадовать разработчики от корпорации Nokia. На данный момент нам предлагаются два пакета классов `com.nokia.mid.ui` и `com.nokia.mid.sound`, которые предоставляют возможности полноэкранного режима, низкоуровневого доступа к экрану, управления звуком, подсветкой, а также вибрацией.

Помним, на чем мы остановились, программируя нашу змею? Название приложения, отображенное на экране, благополучно отбело у нас по 16 пикселей не только

сверху, но и снизу — так, для симметрии (рис. 13.1). Экран и так не слишком велик, чтобы во время игры занимать его никому не нужными заголовками. Решим эту проблему в нашей игре с помощью класса `FullCanvas` из `Nokia API`.

Делается это не просто, а очень просто. Импортируем необходимый нам класс `com.nokia.mid.ui.FullCanvas` и наследуем наш класс `Snake` не от стандартного `Canvas`, а от расширенного `FullCanvas`.

```
import com.nokia.mid.ui.FullCanvas;
```

```
...
private class Snake extends FullCanvas implements Runnable {
    ...
}
```

Вот, собственно, и все. Класс `FullCanvas` сам наследован от стандартного класса `Canvas`, то есть содержит все его поля и методы, а это значит, что больше нам ничего переписывать не придется. Новых методов класса `FullCanvas` не реализует, зато определяет несколько дополнительных констант для клавиш, специфичных для телефонов Nokia. Например, константы `KEY_DOWN_ARROW`, `KEY_LEFT_ARROW`, `KEY_RIGHT_ARROW`, `KEY_UP_ARROW` могут быть использованы в процедуре `keyPressed(int keyCode)` для ре-

ализации управления джойстиком, характерным для аппаратов Nokia.

Итак, компилируем исправленное приложение. Обратим внимание на то, что в строке `Device` (Аппарат) главного окна `KToolbar` должен быть выбран эмулятор Nokia, иначе `WTK` не обнаружит API-класса и выдаст следующую ошибку: `package com.nokia.mid.ui does not exist`. Запускаем новую игру и видим, что мы, не особо напрягаясь, достигли желаемого результата и отвоевали у аппарата $128 \times 32 = 4096$ пикселей (рис. 13.2).

Следующий класс, который мы рассмотрим, — `DeviceControl`, из того же пакета `com.nokia.mid.ui`. Он дает возможность управления подсветкой и вибрацией телефона. В нашем распоряжении оказываются следующие методы:

- `void flashlights(long duration)` — мигание подсветки на протяжении `duration` миллисекунд. Метод возвращается сразу после вызова и не блокирует исполняющий тред на время выполнения эффекта;
- `void setLights(int num, int level)` — управление иллюминацией телефона. Аргумент `num` представляет номер источника света для аппаратов с дополнительными лампочками и подсветками. Фоновая подсветка имеет номер, равный нулю. Аргумент `level` задает уровень интенсивности света в диапазоне 0–100. Значение 0 означает выключение

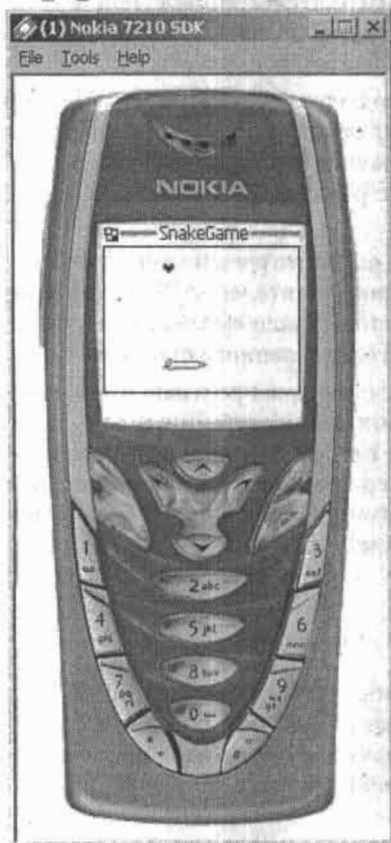


Рис. 13.1. Во время игры часть экрана оказывается недоступной

света. В аппаратах без поддержки уровня интенсивности света любое значение аргумента, большее нуля, означает включение света. Таким образом, команда погашения фоновой подсветки будет выглядеть так: `DeviceControl.setLights(0, 0);`

- `void startVibra(int freq, long duration)` — включает вибрацию телефона на `duration` миллисекунд. Аргумент `freq` задает частоту вибрации в диапазоне 0–100. Значение 0 может быть использовано для определения, поддерживается ли вибрация вообще. Если при нулевом аргументе будет сформировано исключение `IllegalStateException`, значит телефон в принципе не вибрирует. Такое же исключение при другом аргументе может быть сформировано, если система не разрешает использование вибрации. Метод, так же как и `flashLights`, возвращается сразу после вызова и не блокирует выполнение программы на время вибрации;
- `void stopVibra()` — остановить вибрацию, организованную предыдущим методом. Если вибрация не активна, то метод не выполняет никаких действий и не формирует исключений.

Посмотрим на примере, как можно использовать этот класс в нашей игре. Пусть в момент проигрыша телефон забьется в конвульсиях вместе со змеей, включив вибрацию. Первым делом импортируем необходимый класс `com.nokia.mid.ui.DeviceControl`. Затем в методе `paint(Graphics g)`, обрабатывающем действия конца игры, включим вибрацию максимальной частоты на 100 миллисекунд:

```
import com.nokia.mid.ui.DeviceControl;
...
public void paint(Graphics g) {
    ...
    // если поднят флаг конца игры
    if ( gameOverFlag ) {
        // включить вибрацию на 100мс
        DeviceControl.startVibra(100,100);
    }
    ...
}
```

Заметим, что эмулятор умело подражает даже этой функции аппарата, и телефон, изображенный на экране компьютера, заметно потряхивает.

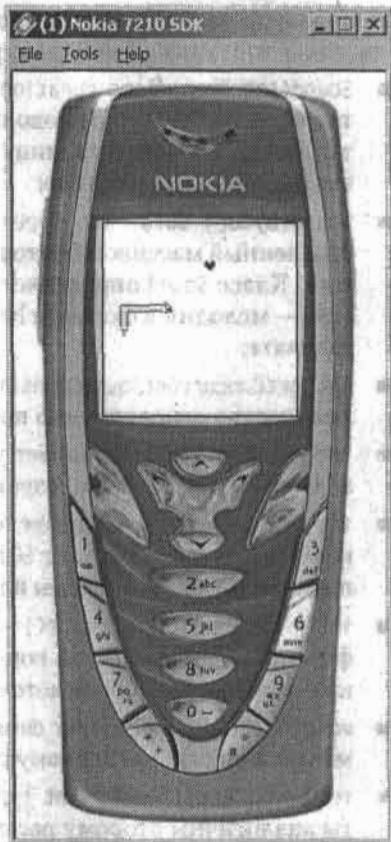


Рис. 13.2. Класс `FullCanvas` предоставляет в наше распоряжение весь экран целиком

Используем также для усовершенствования нашей игры класс `Sound` из пакета `com.nokia.mid.sound`. Для начала рассмотрим методы класса `Sound`:

- `Sound(int freq, long duration)` — конструктор; создает звуковой объект частотой сигнала `freq` герц, продолжительностью `duration` миллисекунд. В документации можно найти таблицу перевода частот в обычные ноты, хотя кому-то это одинаково непонятно;
- `Sound(byte[] data, int type)` — конструктор; создает звуковой объект, представленный массивом байтов `data`. Аргумент `type` задает формат звуковых данных. Класс `Sound` определяет две константы представления формата: `FORMAT_TONE` — мелодия в формате `Nokia`, `FORMAT_WAV` — мелодия в стандартном `MIDI`-формате;
- `int getConcurrentSoundCount(int type)` — возвращает максимально возможное количество одновременно проигрываемых мелодий типа `type`;
- `int getGain()` — возвращает уровень громкости звука в диапазоне 0–255 или значение –1, если используется громкость по умолчанию;
- `int getState()` — возвращает текущее состояние звукового объекта, представленное одной из трех констант: `SOUND_PLAYING`, `SOUND_STOPPED`, `SOUND_UNINITIALIZED` — звук проигрывается, остановлен или не инициализирован соответственно;
- `int[] getSupportedFormats()` — возвращает поддерживаемые аппаратом аудио-форматы в виде массива констант `FORMAT_TONE`, `FORMAT_WAV`. Если воспроизведение звука не поддерживается аппаратом, то возвращается пустой массив;
- `void init(int freq, long duration)` — инициализирует звуковой объект, аргументы аналогичны первому рассмотренному конструктору;
- `void init(byte[] data, int type)` — инициализирует звуковой объект, аргументы аналогичны второму рассмотренному конструктору;
- `void play(int loop)` — воспроизвести звуковой объект `loop` раз. Если аргумент `loop` принимает значение 0, то объект будет воспроизводиться до тех пор, пока не будет принудительно остановлен. Во время воспроизведения объект находится в состоянии `SOUND_PLAYING`;
- `void release()` — освобождает все используемые звуковым объектом ресурсы и переводит его в состояние `SOUND_UNINITIALIZED`. Если объект находится в состоянии `SOUND_PLAYING`, то сначала автоматически вызовется метод `stop()`;
- `void resume()` — возобновить воспроизведение остановленного звукового объекта. Если аппарат поддерживает такую возможность, то воспроизведение начнется с того места, где оно было остановлено;
- `void setGain(int gain)` — установить уровень громкости воспроизведения в диапазоне 0–255;
- `void stop()` — останавливает воспроизведение звукового объекта и переводит его в состояние `SOUND_STOPPED`.

Хотим теперь, чтобы при поедании очередного сердечка змея издавала радостный писк. Для этого нужно импортировать необходимый класс, создать звуковой объект и воспроизвести его в методе `checkMove`, контролирующем передвижение в заданном направлении:

```

import com.nokia.mid.sound.Sound;

...
// проверить возможность передвижения
// в точку с координатами (xH, yH)
public void checkMove(int xH, int yH) {
...
    // проверить совпадение координат передвижения
    // с координатами сердца
    if ( xH == xHeart && yH == yHeart ) {
        eatFlag = true;
        // создать звуковой объект
        // частоты 1000 герц длиной полсекунды
        Sound beep = new Sound(1000, 500);
        // воспроизвести звуковой объект
        beep.play(1);
        // увеличить скорость
        speed--;
    }
...
}

```

Осталось запустить приложение на эмуляторе и не забыть включить колонки. Вместо неприятного однотонного писка можно сделать симпатичный звуковой эффект, прогнав в цикле воспроизведение звуков разных частот, например вот таким образом:

```

// пройти в цикле частоты от 100 до 1000 герц с шагом 50
for(int i=100; i<=1000; i+=50) {
    // создать звуковой объект
    Sound beep = new Sound(i, 50);
    // воспроизвести звуковой объект
    beep.play(1);
}

```

Теперь у вас есть большое поле для экспериментов, чтобы начинить игру звуковыми и тактильными спецэффектами. Кроме использованных нами классов пакет `com.nokia.mid.ui` содержит также интерфейс `DirectGraphics` и класс `DirectUtils`, расширяющие возможности работы с графикой. Здесь нам предоставляется масса интересных методов, таких как прорисовка и заливка треугольников и полигонов, прямой доступ к любой точке экрана, повороты изображения, смена форматов цветовой гаммы и многие другие полезные вещи, интересные лишь Nokia-ориентированным программистам.

Samsung API

На остальных производителях мы не будем останавливаться настолько подробно. Рассмотрим, для сравнения, предоставляемые ими классы. Разработчики

телефонов Samsung предлагают пакет `com.samsung.util`, который содержит аналоги уже знакомых нам классов контроля звука, подсветки и вибрации, а также два новых класса, поддерживающих создание и рассылку SMS-сообщений. Рассмотрим вкратце все предложенные нам классы.

Класс `AudioClip` отличается от Nokia-аналога расширенным списком воспроизводимых форматов, а также отсутствием простой пищалки, задающей частоту тона в герцах. Конструктор `AudioClip(int type, byte[] audioData, int audioOffset, int audioLength)` создает звуковой объект из байтового массива `audioData`, начиная с позиции `audioOffset`, длины `audioLength` байтов. Второй конструктор `AudioClip(int type, String filename)` получает звуковой объект прямо из файла ресурса по имени `filename`. Аргумент `type` может принимать значения следующих констант: `TYPE_MMF`, `TYPE_MP3`, `TYPE_MIDI`, с оговоркой, что на данный момент аппаратами поддерживается только `TYPE_MMF`.

Методы `play(int loop, int volume)`, `stop()`, `pause()` и `resume()` служат для начала, прекращения, приостановки и возобновления воспроизведения звукового объекта соответственно. Метод `boolean isSupported()` возвращает значение `true`, если такая технология воспроизведения звука поддерживается аппаратом.

Класс `Vibration` заведует вибрацией телефона. Его метод `boolean isSupported()` аналогичен такому же методу класса `AudioClip`, а методы `start(int duration, int strength)` и `stop()` начинают и прекращают вибрацию соответственно.

Класс `LCDDLight` занимается лишь фоновой подсветкой, так как других источников света Samsung не предусматривает. Методы `boolean isSupported()`, `on(int duration)` и `off()` говорят о своих действиях сами за себя.

Рассылка SMS поддерживается в Samsung API двумя классами: первый класс, `SM`, содержит все необходимые методы для формирования сообщения, а второй, `SMS`, предоставляет лишь два метода: уже набивший оскомину `boolean isSupported()`, а также `send(SM sm)`, посылающий сообщение адресату. Конструктор `SM(String dest, String callback, String textMessage)` создает объект сообщения для номера `dest` от номера `callback` с текстом `textMessage`. Методы `String getCallbackAddress()`, `String getData()`, `String getDestAddress()`, `void setCallbackAddress(String address)`, `void setData(String textMessage)`, `void setDestAddress(String address)` предназначены для получения и установки трех параметров сообщения, рассмотренных в конструкторе. Обратим внимание на интересный параметр номера отправителя `callback`. Не хотят ли они сказать, что мы с телефона можем послать SMS от чужого имени? В документации говорится, что данный параметр не поддерживается, однако на форуме умельцев с Самсунгами я уже видел ссылку на подобную программку. Хотя не знаю, работает ли она: у меня, как вы уже, наверное, догадались, Nokia.

При попытке найти в Интернете эмуляторы телефонов Samsung, поддерживающие вышеописанные классы, обнаружилась занятная вещь под названием `Samsung JaUmi Wireless Toolkit`. Дизайн программки оказался до боли знакомым, за небольшим отличием списка эмулируемых аппаратов (рис. 13.3).

В меню `About` обнаружилась следующая надпись: «Copyright © SAMSUNG Electronics 2003. All rights reserved». Интересно ради заглянул в аналогичный пункт меню привычного нам `J2ME Wireless Toolkit`: «Copyright © 2002 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, U.S.A. All rights reserved». Видим, что,

оказывается, ничего зазорного в использовании чужого кода нету. А если серьезно, следите за соблюдением авторских прав в производимых и распространяемых вами программных продуктах.

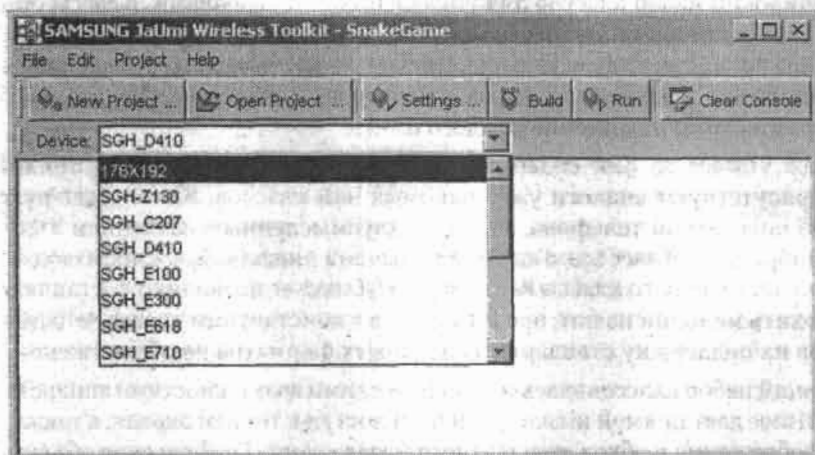


Рис. 13.3. Интерфейс этой программы очень напоминает J2ME Wireless Toolkit

Посмотрим, как будет себя чувствовать наша змейка на телефонах типа Samsung, предварительно заменив классы Nokia API на аналоги от Samsung, а Canvas вернув в исходное положение. Змейка демонстрирует свою живучесть и хорошую совместимость моделей (рис. 13.4).



Рис. 13.4. Самсунги тоже пригодны для обитания нашей змейки

Siemens API

Нужно отдать должное, разработчики компании Siemens переплюнули всех и дали самый широкий набор классов для управления их телефонами. Классы объединены аж в семь пакетов: `com.siemens.mp.game`, `com.siemens.mp.color_game`, `com.siemens.mp.gsm`, `com.siemens.mp.io`, `com.siemens.mp.media`, `com.siemens.mp.media.control`, `com.siemens.mp.ui`. Возможностей у этих классов масса, поэтому не будем вдаваться в подробности, а лишь рассмотрим назначение каждого из них.

Пакет `com.siemens.mp.game` содержит средства разработки игровых приложений. Здесь присутствуют аналоги уже знакомых нам классов. Класс `Light` руководит фоновой подсветкой телефона, а класс с двусмысленным названием `Vibrator` заведует вибрацией. Класс `Sound` является обычной пищалкой, воспроизводящей тон заданной частоты, зато классы `Melody` и `MelodyComposer` позволяют составлять и воспроизводить мелодии из нот, представленных константами класса `MelodyComposer`. Намеков на поддержку стандартных звуковых форматов не обнаружено.

Следующий набор классов заведует графическими возможностями аппарата. Класс `ExtendedImage` дает прямой низкоуровневый доступ к точкам экрана, а также прорисовку изображения в обход стандартного метода `paint`. Графические объекты представлены классом `GraphicObject`, являющимся базовым для классов `TiledBackground` и `Sprite`. Первый из них осуществляет поддержку графического фона экрана, представленного мозаикой.

Класс `Sprite` не имеет отношения к напитку, дающему смелые идеи, этот класс осуществляет поддержку графических объектов типа «спрайт». Спрайтом называется фигура или элемент экрана, который может быть перемещен независимо от остальных. Спрайт может содержать несколько кадров, сменяющих друг друга. С помощью класса `Sprite` реализуется простейшая анимация.

Следующие два пакета `com.siemens.mp.ui` и `com.siemens.mp.color_game` дают новый подход к управлению графическими объектами. Первый пакет содержит лишь один класс `Image`, расширяющий оригинал возможностями масштабирования и зеркального отражения картинки. Второй пакет организует работу с графическими объектами через базовый класс слоя графики `Layer`, от которого наследованы знакомые нам по прошлому пакету классы `Sprite` и `TiledLayer`, а также с помощью класса менеджера слоев `LayerManager`. Кроме этого пакет содержит класс `GameCanvas`, аналогично классу от Nokia, расширяющий стандартный `Canvas`.

Пакет `com.siemens.mp.gsm` предоставляет средства связи, представленные следующими классами: `PhoneBook` для доступа к адресной книге, `Call` для организации телефонного звонка по указанному номеру, `SMS` для формирования и рассылки SMS-сообщений.

Пакеты `com.siemens.mp.media` и `com.siemens.mp.media.control` реализуют универсальный аудиопроигрыватель с опциями воспроизведения, перемотки, возможностью прослушивания треков прямо из Интернета.

Последний пакет `com.siemens.mp.io` позволяет приложению организовать свою файловую систему с помощью класса `File`. Это позволит не мучаться с хранилищем записей, а использовать обычную файловую систему с привычными нам методами доступа. Интерфейс `ConnectionListener` и класс `Connection` реализуют воз-

возможность работы с внешними устройствами ввода-вывода телефона, такими как мультимедийные карты MMC или инфракрасный порт IRDA.

Звучит это все, конечно, заманчиво, осталось только узнать, какие модели поддерживают все эти прелести, а главное — по карману ли они нам. С точки зрения программного обеспечения Siemens опять оказался на высоте и предлагает свой собственный Siemens Mobile Toolkit (SMTK), который интегрируется со средой разработки JBuilder любой версии, начиная с пятой. В этой же программе можно установить различные эмуляторы телефонов. Змейка же продолжает успешно ползать по экранам разных моделей, теперь это Siemens M55 (рис. 13.5).

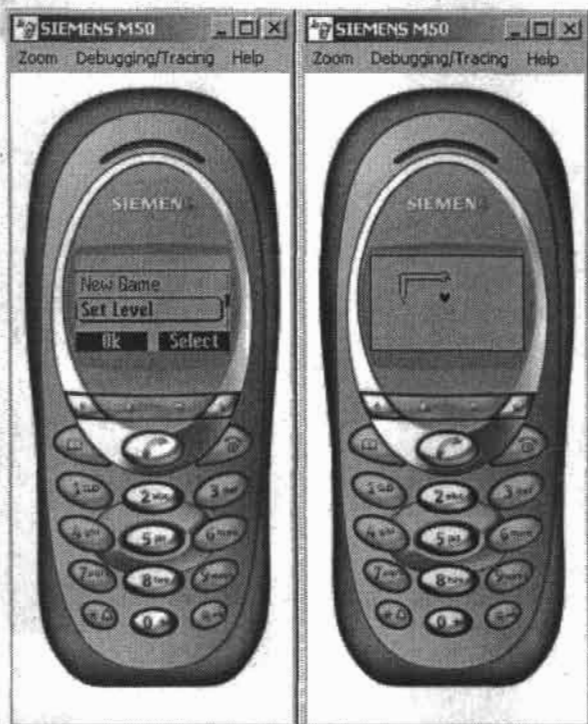
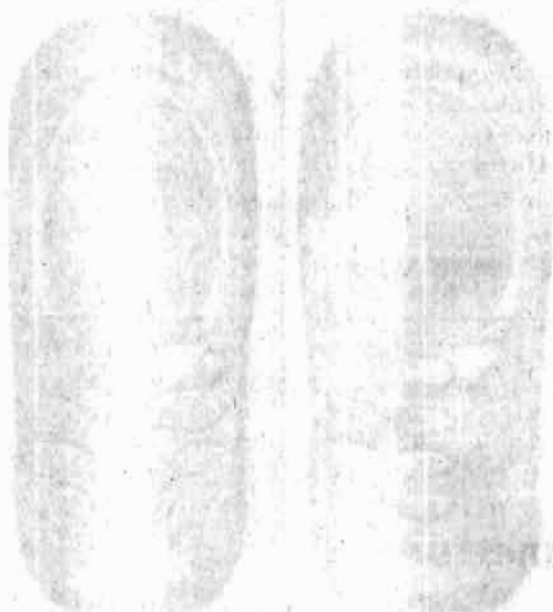


Рис. 13.5. И в Сименсах водятся змеи

На этом этапе мы прощаемся с нашей змейкой, так многому нас научившей. Полный листинг программы для аппарата Nokia приведен в конце книги, однако, надеюсь, что вы не принялись сразу аккуратно списывать код, а действовали вместе со мной поэтапно. Игра, конечно, далека от совершенства, поскольку реализована как демонстрация различных возможностей языка J2ME, в ваших силах теперь довести ее до ума, добавить необходимую логику, красочные картинки, проявив фантазию и смекалку.

По-хорошему, каждая модель телефона требует особой доработки программы, чтобы максимально задействовать возможности каждой конкретной модели. Про-

граммное обеспечение, API-классы, эмуляторы, документацию и прочие необходимые вещи можно скачать на официальных сайтах компаний-производителей. Как правило, делятся они этим безвозмездно. Также в Сети существует большое количество «фан-клубов» (сайтов, порталов, форумов) телефонов основных производителей, где собираются разные умельцы, которые делятся идеями и программами. Там вы сможете получить много интересной информации конкретно о вашем телефоне, а также скачать все необходимое программное обеспечение и даже больше. Перефразируем слова Мичурина: не нужно ждать милостей от телефона, взять их у него — наша задача!



Глава 14

Приложения для работы с Интернетом

Большая часть возможностей мобильного телефона и реализации J2ME уже позади. Мы написали несколько различных приложений, реализующих интеракцию с пользователем, хранение и обмен данными, а также другие функции языка. Мобильный телефон, однако, является все же средством связи и позволяет связываться не только с помощью голоса или SMS. Профайл MIDP предоставляет нам возможности связи с Интернетом, получения и передачи различного вида данных.

Поскольку речь в этой главе пойдет о взаимодействии телефона с Интернетом, то придется расширять сферу нашего обучения с мобильного программирования до разработки серверных приложений на других объектно-ориентированных языках. Тем, кто не знает, что такое сокет (socket), и не сталкивался с разработкой компьютерных приложений типа «клиент-сервер», придется потрудиться. Однако один из приведенных примеров, демонстрирующий работу по протоколу HTTP, если уж вы дошли до этого места, реализовать вам вполне по силам. Разработка приложений типа «клиент-сервер» — это отдельная наука. Мы же ограничимся самыми простыми примерами, демонстрирующими первоначальное взаимодействие телефона с Интернетом.

Класс Connector

Не буду забивать вам голову тем, как устроена связь между беспроводной сетью и Всемирной паутиной, а перейду сразу к делу, то есть к программированию. Сетевая работа и беспроводная коммуникация осуществляются в реализации MIDP средствами пакета `javax.microedition.io`, который содержит набор интерфейсов для различных типов соединений, а также единственный класс `Connector`. Огранизация соединений с любыми ресурсами осуществляется только через этот класс, который предоставляет следующие методы:

- `Connection open(String name)` — основной метод класса `Connector`, обеспечивающий открытие соединения с ресурсом, представленным аргументом `name`. Аргумент имеет формат универсального идентификатора ресурса (URI), который можно представить следующей схемой:

`<протокол>://<адрес>[<параметры>]`

Протокол представляет собой тип передаваемых данных. Реализация MIDP может поддерживать такие протоколы: `http`, `socket`, `datagram`, `file`, `ftp`, `comm`. Совсем не обязательно, чтобы ваша реализация MIDP поддерживала все эти

протоколы, поскольку спецификация MIDP требует обязательной поддержки только одного протокола HTTP 1.1. Не исключена также поддержка протоколов, не указанных в списке. Адресом является обычный URL; в качестве дополнительных параметров может выступать номер порта. Ниже приведены примеры создания различных типов соединений.

```
Connector.open("http://www.voolkansoft.com");  
Connector.open("socket://129.144.111.222:9000");  
Connector.open("datagram://129.144.111.333");  
Connector.open("file://foo.dat");  
Connector.open("comm:0;baudrate=9600");
```

Метод возвращает объект `Connection`, являющийся базовым для интерфейсов, представляющих различные типы соединений, которые мы рассмотрим позже;

- `Connection open(String name, int mode)` — метод, аналогичный предыдущему, с тем отличием, что параметр `mode` задает режим создаваемого соединения. Режимы соединений представлены следующими константами класса `Connector`:
 - `READ` — открыть соединение с доступом только на чтение данных;
 - `WRITE` — открыть соединение с доступом только на запись данных;
 - `READ_WRITE` — открыть соединение с доступом и на чтение, и на запись данных;
- `Connection open(String name, int mode, boolean timeouts)` — метод отличается от двух предыдущих наличием дополнительного параметра `timeouts` — флага, который разрешает или запрещает исключения от тайм-аутов.

Следующие методы класса `Connector` позволяют сразу открывать потоки ввода и вывода данных, рассмотренные в одной из предыдущих глав, минуя интерфейс `Connection`. Когда мы уже работали с потоками ввода и вывода, мы получали данные из файла, являющегося локальным ресурсом приложения. Вся разница заключается в том, что ресурс будет размещаться в Интернете и предоставлять данные в своем формате:

- `DataInputStream openDataInputStream(String name)` — открывает поток ввода данных. Аргумент `name` имеет формат уникального идентификатора ресурса;
- `DataOutputStream openDataOutputStream(String name)` — открывает поток вывода данных;
- `InputStream openInputStream(String name)` — открывает базовый поток ввода данных;
- `OutputStream openOutputStream(String name)` — открывает базовый поток вывода данных.

Интерфейс Connection

Интерфейс `Connection` является базовым для всех последующих типов соединений. Он содержит единственный метод `close()`, закрывающий соединение, которое открывается с помощью метода `open()` класса `Connector`. Этот интерфейс является самым абстрактным во всей структуре возможных типов соединений. Дальнейшие интерфейсы будут его расширять и дополнять новыми методами и возможностями.

Итак, процесс установки соединения и работы с ним выглядит следующим образом. Приложение использует класс `Connector` для открытия соединения с сетевым ресурсом. Метод `open(...)` класса `Connector` анализирует универсальный идентификатор ресурса и проверяет, поддерживается ли необходимый протокол в данной платформе. В случае ошибки формируется исключение, иначе возвращается объект `Connection`, который содержит ссылки на входной и выходной потоки к сетевому ресурсу. Приложение получает объекты типа `InputStream` и `OutputStream` из объекта `Connection` и организует обмен данными с сетевым ресурсом через эти потоки. При завершении работы приложение закрывает соединение с помощью метода `close()` интерфейса `Connection`.

Интерфейсы `InputConnection` и `OutputConnection`

Интерфейсы `InputConnection` и `OutputConnection` наследованы от интерфейса `Connection` и расширяют его потоками ввода и вывода соответственно. Каждый из них определяет по два новых метода для открытия потоков. Названия этих методов такие же, как и у методов открытия потоков класса `Connector`, и отличаются они лишь отсутствием аргументов. Использование этих методов открытия потоков мы еще увидим в примерах.

Интерфейс `StreamConnection`

Интерфейс `StreamConnection` объединяет в себе два предыдущих интерфейса `InputConnection` и `OutputConnection`, но не определяет никаких новых методов. Задача этого интерфейса в том, чтобы представить любой тип соединения, чьи данные могут быть обработаны как поток байтов. В зависимости от протокола соединения активизируются разные реализации интерфейса `StreamConnection`. Наша среда разработки J2ME WTK предоставляет две реализации: одну для соединения с портами связи, например с инфракрасным портом, вторую для соединения через сокеты. Мы рассмотрим второй тип соединения и реализуем пример, демонстрирующий работу соединения такого типа.

Сокет (`socket`) — это сетевой механизм транспортного уровня, который реализует работу по протоколу TCP/IP. Грубо говоря, это та дверь, которая связывает программу с Интернетом. Как любая порядочная дверь в большой мир, сокет имеет свой адрес, состоящий из сетевого адреса URL и номера порта, через который будет осуществляться соединение. Номер порта может быть любым в диапазоне 0–65 535, однако номера, меньшие чем 1024, обычно зарезервированы системой. Структура сокета универсальна, поэтому программа, принимающая соединение, может быть запущена на любой платформе и написана на любом языке, поддерживающем сокеты.

Итак, пришло время реализовать пример соединения, основанного на сокетах. Работа приложения будет заключаться лишь в том, чтобы соединиться с серверной частью, послать туда приветствие, получить в ответ сообщение и продемон-

стрировать его на экране телефона. Для начала рассмотрим клиентскую часть программы, написанную на уже привычном для нас языке J2ME:

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.io.StreamConnection;
import javax.microedition.io.Connector;
import java.io.IOException;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Display;
...
try {
    // открыть соединение через сокет с портом 3000 локальной машины
    StreamConnection sc =
        (StreamConnection)Connector.open("socket://127.0.0.1:3000");
    // получить поток ввода соединения
    DataInputStream dis = sc.openDataInputStream();
    // получить поток вывода соединения
    DataOutputStream dos = sc.openDataOutputStream();
    // записать приветствие в поток вывода
    dos.write("Hello, Server!\n".getBytes(), 0, 15);
    // буфер для приема сообщения
    byte[] buff = new byte[15];
    // получить сообщение из потока ввода
    dis.read(buff);
    // преобразовать сообщение в строку
    String msg = new String(buff);
    // создать форму для отображения сообщения
    Form form = new Form("Message received");
    // добавить полученное сообщение в форму
    form.append(msg);
    // получить менеджер дисплея
    Display display = Display.getDisplay(this);
    // отобразить форму сообщения
    display.setCurrent(form);
    // закрыть соединение
    sc.close();
} catch(IOException ioe) {
    // вывести исключение в область системных сообщений
    System.out.println("ERROR: " + ioe.getMessage());
}
```

С этой стороны нет ничего сложного. Приложение открывает соединение через сокет с портом 3000 локальной машины, представленной постоянным для любого компьютера IP-адресом 127.0.0.1. После установки соединения приложение получает потоки ввода и вывода соединения и через них реализует интеракцию с серверной частью.

Перейдем к тому, что представляет собой серверная часть. В нашем случае это программа, запущенная на локальном компьютере, открывающая сокет и ждущая связи с клиентом. Программу напомним на языке C++ с использованием классов MFC. С таким же успехом она может быть написана на любом языке, поддерживающем технологию сокетов.

Язык C++ не входит в нашу учебную программу, поэтому не будем особо вдаваться в подробности, а просто посмотрим на код некоторых функций, который может быть кому-то полезен. Рассмотрим две основные функции: `OnListen()`, которая открывает сокет, ожидающий клиентского соединения, и `OnReceive()`, которая обрабатывает событие получения сообщения. Функция отображаемого диалога `OnListen()` вызывается при нажатии на кнопку `Listen` основного диалога:

```
void CServerDlg::OnListen()
{
    // получить информацию из поля ввода порта
    UpdateData();
    // создать сокет с локальным адресом
    if (m_Sock->Create(m_Port, SOCK_STREAM, "127.0.0.1"))
    {
        // ждать связи с клиентом
        if (m_Sock->Listen()) {
            // вывести сообщение с номером порта сокета
            CStatic* stat = (CStatic*)GetDlgItem(IDC_STATIC);
            CString str;
            str.Format("Listening on port %d", m_Port);
            stat->SetWindowText(str);
            // вывести IP-адрес сокета
            stat = (CStatic*)GetDlgItem(IDC_MSG);
            UINT port;
            m_Sock->GetSockName(str, port);
            stat->SetWindowText(str);
        }
    }
}
```

Функция использует член класса диалога `m_Port`, связанный с полем ввода порта, и `m_Sock`, являющийся объектом класса `CListenSocket`, наследованный от стандартного MFC-класса `CSocket`. При нажатии на кнопку `Listen` диалог выдаст такие сообщения (рис. 14.1).

Вторая функция `OnReceive()` класса `CListenSocket` отслеживает момент приема сообщения через сокет, выводит его в окне диалога и посылает ответ клиентскому приложению:

```
void CListenSocket::OnReceive(int nErrorCode)
{
    // поля диалога для вывода сообщений
    CStatic* stat = (CStatic*)(m_pDlg->GetDlgItem(IDC_STATIC));
    CStatic* stat_msg = (CStatic*)(m_pDlg->GetDlgItem(IDC_MSG));
    // буфер приема сообщения
```

```
TCHAR buff[20];  
// прочитать сообщение в буфер  
Receive(&buff,20);  
// вывести подтверждение получения сообщения  
stat->SetWindowText(_T("Message Received..."));  
CString str;  
str.Format("%s",buff);  
// вывести полученное сообщение  
stat_msg->SetWindowText(str);  
// копировать ответ в буфер  
strcpy(buff,"Hello, MIDlet!\0");  
// послать ответное сообщение  
Send(&buff,15);  
}
```

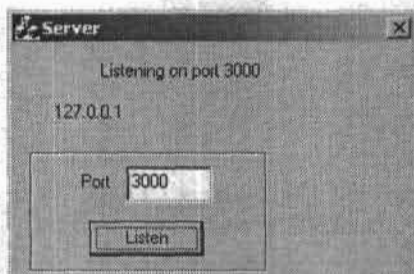


Рис. 14.1. Сервер сообщает, что слушает порт 3000 локальной машины

Те, кто уже знаком со студиями разработки C++, без труда смогут реализовать данный пример, остальные же без труда смогут найти книжки о разработке визуальных приложений на языке C++.

Перейдем к самому интересному, сначала запускаем серверную часть и ждем соединения клиента, в качестве которого будет выступать стандартный эмулятор среды разработки J2ME WTK. При запуске мобильного приложения сервер обнаружит связь и получит сообщение от телефона (рис. 14.2).

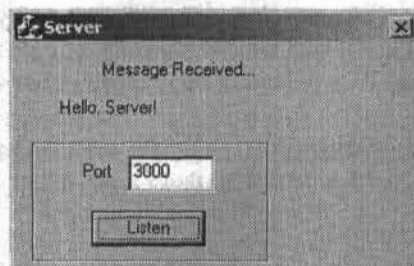


Рис. 14.2. Сервер получил сообщение и вывел его в окне

После приема сообщения сервер формирует в ответ свое приветствие и посылает его мобильному приложению, которое отображает принятое сообщение с помощью обычной формы (рис. 14.3).

Это был простейший пример, шапочное знакомство сервера с телефоном на уровне «привет-привет». Тем не менее, это и есть самый сложный момент. Если уж наши программы встретились и поздоровались, то и договориться смогут, а о чем им договариваться — зависит только от нашей фантазии, простор для которой здесь ничем не ограничен. Можно разработать информационную систему, многопользовательскую онлайн-игру, оперативный центр сбора и обработки информации и многое другое. Сервер — программа, размещенная в Интернете, — должна взять на себя все организационные функции общения с клиентами, которых может быть сколь угодно большое число, причем одновременно. В классических сервер-приложениях отдельный тред отводится для ожидания связи с новым клиентом, для каждого из которых выделяется собственный тред. Таким образом, клиенты не будут мешать друг другу, однако вопросы синхронизации при доступе к общим данным встают особо остро.

Заметим, что не все реализации поддерживают сокетные соединения, поэтому вполне возможна ситуация, что при попытке создания соединения на вашем аппарате будет сформировано исключение. Например, при попытке запуска того же самого приложения на эмуляторе от Nokia мы получим в системной области следующее сообщение: «ERROR: The requested protocol does not exist socket://127.0.0.1:3000».



Рис. 14.3. Эмулятор принял и отобразил приветствие сервера

Интерфейс StreamConnectionNotifier

Функции сервера может выполнять и мобильный телефон с помощью интерфейса StreamConnectionNotifier. Этот интерфейс содержит единственный метод `StreamConnection acceptAndOpen()`, который блокирует все операции до тех пор, пока не появится клиентский запрос на соединение. После чего метод создает новый объект соединения, через который и связывает серверную часть с клиентской.

При использовании данного интерфейса в качестве сервера будет выступать мобильное устройство. Недостатки такой реализации заключаются в том, что устройство должно быть постоянно на связи в ожидании подключения клиентов, что может быть дорого и неудобно.

Интерфейс DatagramConnection

Еще один тип соединений представлен интерфейсом DatagramConnection, который напрямую наследован от интерфейса Connection. Если соединение через сокеты основывалось на протоколе TCP, то данное соединение использует протокол передачи дейтаграмм UDP. Данный протокол используется различными интерне-

службами потоковых трансляций и основан на передаче дейтаграмм. В некоторых случаях он является более предпочтительным из-за эффективной и быстрой пересылки данных, поскольку там нет гарантированной доставки сообщений, исправлений искаженных сообщений и другого контроля, нагружающего связь.

Дейтаграмма представлена интерфейсом `Datagram`, который объединяет в себе два интерфейса `DataInput` и `DataOutput`. Кроме этого интерфейс содержит информацию о передаваемых или принимаемых данных и об адресе пункта назначения. Интерфейс `Datagram` предоставляет следующие методы:

- `String getAddress()` — установить адрес дейтаграммы;
- `byte[] getData()` — возвращает данные дейтаграммы;
- `int getLength()` — возвращает длину дейтаграммы;
- `int getOffset()` — возвращает текущее смещение указателя буфера данных;
- `void reset()` — восстанавливает позицию указателя буфера данных;
- `void setAddress(Datagram reference)` — устанавливает такой же адрес, как и у дейтаграммы, переданной в параметре;
- `void setAddress(String addr)` — устанавливает адрес дейтаграммы;
- `void setData(byte[] buffer, int offset, int len)` — устанавливает буфер данных, смещение указателя буфера и длину данных;
- `void setLength(int len)` — устанавливает длину полезной нагрузки дейтаграммы.

Интерфейс `DatagramConnection` позволяет отправлять и получать дейтаграммы с помощью методов `send(Datagram dgram)` и `receive(Datagram dgram)`. Также интерфейс позволяет создавать новые дейтаграммы с помощью нескольких методов:

- `Datagram newDatagram(byte[] buf, int size)` — возвращает новую дейтаграмму с данными буфера `buff` длины `size`;
- `Datagram newDatagram(byte[] buf, int size, String addr)` — аналогичен предыдущему методу, параметр `addr` задает адрес дейтаграммы;
- `Datagram newDatagram(int size)` — создает новую дейтаграмму заданного размера;
- `Datagram newDatagram(int size, String addr)` — создает новую дейтаграмму с заданным размером и адресом.

Максимально допустимую длину дейтаграммы, а также номинальную длину дейтаграммы можно получить с помощью методов `int getMaximumLength()` и `int getNominalLength()`.

Так же как и `StreamConnection`, интерфейс `DatagramConnection` может быть использован как для клиентской, так и для серверной частей приложения. Если при создании дейтаграммы в параметре адреса не будет указано имя хоста (`datagram://:3000`), то соединение будет работать в серверном режиме. Если имя хоста будет указано (`datagram://127.0.0.1:3000`), то соединение будет работать в клиентском режиме.

Интерфейс ContentConnection

Интерфейс `ContentConnection` дополняет интерфейс `StreamConnection`. Главное отличие заключается в том, что если раньше интерфейс не вникал в суть передава-

емых данных, то теперь из него можно извлечь некоторую информацию, определенную самим протоколом, с помощью следующих методов:

- `String getType()` — возвращает тип данных, передаваемых по протоколу. Для протокола HTTP метод возвратит содержимое поля `content-type` заголовка;
- `String getEncoding()` — возвращает кодировку данных. Для протокола HTTP возвращается содержимое поля `content-encoding` заголовка;
- `long getLength()` — возвращает длину доступных данных. Для протокола HTTP возвращается содержимое поля `content-length` заголовка. Если сведения о длине недоступны, то метод возвращает значение `-1`.

Интерфейс `HttpConnection`

Самым продвинутым интерфейсом во всей этой иерархии является интерфейс `HttpConnection`, представляющий соединения, использующие протокол HTTP. По этому протоколу организуется передача всех знакомых нам интернет-страниц.

Интерфейс `HttpConnection` реализует возможность получения информации полей HTTP-заголовка ресурса, а также поддерживает передачу запросов и получение откликов по протоколу HTTP. Кроме этого интерфейс определяет полный набор констант, представляющих коды ошибок и статуса протокола HTTP. Интеракция с веб-сервером осуществляется с помощью следующих методов интерфейса:

- `long getDate()` — возвращает дату из соответствующего поля заголовка. Дата по традиции представлена количеством миллисекунд, прошедших с 1 января 1970 года;
- `long getExpiration()` — выдает значение поля `expires` заголовка;
- `String getFile()` — возвращает имя файла из поля URL данного соединения;
- `String getHeaderField(int n)` — выдает значение поля заголовка с номером `n`;
- `String getHeaderField(String name)` — возвращает значение поля заголовка, определенного ключом `name`. Если заголовок не содержит поля с запрошенным именем, то возвращается значение `null`;
- `long getHeaderFieldInt(String name, long def)` — возвращает поле заголовка с именем `name`, представленное как целое число. Значение `def` возвращается, если поле с данным именем отсутствует или значение поля нельзя преобразовать в целое число;
- `long getHeaderFieldDate(String name, long def)` — аналогично предыдущему методу возвращает запрошенное поле заголовка, представленное в виде даты;
- `String getHeaderFieldKey(int n)` — возвращает имя поля заголовка с порядковым номером `n`;
- `String getHost()` — получить имя хоста соединения или IP-адрес, если имя хоста не задано;
- `long getLastModified()` — возвращает дату последней модификации ресурса, представленную полем `last-modified` заголовка;
- `int getPort()` — возвращает номер порта соединения; 80 — стандартный номер порта HTTP-соединений;

- `String getProtocol()` — получить имя протокола соединения;
- `String getQuery()` — выдает область запроса адреса URL. Эта часть начинается после знака `?` в строке адреса;
- `String getRef()` — выдает область адреса URL, следующую после знака `#`;
- `String getRequestMethod()` — возвращает установленный метод HTTP-запроса;
- `String getRequestProperty(String key)` — возвращает установленное свойство `key` HTTP-запроса;
- `int getResponseCode()` — получить код ответа сервера. Все возможные коды ответа представлены в интерфейсе константами `HTTP_ACCEPTED`, `HTTP_BAD_GATEWAY`, `HTTP_NOT_FOUND`, `HTTP_FORBIDDEN`, `HTTP_OK` и т. д. Все 38 констант приводить здесь не вижу смысла: кому понадобится, найдет их в документации;
- `String getResponseMessage()` — получить ответ сервера в виде строки с сообщением;
- `String getURL()` — возвращает адрес URL соединения;
- `void setRequestMethod(String method)` — установить метод HTTP-запроса. Допустимы методы, представленные следующими константами интерфейса: `GET`, `POST`, `HEAD`;
- `void setRequestProperty(String key, String value)` — установить значение `value` свойству `key` запроса.

Следующий пример, который мы реализуем, получит по протоколу HTTP данные интернет-страницы, а затем pošлет сообщение в гостевую книгу, размещенную в Интернете. В данном случае нам потребуется реализация лишь клиентской части приложения, поскольку серверная часть представлена обычным веб-сервером, с которым мы связываемся с помощью браузера, заходя на страницу в Интернете.

Для реализации нам потребуются некоторые знания самого протокола, а также языка HTML. Опыты мы будем производить над гостевой книгой официального сайта ФК «Томь», расположенной по адресу `football.tomsk.ru/gb`. Только лишь из большого уважения к этой команде, а не с целью как-то навредить. Для начала стоит взглянуть, как устроена гостевая книга внутри, благо на любой странице можно щелкнуть правой кнопкой и выбрать команду Просмотр в виде HTML, что позволит нам увидеть весь код страницы в текстовом виде. Немного поковырявшись, выясняем всю необходимую нам информацию: сообщения добавляются с помощью HTTP-запроса POST и скрипта `add.php`. Параметры сообщения передаются скрипту из формы в переменных `username`, `city`, `email`, `homepage` и `message`. Этой информации нам вполне достаточно, чтобы реализовать нашу затею.

Вот как будет выглядеть наше приложение, которое соединится с HTTP-сервером, отправит запрос с помощью метода POST, а затем получит различную информацию о сервере и параметрах соединения:

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.io.HttpConnection;
import javax.microedition.io.Connector;
import java.io.OutputStream;
import java.io.IOException;
```



```

...
try {
    // открыть соединение с HTTP-сервером
    HttpURLConnection conn =
        (HttpURLConnection)Connector.open("http://football.tomsk.ru/gb/add.php");
    // установить метод запроса
    conn.setRequestMethod("POST");
    // установить свойства запроса
    conn.setRequestProperty("Connection", "close");
    conn.setRequestProperty("User-Agent", "ConnectDemo (MIDP-1.0; J2ME)");
    conn.setRequestProperty("Content-Type",
        "application/x-www-form-urlencoded");
    conn.setRequestProperty("Accept", "text/plain");
    // получить поток вывода соединения
    OutputStream os = conn.getOutputStream();
    // записать параметры запроса POST в поток вывода
    os.write("username=Nokia7210&message=WeAreTheChampions".getBytes());
    // получить код ответа сервера
    System.out.println("Response Code: "+conn.getResponseCode());
    // получить текстовый ответ сервера
    System.out.println("Response Msg: "+conn.getResponseMessage());
    // получить базовые параметры соединения
    System.out.println("Type: "+conn.getType());
    System.out.println("Length: "+conn.getLength());
    System.out.println("Encoding: "+conn.getEncoding());
    // получить в цикле имена и значения полей заголовка
    for(int i=0;i<12;i++)
        System.out.println(conn.getHeaderFieldKey(i)+
            ": "+conn.getHeaderField(i));
    // получить и вывести параметры соединения
    System.out.println("Data: " + conn.getDate());
    System.out.println("Host: " + conn.getHost());
    System.out.println("Port: " + conn.getPort());
    System.out.println("Protocol: " + conn.getProtocol());
    System.out.println("Query: " + conn.getQuery());
    System.out.println("Ref: " + conn.getRef());
    System.out.println("Request Method: " + conn.getRequestMethod());
    System.out.println("URL: " + conn.getURL());
    // закрыть соединение
    conn.close();
}
catch(IOException ioe) {
    // вывести сообщение об исключении
    System.out.println("ERR:"+ioe.getMessage());
}

```

Если у вас есть выход в Интернет, то можно запускать приложение прямо на эмуляторе. Если вы используете прокси-сервер, то следует указать его адрес в свойствах эмулятора. После запуска смотрим в области системных сообщений J2ME WTK, насколько успешно мы соединились, анализируя ответы и параметры сервера (рис. 14.4).

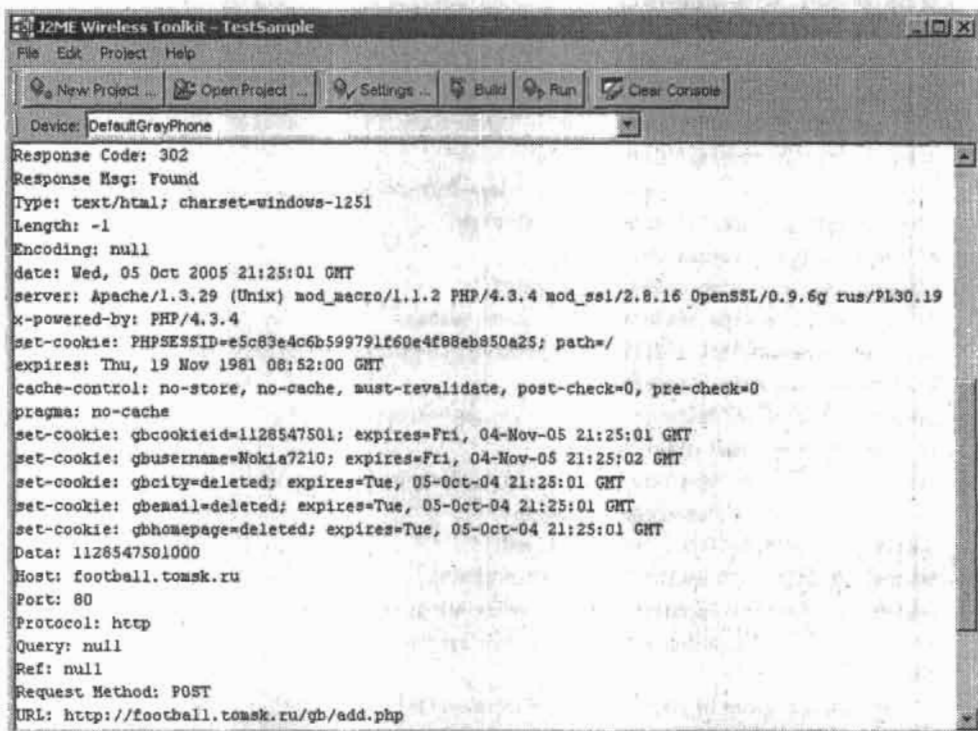


Рис. 14.4. В области системных сообщений выведена информация о соединении с сервером

Видим, что свойства сервера нами успешно получены. Заметьте, что для тестирования программы на реальном аппарате и для успешного соединения с интернет-ресурсом на телефоне должен быть активирован WAP-сервис. Теперь осталось зайти на гостевую книгу по вышеуказанной ссылке и посмотреть результат нашего эксперимента (рис. 14.5).

Интересна реализация следующей идеи: отправка SMS-сообщений с сайта сотовой компании, например www.mts.ru, что, возможно, даст некоторую экономию при отсылке SMS. Основная проблема заключается в том, что на сайте введена спам-защита в виде картинки с трудночитаемыми цифрами и буквами, которые надо ввести в специальном поле. Обойти это можно следующим способом: проанализировать HTML-текст сайта, получить ссылку на файл картинки, продемонстрировать ее на экране мобильного и попросить пользователя ввести пароль вручную. Здесь нас ожидает очередная проблема: формат картинки JPEG не совпадает с форматом PNG, поддерживаемым телефонами, поэтому для отображения следует сначала конвертировать картинку в нужный нам формат. Можно поискать готовый класс конвер-

тора (что-нибудь с именем JpegViewer, например), который можно встроить в приложение, или самому связаться с таблицами Хофмана и прочими премудростями структур графических файлов. Первый путь значительно проще, однако над реализацией такого приложения придется изрядно потрудиться. А вам слабо?

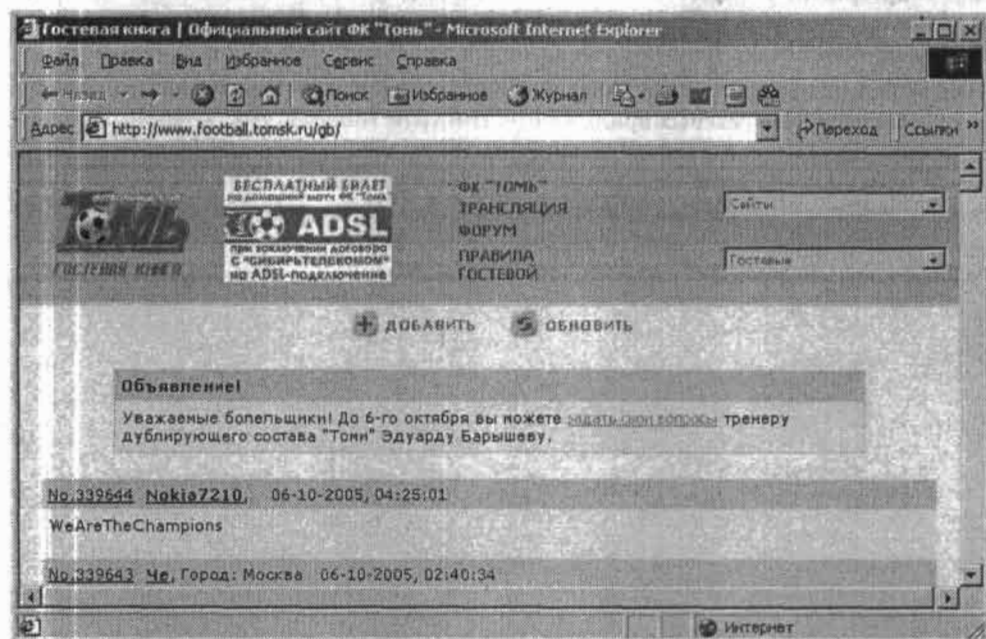


Рис. 14.5. В гостевой книге появилось сообщение от Nokia7210

Таким образом, полную иерархию всех рассмотренных типов соединений можно представить следующей схемой (рис. 14.6).

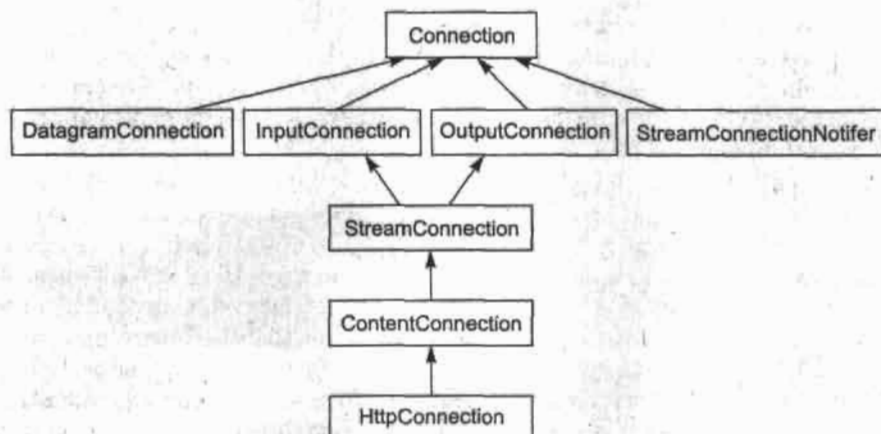


Рис. 14.6. Иерархия типов соединений

Мы рассмотрели все типы соединений этой иерархии, выделив четыре основных типа: потоковые соединения с коммуникационными портами, например IRDA, соединения уровня приложений по протоколу HTTP, либо через сокеты по протоколу TCP/IP, дейтаграммные соединения по протоколу UDP. Из всех этих типов спецификация MIDP определяет обязательную реализацию лишь для поддержки протокола HTTP. Попытка использования остальных типов соединений может сформировать исключение на вашей модели телефона.

Информация, приведенная в этой главе, является лишь базовыми сведениями для программирования сетевых приложений. Никто не говорит, что, потратив несколько минут на чтение, вы сможете написать интернет-браузер для вашего мобильного, однако, приложив некоторые усилия, организовать общение вашего телефона с интернет-ресурсами вам вполне по зубам. И да пребудет с вами сила!

Глава 15

Защита приложений

Если вы добрались до этого места книги, значит указанная тема должна быть для вас уже актуальна. Допускаю, что бессонными ночами, тернистым путем проб и ошибок, вы улучшали игру в змейку, довели ее до такого совершенства, что разработчики корпорации Nokia ахнули бы и обзавидовались. Допускаю также, что вы разработали уникальные алгоритмы для реализации игры, использовали нестандартные решения и феноменальные ходы.

Проблема заключается в том, что наряду с процессом компиляции, который переводит программу из текстового вида в машинные команды, существует обратный процесс декомпиляции, который позволяет восстановить исходный вид программы из конечного приложения. Конечно, есть альтруисты, которые практикуют «open source»-программирование и публикуют все исходники программ, но если мы хотим получать за наши труды хоть какое-то материальное вознаграждение, то попытки расковырять наш продукт нужно пресекать.

Java Decompiler

Посмотрим для начала врагу в лицо и разберемся, как получить текст чужой программы (конечно же, только с образовательной точки зрения). Брать чужое нехорошо, особенно извлекать из этого коммерческую выгоду, поэтому мы ограничимся лишь просмотром и редактированием в своих, сугубо личных целях. Можно адаптировать приложение для своей модели телефона, жизнью побольше поставить в игрушку или модернизировать полезную программку.

В первой главе мы рассматривали, как при помощи дата-кабеля и соответствующих программ закачать приложение в телефон. Аналогичным образом можно извлечь приложение из телефона. Далеко за примером ходить не будем и вытащим стандартный сборник шахматных задач «Chess Puzzle» из Nokia 6100 или чего-то похожего. Если вы уже благополучно его стерли или никогда и не имели, то найти это приложение в Интернете в свободном доступе не составит труда. Итак, вот он, заветный файл chesspuzzle.jar, и нам, как совсем юным мобильным программистам, ну очень интересно развинтить его и посмотреть, как он устроен. Вооружаемся отверткой типа WinRAR и смотрим, что же внутри у этого приложения (рис. 15.1).

Здесь мы наблюдаем два файла ресурсов lang.xx и Puzzles.BIN, папку с картинками icons и папку с манифестом приложения META-INF. Все самое интересное запрятано в глубине папки с названием com. В ней мы обнаружим несколько файлов с расширением *.class. Это файлы откомпилированных классов приложения. Вытащим их

из архива и сохраним у себя на компьютере. Внутри этих файлов лучше и не соваться: машинный язык доступен нам, простым смертным, лишь с переводчиком.

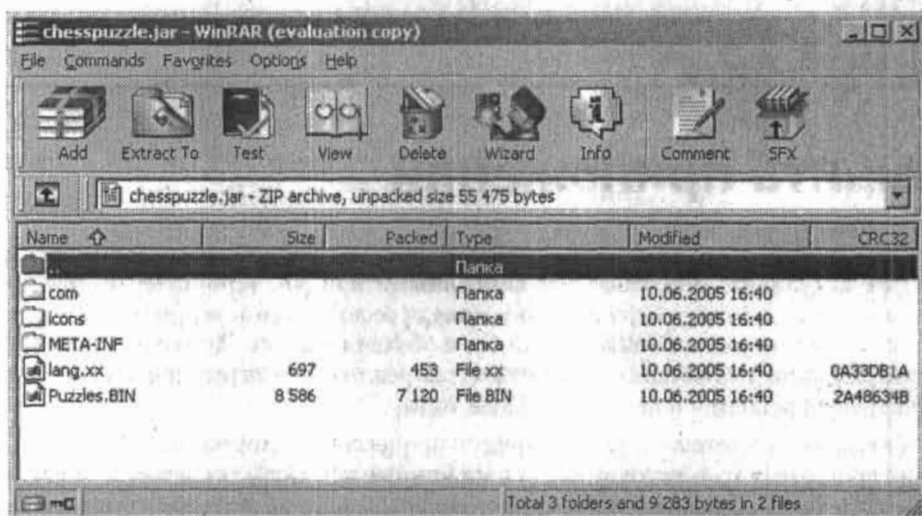


Рис. 15.1. Архив мобильного приложения изнутри

Теперь самое интересное. Простой отверткой нам уже не обойтись, нужны более мощные средства. Благодаря идеологии языка Java, связанной с транспортабельностью приложений, очень много информации можно восстановить из конечного приложения. Переводчик с машинного языка на привычный уже нам язык J2ME называется Java-декомпилятором (Java Decompiler). Думаю, вы без труда найдете программное обеспечение подобного рода. В рамках этой книги мы рассмотрим лишь один из возможных декомпиляторов под названием DJ Java Decompiler (по адресу dj.navexpress.com — информация о программе и разработчиках, а также свободное распространение последней версии).

Качаем, устанавливаем, запускаем. А теперь смело открываем основной класс приложения `ChessPuzzleMidlet.class` и видим вот такую чудесную картину (рис. 15.2). Создадим в обычном J2ME WTK проект с названием `ChessPuzzleMidlet`, декомпилируем все классы приложения и сохраним в папке исходного кода `/src` с расширением `.java`. Классы со знаком `$` в названии декомпилировать не нужно, поскольку они уже содержатся внутри класса, название которого предшествует знаку `$`. Файлы ресурсов `lang.txt`, `Puzzles.BIN` и картинки разархивируем и сохраним в папке ресурсов `/res` с сохранением структуры каталогов.

Обратим внимание, что все классы объединены в один пакет `package com.nokia.mid.app1.cher`, поэтому, чтобы компилятор обнаружил необходимые классы, нужно в свойствах созданного проекта заменить название основного класса мидлета на его полное имя, содержащее название пакета: `com.nokia.mid.app1.cher.ChessPuzzleMidlet`. Делается это следующим образом: в пункте меню **Project** (Проект) выбираем команду **Settings** (Параметры), где идем на вкладку **MIDlets** и редактируем поле **Class** (рис. 15.3).

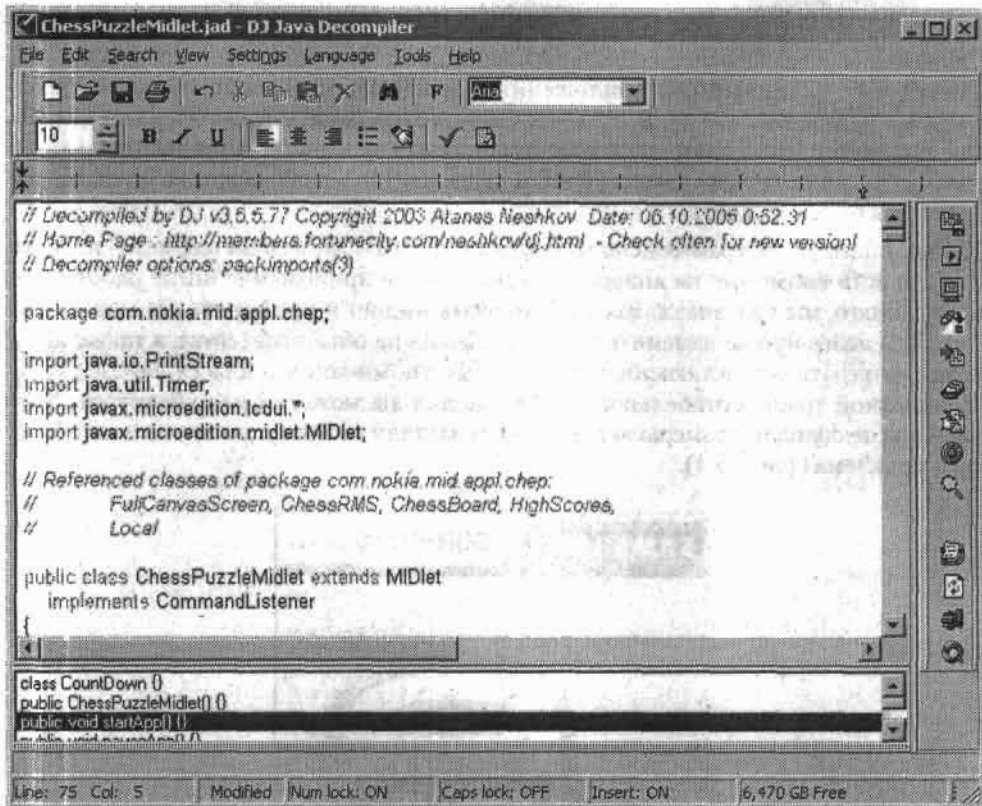


Рис. 15.2. Декомпилированное приложение снова доступно для нашего понимания

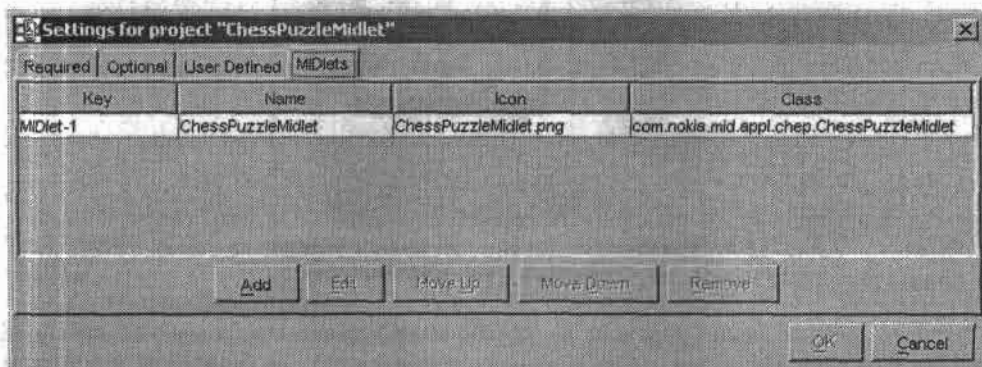


Рис. 15.3. Изменение названия основного класса мидлета

Пришло время попытаться откомпилировать наш проект. Здесь все, как в детстве с заводными игрушками: после того как разобрал и снова собрал, главное — чтобы опять заработало. Возможны небольшие различия в трактовке кода разными компиляторами. Мне пришлось добавить несколько приведений возвращаемых

типов, чтобы WTK перестал ругаться и выдавать ошибки. В остальном операция прошла успешно, и пациент ожил на эмуляторе в лучшем виде.

Теперь весь исходный код приложения в нашем полном распоряжении. Разбираться в чужой программе всегда не очень просто, однако ничего нового для нас мы не увидим, поскольку практически все классы и методы нами уже рассмотрены. Любое приложение начинается с метода `startApp()`, это и есть та печка, от которой мы будем плясать.

Как мы видим, в программе используются классы `FullCanvas` и `DeviceControl` из Nokia API, то есть кроме как на аппаратах Nokia данное приложение нигде работать не будет. Благо, мы уже знаем, как адаптировать мидлет и расширить его модельный ряд. Для этого нужно заменить класс `FullCanvas` на обычный `Canvas`, а также аккуратно вырезать всю иллюминацию и любые упоминания о классе `DeviceControl`. В успешной транспортабельности приложения вы можете сами убедиться. Хотя немного не совпали размеры экрана и часы заехали на доску, но это не такая большая проблема (рис. 15.4).



Рис. 15.4. Перекомпилированное приложение успешно запущено на телефоне другого производителя

Вопроса локализации (перевода на другой язык) приложений мы уже касались в одной из глав. В программе шахматных задач у нас есть возможность поучиться хорошему стилю программирования. Весь текст, который выдает приложение на английском языке, — инструкции, пункты меню и прочее — содержится в файле ресурса `lang.xx`. Все вопросы локализации решаются в одном классе `Local`. Рассмотрим процедуру получения требуемого текста:

```
public static synchronized String getText(int i, String as[]) {
    try {
```

```
if(loc == null)
    loc = new Local();
InputStream inputStream = loc.getClass().getResourceAsStream
    ("/lang." + phoneLang);

if(inputStream == null)
    inputStream = loc.getClass().getResourceAsStream("/lang.xx");
if(inputStream == null)
    return "NoLang";
DataInputStream dataInputStream = new DataInputStream(inputStream);
dataInputStream.skipBytes(i * 2);
short word0 = dataInputStream.readShort();
dataInputStream.skipBytes(word0 - i * 2 - 2);
String s = dataInputStream.readUTF();
dataInputStream.close();
if(as != null)
    if(as.length == 1) {
        s = replace(s, "%U", as[0]);
    } else {
        for(int j = 0; j < as.length; j++)
            s = replace(s, "%" + j + "U", as[j]);
    }
return s;
}
catch(Exception exception) {
    return "Err";
}
}
...
public static final String phoneLang =
    System.getProperty("microedition.locale");
```

Видим, что для перевода приложения на русский язык достаточно посмотреть код локализации телефона, возвращаемый методом `getProperty` системного класса `System`, и создать файл `lang` с полученным расширением. Ниже в классе `Local` определены числовые константы для всех выводимых строчек. По коду следует разобраться, каким образом проиндексирован файл ресурса. Сначала указаны номера позиций файла, с которых начинаются требуемые строчки, затем идут сами строчки. Осталось корректно сформировать файл с русским переводом. Возможно, потребуется дополнительная логика перевода в Unicode, которой мы пользовались в программе `BookReader`. Вот и вся локализация. Если вы внимательно читали книгу и выполняли упражнения, то вполне способны справиться с этой задачей самостоятельно.

Как вы уже поняли, все необходимое в наших руках, и можно воротить все, что душе угодно. Например, не нравится нам возможность обнуления установленного рекорда: что это за рекорд такой, который можно отменить одним нажатием. Дело за малым: найти место, где формируется меню `Top Scores`, а затем исключить

из него последний пункт `Reset`. В основном классе приложения видим, что любое меню формируется с помощью метода `InitMenuScr(int i, int ai[], ...)` с помощью следующих команд:

```
for(int k = 0; k < ai.length; k++)
    m_ListScr.append(Local.getText(ai[k]), null);
```

Видим, что пункты меню формируются из строчек, представленных константами, переданными методу в массиве `ai`. Теперь ищем вызов метода `InitMenuScr`, соответствующий формированию раздела меню `Top Scores`. Без труда обнаруживаем нужный вызов в методе `MainMenuScreenDone`:

```
...
case 4: // '\004'
    int ai3[] = {
        11, 10, 22
    };
    InitMenuScr(HIGHSCORE_MENU_SCR, ai3, Local.getText(25), null, null, 0);
    break;
...
```

В массиве `ai3` содержатся константы, представляющие три пункта меню: `Easy`, `Difficult` и `Reset`. Очевидно, что следует оставить только два первых элемента массива, а последний аккуратно удалить:

```
...
int ai3[] = { 11, 10 };
```

Ломать — не строить. Мы удалили всего лишь несколько символов. Главное — найти точку применения, а сами усилия могут быть мизерными. Компилируем и запускаем приложение. А теперь, как говорится, найдите десять отличий (рис. 15.5).



Рис. 15.5. Ликвидация пункта меню сводится к удалению из кода трех символов

Кроме бесценного опыта программистов от Nokia, что еще полезного можно извлечь из кода данного приложения? Решили вы, например, написать шахматы для

мобильного телефона, чтобы в дороге играть с товарищем. Кое-что можно будет использовать, например доску, фигурки, алгоритм перемещения фигур по доске и секундомер. Однако если вы решите продавать свои шахматы, то придется рисовать и программировать все заново, чтобы не нарушить авторские права. Преимущество заключается в том, что всегда можно проконсультироваться с готовым кодом и разобраться, как это делают профессионалы.

Защита приложений

Легальное распространение мобильных приложений происходит через WAP-сайты или SMS-сервис специализированных мобильных издательств, которые внимательно следят за авторскими правами и отчислениями разработчикам продуктов. При уведомлении об успешной загрузке приложения на ваш телефон система инициализации генерирует счет, который вы оплачиваете через вашего оператора сотовой связи. Удобно, надежно, а если что, можно обратиться в службу поддержки издательства: крупные и солидные конторы обычно заботливо относятся к своим клиентам.

Теперь допустим, что вы приобрели какое-то приложение легальным образом. Кто мешает скачать его с телефона на компьютер и выложить на общий доступ в Интернет? Кто после этого будет платить деньги, когда ту же самую игру можно получить бесплатно? Для защиты от подобного копирования можно ввести регистрацию: сам продукт распространяется бесплатно, а вот его активация или расширение пробной версии до полнофункциональной стоит денег. Вам потребуется реализовать алгоритм регистрации и организовать удобную систему оплаты. Как вариант, можно при первом запуске приложения сгенерировать случайным образом уникальный ключ и сохранить его в хранилище записей. Регистрационный механизм должен выслать код активации, соответствующий лишь данному ключу. При вводе кода активации приложение генерирует собственный код, соответствующий сгенерированному ключу, и при совпадении разблокирует необходимую функциональность.

Существует масса методов защиты и регистрирования, однако вернемся к началу этой главы. Казалось бы, что может быть проще: декомпилировать упрямое приложение, вырезать все проверки авторизации и запросы регистрационных кодов, а затем вернуть в исходное положение. Против таких умельцев, как мы, и действий, именуемых не иначе, как «взлом», существует способ защиты, называемый обфускацией. Не пугайтесь этого слова. Оно образовано от английского *obfuscate* — запутывать, сбивать с толку. Сам метод заключается в предварительной модификации исходного кода программы с целью затруднить процесс декомпиляции и анализа приложения.

Обфускация

Более всего распространена символьная обфускация, которая меняет имена переменных, названия классов, полей и методов на бессмысленный набор символов. Например, был у нас метод `GetPassword()`, а после обфускации он превратился в `LK23gdcasjJKHJh()`. Однако многие декомпиляторы уже справились с этим

подвохом и заменяют длинные нечитаемые имена на более удобные, типа `Method_1()`, `Method_2()` и т. п.

Существует и другой метод символьной обфускации, реализованный в J2ME WTK, работу которого мы рассмотрим на примере. Этот метод действует с точностью до наоборот, и переименовывают классы, поля и методы однобуквенными именами, повторяющимися в каждом классе. Согласитесь, чтобы понять вот такой пассаж:

```
b a = a.c(a, c.a);
```

потребуется немало усилий и тщательного изучения всей структуры программы, которая может быть совсем не маленькой.

Обфускатор — это не что иное, как утилита, модифицирующая исходный код приложения до неузнаваемости. К символьным относится также обфускатор, встроенный в J2ME WTK. Для того чтобы скомпилировать приложение с предварительной обфускацией, при упаковке проекта вместо пункта `Create Package` (Создать архив) следует выбрать команду `Create Obfuscated Package` (Создать архив с обфускацией). Попробуем в деле встроенный обфускатор на тех же шахматах.

Попробуем теперь декомпилировать получившиеся классы. Сразу бросается в глаза, что название сохранил лишь основной класс приложения, остальные называются теперь нехитро: `a`, `b`, `c`, `d`, `e`, `f`, ... Посмотрим на оригинал конструктора основного приложения. Так он выглядел до нашей обфускации:

```
public ChessPuzzleMidlet()
{
    m_Display = null;
    m_HighScoreArray = null;
    m_CurrentScreen = 0;
    m_LastScreen = 0;
    MAIN_MENU_SCR = 1;
    MODE_MENU_SCR = 2;
    PUZZLE_MENU_SCR = 3;
    HIGHSCORE_MENU_SCR = 4;
    SPLASH_SCR = 5;
    PUZZLE_SCR = 6;
    LEVEL_MENU_SCR = 7;
    HIGHSCORE_DIS_SCR = 8;
    HIGHSCORE_DEL_SCR = 9;
    INSTRUCTION_SCR = 10;
    TIMED_MODE = 1;
    CASUAL_MODE = 0;
    m_LastSelectedMenuIndex = 0;
    m_AppMessage = "";
    m_PlayMenu = false;
    m_ScreenWidth = 128;
    m_ScreenHeight = 128;
}
```


С такими именами констант и переменных разобраться в программе и внести в нее изменения нам не составило труда. Боюсь даже предположить, сколько бы это заняло времени и сил, если бы код выглядел следующим образом:

```
public ChessPuzzleMidget()
{
    d = null;
    flddo = null;
    ab = 0;
    s = 0;
    p = 1;
    W = 2;
    Y = 3;
    f = 4;
    i = 5;
    z = 6;
    aa = 7;
    T = 8;
    D = 9;
    ac = 10;
    _fldchar = 1;
    Q = 0;
    _fldcase = 0;
    P = "";
    ae = false;
    v = 128;
    m = 128;
}
```

Если и до этого разобраться в чужой программе было не так просто, то теперь это становится пыткой не для слабонервных, и поверьте, потраченное на расшифровку время уж точно не стоит двух-трех долларов.

Кроме символьной, существуют и другие методы обфускации, реализованные в обфускаторах второго порядка. Такие обфускаторы наряду с методами символьной обфускации используют также обфускацию данных и графа потока управления. Здесь есть большой простор для введения людей в заблуждение: изменение структур и местоположения данных, кодировка данных, объединение данных в массивы или наоборот, разделение массивов на отдельные переменные; добавление неиспользуемых данных и формирование на их основе ложных условий, замена статических присваиваний на вызовы методов; замена условий циклов на более сложные и масса других хитростей, об которые можно сломать мозги, пытаясь разобраться в зашифрованном алгоритме. Следует отметить, что если обфускаторы первого порядка немного сокращали размер программы, экономя на названиях, то второй порядок обфускации может разбазарить столь ценные нам килобайты.

Как вы сами понимаете, обфускация не является панацеей от взлома, будь то не-санкционированное расширение пробной версии или просмотр секретного алго-

ритма. Единственная задача обфускаторов — сбить взломщика с толку и максимально затруднить понимание, анализ и модификацию программы. В этой главе мы подошли к проблеме с двух сторон — декомпиляция и защита. Нам придется сталкиваться и с тем, и с другим. В любом случае, при компиляции своих программ не брезгуйте обфускаторами, даже если ваши программные изыски никого не интересуют, символьная обфускация немного оптимизирует вашу программу и освободит пару килобайтов для дополнительной функциональности или лишней картинке.

Как ни печально, но на этом этапе нам придется распрощаться. Я надеюсь, что чтение не прошло для вас даром, что вы многому научились и открыли для себя увлекательный мир мобильного программирования. Реализация приведенных примеров, их понимание и совершенствование — вот золотой ключик от дверей в чудо-мир мобильных технологий. Пробуйте, ошибайтесь, но не опускайте руки. Опыт приходит с практикой!

* * *

Эта книга не является учебником, документацией или серьезным методическим пособием, это именно самоучитель — практические советы, минимум теории и максимум практики, личные наблюдения автора, начиная с самого нуля. Если эта книжка уже вся исчеркана разными пометками, изрядно потрепана и залита кофе, значит потраченное мной и вами время не прошло даром.

Если вы действительно серьезно увлеклись мобильным программированием, то у вас неизбежно появится масса вопросов, благо на сайте разработчиков языка java.sun.com/j2me/docs/ имеется исчерпывающая документация, масса книг и статей (правда, на английском языке). Также за советом можно сходить на специализированные форумы, например, forum.juga.ru/, раздел которого так и называется: J2ME — форум о Java в мобильнике. Пользуйтесь поиском, не стесняйтесь спрашивать: программисты любят помогать друг другу.

Количество обладателей мобильных телефонов уже приблизилось к двум миллиардам (!) и неуклонно растет, и каждый из них — потенциальный пользователь ваших программ. Простота в использовании и всеобщая доступность в любое время — вот залог успеха. Будущее именно за мобильными, портативными технологиями, так что запрыгивайте в поезд, пока не поздно. В качестве бонус-трека вас ожидают практические советы по продаже мобильных приложений.

Удачи вам!

Bonus

Распространение мобильных приложений. Практические советы по продажам мобильного контента

Теперь у нас есть все необходимое для разработки игр и приложений для мобильных телефонов: аппарат с поддержкой J2ME, кабель для соединения с компьютером, необходимое программное обеспечение, а главное — знания и практические навыки.

Если вы уже начинили свой телефон самодельными играми и приложениями, поделились ими с друзьями, похвастали перед коллегами, то можно сделать и следующий шаг: попытаться продать свои труды. И пусть вас не смущает, что программка вышла коротенькая, а разработка заняла не более нескольких дней: специфика программирования для мобильных устройств играет сейчас нам на руку.

Игры для мобильных телефонов существенно отличаются от обычных компьютерных игр, которые пишут целые команды разработчиков, крупные фирмы с большим штатом сотрудников, а потому написать что-то конкурентоспособное на этом рынке одному человеку просто не под силу.

Ограниченная функциональность платформы, жесткие требования к размеру приложения не позволяют писать зрелищные, интерактивные побоища типа Quake. Маленькое 64-килобайтное тельце программы не способно вместить логику серьезной военно-экономической стратегии, да и не стоит велосипеду тягаться с гоночной машиной. Игра для мобильного и не должна быть сложной и навороченной. Обычно она заполняет небольшую паузу ввиду доступности в любое время и в любом месте.

Красивая, элегантная, аккуратно реализованная идея имеет хорошие шансы принести создателю некоторую прибыль. Остановимся на процедуре получения денег за свой программный продукт, схеме движения финансовых потоков и, вообще, на том, как это работает.

Допустим, что вы реализовали какое-то нехитрое, но оригинальное приложение, например шуточный мобильный тест на беременность, с предложением приложить палец к обведенному месту на экране и с какой-то вероятностью получить один из заготовленных ответов. Если вы считаете, что данная программа может принести вам несколько долларов, то можно смело попытаться ее продать. В любом случае, хуже не будет, ведь денег за эту попытку с вас никто не возьмет. Главный вопрос: кто и каким образом будет заниматься продажей? Для этого и существуют фирмы, которые специализируются на продажах мобильного контента. За определенный

процент они размещают ваш продукт на своей платформе, рекламируют его, осуществляют всю техническую поддержку, связанную с процедурой продажи.

Для начала нужно связаться с представителем такой фирмы и заключить договор о передаче прав на продажу, где и будет оговорена сумма, которую вы получите с каждой проданной копии программы.

Обычно договор не передает фирме все права на продукт, а только предоставляет право распространения. То есть договор не должен препятствовать вам заключать соглашения параллельно с несколькими фирмами.

Устоявшаяся цена в России на момент написания этой книги, которую платит конечный пользователь за скачанную игру или программу, — 2 доллара. Теперь посмотрим, какая часть от этой суммы окажется у вас в кармане. Деньги за приложение снимаются со счета пользователя и поступают оператору сотовой связи. Оператор, как правило, половину этой суммы забирает себе, а половину переводит на счет фирмы-распространителя.

На этом этапе фирма имеет свой интерес и делит с вами прибыль согласно договору. При заключении договора не стесняйтесь торговаться в пределах разумного. Здесь ставка по России колеблется в пределах 30–50 центов в пользу разработчика за каждую проданную копию программы.

Очевидно, что повышение вашего благосостояния напрямую зависит от числа людей, скачавших программный продукт. Дело за малым: заинтересовать пользователя полетом своей фантазии, оригинальностью, новизной, практической пользой. Здесь уже все в ваших руках.

Особое внимание следует уделить соблюдению авторских прав, ответственность за нарушение которых лежит целиком на разработчике. Когда речь идет о коммерческом использовании, последствия нарушений могут быть очень серьезными, и наступают они куда с большей вероятностью, чем при установке операционной системы с пиратского диска. Так что использование текстов, картинок, программного кода не должно идти вразрез с чьими бы то ни было интересами.

Чтобы не быть голословными, приведем конкретные примеры и адреса наиболее крупных в России фирм-распространителей контента для мобильных телефонов: www.playmobile.ru, www.playfon.ru, www.infon.ru, www.javagames.ru. В любом случае, такие фирмы совсем не трудно найти в поисковиках по любому запросу с ключевыми словами данной тематики. Серьезные фирмы отличаются качественно выполненным сайтом, ежедневно обновляемыми новостями, наличием менеджеров, готовых оперативно выйти на связь.

Если вы попробовали свои силы на российском рынке мобильного программирования и чувствуете высокий коммерческий потенциал вашей программы, то можно шагнуть и за границу. Здесь также существуют давно налаженные схемы распространения программ, например через сайт forum.nokia.com/business.

В современном мире сотовый телефон стал неотъемлемым атрибутом нашей жизни. Человек без мобильного вызывает сейчас не меньшее удивление, чем лет 10 назад человек с мобильником. Именно поэтому технология чрезвычайно перспективна. Каждый третий человек на планете — обладатель мобильного телефона, а значит, и потенциальный потребитель программного продукта. Всех благ!

Приложение 1

Полный листинг программы «Змейка»

```
//-----  
// Пример игры "Змейка" для телефона Nokia  
//  
// Реализован в рамках книги  
// "Программируем игры для мобильных телефонов"  
//  
// (c) Voolkan  
//-----  
  
import javax.microedition.midlet.MIDlet;  
  
import javax.microedition.lcdui.Canvas;  
import javax.microedition.lcdui.ChoiceGroup;  
import javax.microedition.lcdui.Command;  
import javax.microedition.lcdui.CommandListener;  
import javax.microedition.lcdui.Display;  
import javax.microedition.lcdui.Displayable;  
import javax.microedition.lcdui.Form;  
import javax.microedition.lcdui.Gauge;  
import javax.microedition.lcdui.Graphics;  
import javax.microedition.lcdui.Image;  
import javax.microedition.lcdui.ImageItem;  
import javax.microedition.lcdui.Item;  
import javax.microedition.lcdui.List;  
import javax.microedition.lcdui.StringItem;  
  
import javax.microedition.rms.RecordStore;  
import javax.microedition.rms.RecordEnumeration;  
import javax.microedition.rms.RecordStoreException;  
  
import java.util.Vector;  
import java.util.Random;  
  
import java.io.IOException;  
  
import com.nokia.mid.ui.FullCanvas;
```

```
import com.nokia.mid.ui.DeviceControl;
import com.nokia.mid.sound.Sound;
```

// основной класс SnakeGame, реализующий сценарий смены экранов

```
public class SnakeGame extends MIDlet implements CommandListener {
```

```
    private Display display;           // менеджер дисплея
    private Snake curSnake;           // объект змейки
    private List menu;                 // стартовое меню
    private Command ok;                // команда выбора пункта меню
    private ChoiceGroup levelChoice;   // меню выбора уровня сложности игры
    private Command set;               // команда выбора уровня сложности игры
```

```
    public void destroyApp(boolean destroy) {
        curSnake=null;
        notifyDestroyed();
    }
```

```
    public void pauseApp() {}
```

// стартовый метод мидлета

```
    public void startApp() {
        // массив строк с названиями пунктов меню
        String menuOptions[] = {"New Game", "Set Level", "High Score"};
        // создать стартовое меню
        menu = new List("", List.IMPLICIT, menuOptions, null);
        // создать команду выбора пункта меню
        ok = new Command("Ok", Command.OK, 1);
        // добавить команду в меню
        menu.addCommand(ok);
        // создать команду установки уровня игры
        set = new Command("Set", Command.BACK, 1);
        // установить блок прослушивания команд для меню
        menu.setCommandListener(this);
        // получить ссылку на менеджер дисплея
        display = Display.getDisplay(this);

        // отобразить на экране стартовую заставку
        Image title = null;
        try {
            // создать объект заставки
```



```
title = Image.createImage("/title.png");
} catch (IOException ioe) {}

// создать новый элемент формы
ImageItem item = new ImageItem("", title, ImageItem.LAYOUT_CENTER, "");
// отобразить заставку
showNewScreen("SNAKE", ok, item);
}

// блок прослушивания команд
public void commandAction(Command c, Displayable d) {
    // если была команда Ok из главного меню игры
    if (c == ok && d == menu) {
        // получить выбранный пункт меню
        int selIndex = menu.getSelectedIndex();
        switch(selIndex) {
            // первый пункт меню: новая игра
            case 0 :
                // создать объект змейки
                curSnake = new Snake();
                // создать объект треда автоматического передвижения
                Thread moveThread = new Thread(curSnake);
                // начать выполнение треда
                moveThread.start();
                // отобразить змейку на экране
                display.setCurrent(curSnake);
                break;
            // второй пункт меню: установить уровень сложности
            case 1 :
                // массив строк с названиями пунктов меню
                String levelOptions[] = {"Level 1", "Level 2",
                                         "Level 3", "Level 4"};
                // создать меню выбора уровня игры
                levelChoice = new ChoiceGroup("", List.EXCLUSIVE,
                                              levelOptions, null);
                // выделить пункт меню, соответствующий
                // текущему уровню сложности
                levelChoice.setSelectedIndex(5 - getParameter(1), true);
                // отобразить новую форму
                showNewScreen("Set Level", set, levelChoice);
                break;
            // третий пункт меню: посмотреть текущий рекорд
            case 2 :
                // получить строку с текущим рекордом
```

```

        String strScore = (new Integer(getParameter(0))).toString();
        // создать объект элемента-строки
        StringItem highScoreItem = new StringItem("", strScore);
        // отобразить новую форму
        showNewScreen("HIGH SCORE", ok, highScoreItem);
        break;
    }
}

// если была команда Ok, вызванная не из меню
if(c==ok && d!=menu)
    // отобразить на экране стартовое меню
    display.setCurrent(menu);

// если была команда Set из меню установки уровня сложности
if(c == set) {
    try {
        // массив параметров игры
        byte buff[] = {0,5};
        // открыть хранилище записей с именем "SNAKE"
        RecordStore recordStore = RecordStore.openRecordStore("SNAKE",
                                                                true);

        // получить список записей хранилища
        RecordEnumeration re = recordStore.enumerateRecords(null, null,
                                                                false);

        // если хранилище не пусто
        if (re.numRecords()!=0) {
            // получить id записи параметров игры
            int id = re.nextRecordId();
            // считать запись с параметрами игры
            buff = recordStore.getRecord(id);
            // вычислить текущий уровень игры
            buff[1] = (byte)(5 - levelChoice.getSelectedIndex());
            // записать параметры игры
            recordStore.setRecord(id, buff, 0, 2);
        } else {
            // добавить новую запись параметров игры
            recordStore.addRecord(buff, 0, 2);
        }
    } catch(RecordStoreException rse) {
    }
    // отобразить на экране стартовое меню
    display.setCurrent(menu);
}

```

```

    }

    // метод showNewScreen создает и отображает форму с элементом item
    private void showNewScreen(String title, Command c, Item item) {
        // создать форму для нового экрана
        Form newScreenForm = new Form(title);
        // добавить команду для возврата из формы
        newScreenForm.addCommand(c);
        // добавить элемент в форму
        newScreenForm.append(item);
        // установить блок прослушивания команд
        newScreenForm.setCommandListener(this);
        // отобразить форму на экране
        display.setCurrent(newScreenForm);
    }

    // метод getParameter считывает из хранилища записей параметры игры
    // и возвращает один из них, определенный входным параметром index
    private byte getParameter(int index) {
        // массив параметров игры
        byte buff[] = {0,0};
        try {
            // открыть хранилище записей с именем "SNAKE"
            RecordStore recordStore = RecordStore.openRecordStore("SNAKE",
                                                                    true);

            // получить список записей хранилища
            RecordEnumeration re = recordStore.enumerateRecords(null, null,
                                                                false);

            // если хранилище не пусто
            if (re.numRecords() != 0) {
                // получить id и считать запись параметров игры
                buff = recordStore.getRecord(re.nextRecordId());
            }
        } catch (RecordStoreException rse) {
        }
        // вернуть параметр
        return buff[index];
    }

    // класс SnakePart представляет каждый элемент змейки
    private class SnakePart extends Object {

        private int x;        // координата x
    }

```

```

private int y;      // координата y
private int part;   // код части тела змеи
private int dir;    // код направления движения

// конструктор; принимает все поля как аргументы
public SnakePart(int _x, int _y, int _part, int _dir) {
    x=_x;
    y=_y;
    part=_part;
    dir=_dir;
}

// получить код части тела змеи
private int getPart() { return part; }
// получить код направления движения
private int getDir() { return dir; }
// получить координату x
private int getX() { return x; }
// получить координату y
private int getY() { return y; }
// установить тип части и направление
private void setPartDir(int _part, int _dir) {
    // если второй аргумент -1,
    // то направление остается неизменным
    part=_part;
    if(_dir!=-1) dir=_dir;
}

}

// класс змейки
private class Snake extends FullCanvas implements Runnable {

// константы, определяющие направление движения
private int UP = 0;      // вверх
private int DOWN = 3;    // вниз
private int LEFT = 1;    // влево
private int RIGHT = 2;   // вправо

private int HEAD = 0;    // голова змеи
private int TAIL = 1;    // хвост змеи
private int BODY = 2;    // тело змеи
private int ACLOCKWISE_TURN = 3; // сгиб тела против часовой стрелки
private int CLOCKWISE_TURN = 4;  // сгиб тела по часовой стрелке

private Vector snake;     // хранилище элементов змейки

```

```

private int width;        // ширина экрана
private int height;       // высота экрана
private int xHead;        // координата x головы змейки
private int yHead;        // координата y головы змейки
private Image heart;       // картинка сердца
private int xHeart;       // координата x сердца
private int yHeart;       // координата y сердца
private Image[] images;   // массив картинок элементов змейки
private int imageSize;    // размер картинки элемента змейки

private int direction;     // текущее направление змейки
private boolean eatFlag;   // флаг удлинения змейки
private boolean gameOverFlag; // флаг конца игры

private RecordStore recordStore; // хранилище данных
private int recordID;         // ID записи параметров
private byte level;           // уровень сложности игры
private byte speed;           // коэффициент вычисления задержки треда
private byte highScore;       // текущий рекорд

// конструктор класса Snake
public Snake() {
    // конструктор родительского класса
    super();
    // создать массив для картинок-частей змейки
    images = new Image[20];
    // создать объекты картинок для каждой части
    try {
        // голова во всех направлениях
        images[0] = Image.createImage("/HeadUp.png");
        images[1] = Image.createImage("/HeadLeft.png");
        images[2] = Image.createImage("/HeadRight.png");
        images[3] = Image.createImage("/HeadDown.png");
        // хвост во всех направлениях
        images[4] = Image.createImage("/TailUp.png");
        images[5] = Image.createImage("/TailLeft.png");
        images[6] = Image.createImage("/TailRight.png");
        images[7] = Image.createImage("/TailDown.png");
        // тело
        images[8] = Image.createImage("/BodyUp.png");
        images[9] = Image.createImage("/BodyLeft.png");
        images[10] = images[9];
        images[11] = images[8];
        // сгиб тела против часовой стрелки
    }
}

```

```
images[12] = Image.createImage("/TurnDownLeft.png");
images[13] = Image.createImage("/TurnUpLeft.png");
images[14] = Image.createImage("/TurnDownRight.png");
images[15] = Image.createImage("/TurnUpRight.png");
// сгиб тела по часовой стрелке
images[16] = images[14];
images[17] = images[12];
images[18] = images[15];
images[19] = images[13];
// сердце
heart = Image.createImage("/heart.png");
}
catch (IOException ioe) {}

imageSize = images[0].getWidth();
// создать вектор, хранящий элементы змейки
snake = new Vector();
// получить ширину экрана
width = getWidth();
// получить высоту экрана
height = getHeight();
// установить координаты головы в центре экрана
// для транспортабельности координаты кратны imageSize
xHead = ((width/2)/imageSize)*imageSize;
yHead = ((height/2)/imageSize)*imageSize;
// создать элемент головы змейки
SnakePart partS = new SnakePart(xHead,yHead,HEAD.UP);
// добавить элемент в вектор
snake.addElement(partS);
// создать элемент тела змейки
partS = new SnakePart(xHead,yHead+imageSize,BODY.UP);
snake.addElement(partS);
// создать элемент хвоста змейки
partS = new SnakePart(xHead,yHead+2*imageSize,TAIL.UP);
snake.addElement(partS);
// установить текущее направление движения
direction = UP;
// получить новые координаты для сердца
setNewHeart();
// сбросить флаг окончания игры
gameOverFlag = false;

// параметры для долговременного хранения
// первый байт - текущий рекорд
// второй байт - уровень сложности игры
```



```

byte buff[] = {0.5};
try {
    // открыть хранилище записей с именем "SNAKE"
    recordStore = RecordStore.openRecordStore("SNAKE", true);
    // получить список записей хранилища
    RecordEnumeration re = recordStore.enumerateRecords(null, null,
                                                         false);
    // если хранилище пусто
    if (re.numRecords() == 0)
        // добавить новую запись параметров игры
        recordID = recordStore.addRecord(buff, 0, 2);
    else
        // получить id записи параметров игры
        recordID = re.nextRecordId();
    // считать запись параметров игры
    buff = recordStore.getRecord(recordID);
    // первый байт: текущий рекорд
    highScore = buff[0];
    // второй байт: уровень сложности игры
    level = buff[1];
    // установить коэффициент вычисления задержки
    speed = 100;
} catch (RecordStoreException rse) {
}
}

// метод перерисовки экрана
public void paint(Graphics g) {
    // очистить экран
    g.setColor(0xFFFFFFFF);
    g.fillRect(0, 0, width, height);
    g.setColor(0x000000);
    // нарисовать рамку
    g.drawRect(0, 0, width-1, height-1);
    // получить длину змейки
    int snakeLen = snake.size();
    // если поднят флаг конца игры
    if ( gameOverFlag ) {
        // включить вибрацию на 100 мс
        DeviceControl.startVibra(100, 100);
        // вывести сообщение конца игры
        g.drawString("GAME OVER", width/2, height/2-10, g.HCENTER | g.TOP);
        // строка сообщения счета игры
    }
}

```

```

String scoreStr;
// массив долговременных данных
byte buff[] = new byte[2];
// если установлен новый рекорд
if(highScore < snakeLen) {
    highScore = (byte)snakeLen;
    // инициализировать массив новыми параметрами
    buff[0] = highScore;
    buff[1] = level;
    try {
        // записать новый рекорд в хранилище
        recordStore.setRecord(recordID,buff,0,2);
    } catch (RecordStoreException rse) {
    }
    // установить цвет текста красным
    g.setColor(0xff0000);
    // сформировать строку с новым рекордом
    scoreStr = new String("HIGH SCORE: "+snakeLen);
// если рекорд не установлен
} else
    // сформировать строку со счетом
    scoreStr = new String("YOUR SCORE: "+snakeLen);
// вывести счет
g.drawString(scoreStr,width/2,height/2+10,g.HCENTER|g.TOP);
// если флаг конца игры не поднят
} else {
    // для каждого элемента змейки
    for ( int i=0; i<snakeLen; i++) {
        // вывести картинку, индекс которой вычисляется
        // с помощью комбинации кода части тела и кода направления
        g.drawImage(images[4*((SnakePart)snake.elementAt(i)).getPart()+
            ((SnakePart)snake.elementAt(i)).getDir()],
            ((SnakePart)snake.elementAt(i)).getX(),
            ((SnakePart)snake.elementAt(i)).getY(),
            g.HCENTER | g.VCENTER);
    }
    // вывести картинку сердца
    g.drawImage(heart.xHeart,yHeart,g.HCENTER | g.VCENTER);
}
}

// метод setNewHeart: получает новые координаты для сердца
public void setNewHeart() {
    // получить количество возможных позиций по координате x

```

```

int xPartMaxNum = (width-5)/imageSize;
// получить количество возможных позиций по координате y
int yPartMaxNum = (height-5)/imageSize;
// получить длину змеи
int len = snake.size();
// создать объект генератора случайных чисел
Random rnd = new Random();
// флаг корректной генерации
boolean setFlag = false;
// повторяем, пока не сгенерируем корректные координаты сердца
while ( !setFlag ) {
    // поднять флаг корректной генерации
    setFlag = true;
    // сгенерировать случайную координату x
    xHeart = imageSize*(Math.abs(rnd.nextInt())%xPartMaxNum+1);
    // сгенерировать случайную координату y
    yHeart = imageSize*(Math.abs(rnd.nextInt())%yPartMaxNum+1);
    // проверить совпадение координат сердца
    // с координатами элементов змеи
    for ( int i=0; i<len-1; i++ )
        if ( xHeart == ((SnakePart)(snake.elementAt(i))).getX() &&
            yHeart == ((SnakePart)(snake.elementAt(i))).getY() )
            // опустить флаг корректной генерации
            setFlag = false;
}
}

// метод keyPressed обработки нажатий клавиш
public synchronized void keyPressed(int keyCode) {
    // если флаг конца игры не поднят
    if ( !gameOverFlag ) {
        switch (keyCode) {
            // проверить, можно ли передвигаться в заданном направлении
            // и вызвать соответствующую функцию передвижения
            case KEY_NUM2:
                checkMove(xHead, yHead-imageSize);
                moveUp();
                break;
            case KEY_NUM4:
                checkMove(xHead-imageSize, yHead);
                moveLeft();
                break;
            case KEY_NUM6:
                checkMove(xHead+imageSize, yHead);

```

```

        moveRight();
        break;
    case KEY_NUM8:
        checkMove(xHead, yHead+imageSize);
        moveDown();
        break;
    }

    // вызвать перерисовку экрана
    repaint();
} else {
    // GameOver! отобразить стартовое меню
    display.setCurrent(menu);
}
}

// метод автоматического продвижения змейки
public void run() {
    // пока не поднят флаг конца игры
    while (!gameOverFlag) {
        // синхронизировать с управлением
        synchronized (this) {
            // продвинуть змейку в текущем направлении
            if(direction==UP) {checkMove(xHead, yHead-7); moveUp();}
            if(direction==LEFT) {checkMove(xHead-7, yHead); moveLeft();}
            if(direction==RIGHT) {checkMove(xHead+7, yHead); moveRight();}
            if(direction==DOWN) {checkMove(xHead, yHead+7); moveDown();}
        }
        // вызвать перерисовку экрана
        repaint();
        try {
            // приостановить тред на sleep*level миллисекунд
            Thread.sleep(level*speed);
        } catch (InterruptedException e) {
        }
    }
}

// проверить возможность передвижения
// в точку с координатами (xH,yH)
public void checkMove(int xH, int yH) {
    // для каждого элемента змеи
    for ( int i=3; i<snake.size()-1; i++ ) {

```

```

// если проверяемая точка совпадает с координатами элемента
if ( xH == ((SnakePart)(snake.elementAt(i))).getX() &&
    yH == ((SnakePart)(snake.elementAt(i))).getY() ) {
    // поднять флаг конца игры
    gameOverFlag=true;
}

// проверить совпадение координат передвижения
// с координатами сердца
if ( xH == xHeart && yH == yHeart ) {
    // поднять флаг удлинения змейки
    eatFlag = true;
    // пройти в цикле частоты от 100 до 1000 герц с шагом 50
    for(int i=100; i<=1000; i+=50) {
        // создать звуковой объект
        Sound beep = new Sound(i,50);
        // воспроизвести звуковой объект
        beep.play(1);
    }
    // увеличить скорость
    speed--;
}
else
    // опустить флаг удлинения змейки
    eatFlag = false;
}

// передвижение влево
private void moveLeft() {
    // если текущее направление направо, то возврат
    if ( direction==RIGHT ) return;
    // сдвинуть координату головы влево
    xHead-=imageSize;
    // проверить выход за левую границу экрана
    if ( xHead <= imageSize/2+1 ) { gameOverFlag=true; return; }
    // получить первый элемент змеи
    SnakePart head = (SnakePart)(snake.firstElement());
    // если текущее направление вверх
    if ( direction==UP )
        // заменить голову на змею против часовой стрелки
        head.setPartDir(ACLOCKWISE_TURN,LEFT);
    else {
        // если текущее направление вниз
        if ( direction==DOWN )

```

```

        // заменить голову на сгиб по часовой стрелке
        head.setPartDir(CLOCKWISE_TURN.LEFT);
    else // если текущее направление влево
        // заменить голову на тело
        head.setPartDir(BODY.LEFT);
    }
    // создать новый элемент для головы
    SnakePart sPart = new SnakePart(xHead,yHead,HEAD.LEFT);
    // добавить голову первым элементом вектора
    snake.insertElementAt(sPart,0);

    // если флаг удлинения змеи поднят
    if (eatFlag) {
        // получить новые координаты для сердца
        setNewHeart();
    }
    else {
        // продвинуть хвост.
        // удалить последний элемент
        snake.removeElement(snake.lastElement());
        // назначить хвостом предпоследний элемент
        - ((SnakePart)snake.lastElement()).setPartDir(TAIL,-1);
    }
    // установить текущее направление
    direction = LEFT;
}

// передвижение вправо
private void moveRight() {
    // если текущее направление налево, то возврат
    if ( direction==LEFT ) return;
    // сдвинуть координату головы вправо
    xHead+=imageSize;
    // проверить выход за правую границу экрана
    if ( xHead >= width-imageSize/2 ) { gameOverFlag=true; return; }

    // если текущее направление змейки вверх.
    // заменить голову на сгиб по часовой стрелке
    if ( direction == UP ) ((SnakePart)(snake.firstElement())).
        setPartDir(CLOCKWISE_TURN.RIGHT);
    else {
        // если текущее направление змейки вниз.
        // заменить голову на сгиб против часовой стрелки
        if ( direction == DOWN ) ((SnakePart)(snake.firstElement())).

```



```

        setPartDir(ACLOCKWISE_TURN,RIGHT);
        // продвижение по прямой, заменить голову на тело
        else ((SnakePart)(snake.firstElement())).setPartDir(BODY,RIGHT);
    }

    // добавить новый элемент головы змейки
    SnakePart sPart = new SnakePart(xHead,yHead.HEAD,RIGHT);
    snake.insertElementAt(sPart,0);
    // если поднят флаг удлинения
    if (eatFlag)
        // получить новые координаты для сердца
        setNewHeart();
    else {
        // продвинуть хвост
        snake.removeElement(snake.lastElement());
        ((SnakePart)snake.lastElement()).setPartDir(TAIL,-1);
    }
    // установить текущее направление движения
    direction = RIGHT;
}

// передвижение вверх
private void moveUp() {
    // если текущее направление вниз, то возврат
    if ( direction == DOWN ) return;

    // сдвинуть координату головы вверх
    yHead-=imageSize;
    // проверить выход за верхнюю границу экрана
    if ( yHead <= imageSize/2+1 ) { gameOverFlag=true; return; }

    // если текущее направление змейки влево,
    // заменить голову на сгиб по часовой стрелке
    if ( direction==LEFT ) ((SnakePart)(snake.firstElement())).
        setPartDir(CLOCKWISE_TURN,UP);
    else {
        // если текущее направление змейки направо,
        // заменить голову на сгиб против часовой стрелки
        if ( direction==RIGHT ) ((SnakePart)(snake.firstElement())).
            setPartDir(ACLOCKWISE_TURN,UP);
        // продвижение по прямой, заменить голову на тело
        else ((SnakePart)(snake.firstElement())).setPartDir(BODY,UP);
    }

    // добавить новый элемент головы змейки

```

```

SnakePart sPart = new SnakePart(xHead,yHead,HEAD,UP);
snake.insertElementAt(sPart,0);

// если поднят флаг удлинения
if (eatFlag)
    // получить новые координаты для сердца
    setNewHeart();
else {
    // продвинуть хвост
    snake.removeElement(snake.lastElement());
    ((SnakePart)snake.lastElement()).setPartDir(TAIL,-1);
}
// установить текущее направление движения
direction = UP;
}

// передвижение вниз
private void moveDown() {
    // если текущее направление вверх. то возврат
    if ( direction==UP ) return;
    // сдвинуть координату головы вниз
    yHead+=imageSize;
    // проверить выход за нижнюю границу экрана
    if ( yHead >= height-imageSize/2) { gameOverFlag=true; return; }

    // если текущее направление змейки влево,
    // заменить голову на сгиб против часовой стрелки
    if ( direction==LEFT) ((SnakePart)(snake.firstElement())).
        setPartDir(ACLOCKWISE_TURN,DOWN);
    else {
        // если текущее направление змейки направо,
        // заменить голову на сгиб по часовой стрелке
        if ( direction==RIGHT) ((SnakePart)(snake.firstElement())).
            setPartDir(CLOCKWISE_TURN,DOWN);
        // продвижение по прямой, заменить голову на тело
        else ((SnakePart)(snake.firstElement())).setPartDir(BODY,DOWN);
    }
}

// добавить новый элемент головы змейки
SnakePart sPart = new SnakePart(xHead,yHead,HEAD,DOWN);
snake.insertElementAt(sPart,0);

// если поднят флаг удлинения
if (eatFlag)

```

```
// получить новые координаты для сердца
setNewHeart();
else {
    // продвинуть хвост
    snake.removeElement(snake.lastElement());
    ((SnakePart)snake.lastElement()).setPartDir(TAIL, -1);
}
// установить текущее направление движения
direction = DOWN;
}
} // class Snake
} // class SnakeGame
```

Приложение 2

Полный листинг программы «BookReader»

```
//-----  
// Пример программы "BookReader"  
//  
// Реализован в рамках книги  
// "Программируем игры для мобильных телефонов"  
//  
// (c) Voolkan  
//-----  
  
import javax.microedition.midlet.MIDlet;  
import javax.microedition.lcdui.*;  
import java.util.Stack;  
  
import java.io.InputStream;  
import java.io.EOFException;  
import java.io.IOException;  
  
public class BookReader extends MIDlet {  
  
    private Display display;    // менеджер дисплея  
    private BookCanvas fCanvas; // экранный фрагмент книги  
  
    public void destroyApp(boolean flag) {  
    }  
  
    public void pauseApp() {  
    }  
  
    // стартовый метод мидлета  
    public void startApp() {  
        // получить ссылку на менеджер дисплея  
        display = Display.getDisplay(this);
```

```
// создать новую форму с именем основного класса
Form form = new Form(getClass().getName());
// создать объект экранного фрагмента книги
fCanvas = new BookCanvas();
// отобразить экранный фрагмент книги
display.setCurrent(fCanvas);
}
```

```
// класс экранного фрагмента книги
```

```
public class BookCanvas extends Canvas {
```

```
    private Stack PageIndex; // стек смещений страниц от начала текста
```

```
    public BookCanvas() {
```

```
        // создать новый объект стека номеров страниц
```

```
        PageIndex = new Stack();
```

```
    }
```

```
// метод перерисовки объекта экранного фрагмента
```

```
protected void paint(Graphics g) {
```

```
    // получить ширину и высоту рабочей области экрана
```

```
    int gw = g.getClipWidth();
```

```
    int gh = g.getClipHeight();
```

```
    // создать шрифт
```

```
    Font font = Font.getFont(Font.FACE_MONOSPACE,
```

```
                             Font.STYLE_PLAIN,
```

```
                             Font.SIZE_SMALL);
```

```
    // установить шрифт
```

```
    g.setFont(font);
```

```
    // инициализировать поток
```

```
    InputStream is = getClass().getResourceAsStream("/story.txt");
```

```
    MyDataInputStream mdis = new MyDataInputStream(is);
```

```
    int x=0,y=0; // текущее положение вывода на экране
```

```
    int offset=0; // смещение по исходному тексту
```

```
    // очистка экрана
```

```
    g.setColor(255,255,255);
```

```
    g.fillRect(0,0,gw,gh);
```

```
    g.setColor(0,0,0);
```

```
    // получить смещение для очередной страницы
```

```
    if(!PageIndex.empty())
```

```

offset=((Integer)PageIndex.peek()).intValue();

try{
    // сместиться по тексту до необходимой страницы
    mdis.skipBytes(offset);
} catch(IOException ioe) {}

String sWord;
do {
    // прочитать очередное слово из потока
    sWord = mdis.readWord();
    // продвинуть смещение по файлу на одно слово
    offset+=sWord.length();
    // если текущая позиция и пиксельная ширина слова
    // не выходят за экран
    if(x+font.stringWidth(sWord)<=gw) {
        // отобразить слово, начиная с текущей позиции
        g.drawString(sWord, x,y, g.TOP|g.LEFT);
        // сместить текущую позицию на пиксельную ширину слова
        x+=font.stringWidth(sWord);
        // если слово последнее в строке, перевести текущую позицию
        // в начало новой строки на дисплее
        if (mdis.b_endLine) { y+=font.getHeight(); x=0; }
    }
    // слово не входит по ширине с текущей позиции
    else {
        // перейти на новую строку
        y+=font.getHeight();
        // если на экране есть место для новой строки,
        // отобразить слово с начала строки
        if(y+font.getHeight()<gh)
            g.drawString(sWord, 0,y, g.TOP|g.LEFT);
        // перевести текущую позицию на ширину слова
        x=font.stringWidth(sWord);
    }
    // повторять до тех пор, пока есть место
    // для новой строки или не кончится файл
} while(y+font.getHeight()<gh && !mdis.b_endFile);

int index;    // смещение начала страницы

// если цикл закончился чтением слова, которое не было
// отображено на экране, сместить начало новой страницы
// назад на одно слово
if(mdis.b_endLine) index=offset; else index=offset-sWord.length();

```

```
// поместить в стек начало следующей страницы
if(!mdis.b_endFile) PageIndex.push(new Integer(index));
// закрыть поток
try{
    mdis.close();
} catch(IOException ioe) {}
}
```

```
// метод обработки нажатий клавиш
public void keyPressed(int keyCode){
    // обработка нажатий клавиш
    switch(keyCode) {
        case KEY_NUM1:
            // удалить из стека начала
            // следующей и текущей страницы
            PageIndex.pop();
            if(!PageIndex.empty()) PageIndex.pop();
            repaint();
            break;
        case KEY_NUM2:
            // перерисовать экран
            // (начало следующей страницы уже в стеке)
            repaint();
            break;
        case KEY_NUM0:
            // очистить стек и перерисовать экран
            PageIndex.removeAllElements();
            repaint();
            break;
    }
}
} // class BookCanvas
} // class BookReader
```

```
import java.io.InputStream;
import java.io.DataInputStream;
import java.io.EOFException;
import java.io.IOException;
```

```
public class MyDataInputStream extends DataInputStream {
```

```
// кодовая строка преобразования символов
private String WIN1251_TO_UNICODE =
```


1

Приложение 3

Полный листинг программы «AdressBook»

```
//-----  
// Пример программы "AdressBook"  
//  
// Реализован в рамках книги  
// "Программируем игры для мобильных телефонов"  
//  
// (c) Voolkan  
//-----  
  
import javax.microedition.midlet.MIDlet;  
import javax.microedition.lcdui.Command;  
import javax.microedition.lcdui.CommandListener;  
import javax.microedition.lcdui.Display;  
import javax.microedition.lcdui.Displayable;  
import javax.microedition.lcdui.Form;  
import javax.microedition.lcdui.List;  
import javax.microedition.lcdui.TextField;  
import javax.microedition.lcdui.DateField;  
import javax.microedition.lcdui.TextBox;  
import java.util.Date;  
import java.util.Calendar;  
import java.io.ByteArrayOutputStream;  
import java.io.ByteArrayInputStream;  
import java.io.DataOutputStream;  
import java.io.DataInputStream;  
import java.io.IOException;  
import javax.microedition.rms.RecordStore;  
import javax.microedition.rms.RecordEnumeration;  
import javax.microedition.rms.RecordComparator;  
import javax.microedition.rms.RecordFilter;  
import javax.microedition.rms.RecordStoreException;  
  
public class AddressBook extends MIDlet implements CommandListener  
{
```

```
private Display display;           // менеджер дисплея
private RecordStore recordStore;   // хранилище записей
private List nameList;             // список имен
private int recIndexes[];          // массив ID записей,
                                   // соответствующий списку имен
private Command add,ok,next,back;  // команды переходов между экранами
private TextBox tbName,tbPhone,tbEMail; // экраны ввода параметров записи
private Form dateForm;             // форма ввода дня рождения
private DateField dateField;       // поле ввода дня рождения
```

```
public void destroyApp(boolean destroy) {
    notifyDestroyed();
}
```

```
public void pauseApp() {}
```

```
public void startApp() {
    try {
        // открыть хранилище записей с именем "Address-Book"
        recordStore = RecordStore.openRecordStore("Address-Book", true);
    } catch (RecordStoreException rse) {}
}
```

```
// получить ссылку на менеджер дисплея
display = Display.getDisplay(this);
```

```
// создание объектов команд
ok = new Command("Ok", Command.OK, 1);
add = new Command("Add", Command.BACK, 1);
next = new Command("Next", Command.OK, 1);
back = new Command("Back", Command.BACK, 1);
```

```
// поле ввода имени
tbName = new TextBox("Name:","",15,TextField.ANY);
tbName.addCommand(next);
tbName.addCommand(back);
tbName.setCommandListener(this);
```

```
// поле ввода номера телефона
tbPhone = new TextBox("Number:","",15,TextField.PHONENUMBER);
tbPhone.addCommand(next);
tbPhone.addCommand(back);
tbPhone.setCommandListener(this);
```

```
// поле ввода электронной почты
tbEMail = new TextBox("E-Mail:","",35,TextField.EMAILADDR);
```

```
tbEMail.addCommand(next);
tbEMail.addCommand(back);
tbEMail.setCommandListener(this);

// экран ввода дня рождения
dateField = new DateField("Birthday", DateField.DATE);
dateForm = new Form("");
dateForm.append(dateField);
dateForm.addCommand(next);
dateForm.addCommand(back);
dateForm.setCommandListener(this);

// создать список имен
BuildNameList();
// поиск именинника
String name = SearchBirthday();
// если имя найдено
if(name!=null) {
    // создать форму напоминания
    Form remindForm = new Form("Reminder");
    // добавить напоминание в форму
    remindForm.append(name + " has a birthday today!");
    // добавить команду возврата
    remindForm.addCommand(ok);
    remindForm.setCommandListener(this);
    // отобразить форму
    display.setCurrent(remindForm);
} else
    // отобразить список имен на экране
    display.setCurrent(nameList);
}

// метод SearchBirthday, возвращает строку с именем именинника
// если в этот день именинников нет, то возвращает null
private String SearchBirthday() {
    String name = null;
    try {
        // создать объект фильтра по дню рождения
        BirthdayFilter filter = new BirthdayFilter();
        // получить список записей с подходящим днем рождения
        RecordEnumeration re = recordStore.enumerateRecords(filter, null,
                                                             false);

        // получить ID записи
        int id = re.nextRecordId();
        // получить запись по ID
```

```
byte[] record = recordStore.getRecord(id);
// преобразовать запись в байтовый поток
ByteArrayInputStream bais = new ByteArrayInputStream(record);
// создать поток, поддерживающий чтение по типу
DataInputStream dis = new DataInputStream(bais);
// считать из потока строку с именем
name = dis.readUTF();
}
catch(RecordStoreException rse) {}
catch(IOException ioe) {}
// вернуть имя именинника
return name;
}

// метод BuildNameList: создает список имен из адресной книги
private void BuildNameList() {
    // создать объект списка
    nameList = new List("Address-Book", List.IMPLICIT);
    nameList.setCommandListener(this);
    // команда добавления записи
    nameList.addCommand(add);
    // команда получения параметров записи
    nameList.addCommand(ok);
    try {
        // получить количество записей
        int size = recordStore.getNumRecords();
        // создать массив для хранения ID записей
        recIndexes = new int[size];
        // создать объект компаратора в алфавитном порядке
        AlphabeticalOrdering comparator = new AlphabeticalOrdering();
        // получить список записей хранилища
        RecordEnumeration re = recordStore.enumerateRecords(null, comparator,
                                                             false);
        // индекс массива хранения ID записей
        int i=0;
        // бесконечный цикл: выход из цикла происходит по
        // формированию исключения получения ID следующей записи,
        // после того как было получено ID последней записи
        while(true) {
            // получить ID следующей записи
            int id = re.nextRecordId();
            // записать ID в массив
            recIndexes[i++]=id;
            // получить запись по ID
            byte[] record = recordStore.getRecord(id);
```

```
// преобразовать запись в байтовый поток
ByteArrayInputStream bais=new ByteArrayInputStream(record);
// создать поток, поддерживающий чтение по типу
DataInputStream dis = new DataInputStream(bais);
// считать из потока первую строку и добавить в список
nameList.append(dis.readUTF(),null);
}
}
catch(RecordStoreException rse) {}
catch(IOException ioe) {}
}

// блок прослушивания команд
public void commandAction(Command c, Displayable d) {
    // команда Add отображает экран ввода имени
    if(c==add)
        display.setCurrent(tbName);
    // команда Ok
    if(c==ok) {
        // если команда вызвана из экрана со списком имен
        if(d==nameList) {
            try {
                // получить в массиве recIndexes ID,необходимой записи
                int id = recIndexes[nameList.getSelectedIndex()];
                // считать необходимую запись
                byte[] record = recordStore.getRecord(id);
                // преобразовать запись в байтовый поток
                ByteArrayInputSteam bais=new ByteArrayInputSteam(record);
                // создать поток, поддерживающий чтение по типу
                DataInputStream dis = new DataInputStream(bais);
                // создать форму для отображения параметров
                Form infoForm = new Form("");
                // считать строки с параметрами и добавить в форму
                infoForm.append(dis.readUTF()+"\n");
                infoForm.append(dis.readUTF()+"\n");
                infoForm.append(dis.readUTF());
                // создать объект даты рождения
                Date birthday = new Date(dis.readLong());
                // создать объект поля ввода даты
                DateField df = new DateField("",DateField.DATE);
                // установить дату рождения в поле ввода даты
                df.setDate(birthday);
                // добавить даты рождения в форму отображения параметров
                infoForm.append(df);
                // добавить команду возврата к списку имен
```

```

infoForm.addCommand(ok);
infoForm.setCommandListener(this);
// отобразить форму на экране
display.setCurrent(infoForm);
}
catch (RecordStoreException rse) {}
catch (IOException ioe) {}
} else { // если команда вызвана из формы параметров записи
// вернуться к списку имен
display.setCurrent(nameList);
}
}

// команда Next: перейти к следующему полю ввода
if(c==next) {
// из поля ввода имени - к полю ввода номера телефона
if(d==tbName) display.setCurrent(tbPhone);
// из поля ввода номера телефона - к полю ввода e-mail
if(d==tbPhone) display.setCurrent(tbEMail);
// из поля ввода e-mail - к дню рождения
if(d==tbEMail) display.setCurrent(dateForm);
// из поля ввода дня рождения
if(d==dateForm) {
// создать байтовый поток вывода
ByteArrayOutputStream baos = new ByteArrayOutputStream();
// создать поток вывода, поддерживающий запись по типу
DataOutputStream dos = new DataOutputStream(baos);
try {
// записать введенные параметры в поток вывода
dos.writeUTF(tbName.getString());
dos.writeUTF(tbPhone.getString());
dos.writeUTF(tbEMail.getString());
dos.writeLong(dateField.getDate().getTime());
// добавить запись в хранилище
recordStore.addRecord(baos.toByteArray(), 0, baos.size());
}
catch (IOException ioe) {}
catch (RecordStoreException rse) {}
// создать список имен
BuildNameList();
// отобразить список имен на экране
display.setCurrent(nameList);
}
}

// команда Back: возе

```



```

if(c==back) {
    // из поля ввода имени - к списку имен
    if(d==tbName) display.setCurrent(nameList);
    // из поля ввода номера телефона - к полю ввода имени
    if(d==tbPhone) display.setCurrent(tbName);
    // из поля ввода e-mail - к полю ввода номера телефона
    if(d==tbEMail) display.setCurrent(tbPhone);
    // из поля ввода даты рождения - к полю ввода e-mail
    if(d==dateForm) display.setCurrent(tbEMail);
}
}

// класс фильтра записей по дням рождения
private class BirthdayFilter implements RecordFilter {

    // метод сравнения записей
    public boolean matches(byte[] candidate) {
        // преобразовать запись в байтовый поток
        ByteArrayInputStream bais = new ByteArrayInputStream(candidate);
        // создать поток поддерживающие чтение по типу
        DataInputStream dis = new DataInputStream(bais);
        // день рождения
        Date birthDate = new Date();
        try {
            // считать строковые параметры записи
            dis.readUTF();
            dis.readUTF();
            dis.readUTF();
            // считать дату рождения
            birthDate.setTime(dis.readLong());
        }
        catch (IOException ioe) { return false; }

        // получить два календаря с текущей датой
        Calendar rightNow = Calendar.getInstance();
        Calendar birthday = Calendar.getInstance();
        // установить дату рождения именинника
        birthday.setTime(birthDate);
        // сравнить день и месяц рождения с текущей датой
        if(rightNow.get(Calendar.DAY_OF_MONTH)==
            birthday.get(Calendar.DAY_OF_MONTH) &&
            rightNow.get(Calendar.MONTH)==
            birthday.get(Calendar.MONTH))
            return true;
        else

```

```
        return false;
    }
}

// класс компаратора записей по алфавиту
private class AlphabeticalOrdering implements RecordComparator {
    // метод сравнения записей
    public int compare(byte[] rec1, byte[] rec2) {
        // преобразовать записи в байтовый поток
        ByteArrayInputStream bais1 = new ByteArrayInputStream(rec1);
        ByteArrayInputStream bais2 = new ByteArrayInputStream(rec2);
        // создать потоки, поддерживающие чтение по типу
        DataInputStream dis1 = new DataInputStream(bais1);
        DataInputStream dis2 = new DataInputStream(bais2);
        // строки имен
        String name1 = null;
        String name2 = null;
        try {
            // считать строки с именами
            name1 = dis1.readUTF ();
            name2 = dis2.readUTF ();
        }
        catch (IOException ioe) {}
        // лексикографическое сравнение строк
        int result = name1.compareTo(name2);
        if (result < 0)
            // первая запись предшествует второй
            return RecordComparator.PRECEDES;
        else
            if (result == 0)
                // записи содержат идентичные имена
                return RecordComparator.EQUIVALENT;
            else
                // вторая запись предшествует первой
                return RecordComparator.FOLLOWS;
    }
}

} // class AddressBook
```

Литература

- Брюс Эккель. Философия Java. 3-е изд. СПб.: Питер, 2003.
- Жерздев С. В. Java 2 Micro Edition. ИТЛаб, ННГУ, ВМК, 2003.
- Вартан Пирумян. Платформа программирования J2ME для портативных устройств. КУДИЦ-Образ, 2002.
- Кен Арнольд, Джеймс Гослинг. Язык программирования JAVA. СПб.: Питер, 1997.

Использованы материалы порталов и форумов:

- java.sun.com/j2me/docs
- www.juga.ru
- www.javagu.ru

Евгений Леонидович Буткевич
Пищем программы и игры для сотовых телефонов

Главный редактор
Заведующий редакцией
Руководитель проекта
Литературный редактор
Иллюстрации
Художник
Корректоры
Верстка

Е. Строганова
А. Кривоцов
А. Крузенштерн
Е. Бурнашова
В. Демидова, С. Романов
С. Маликова
И. Смирнова, Н. Шелковникова
С. Романов

Лицензия ИД № 05784 от 07.09.01.

Подписано в печать 18.11.05. Формат 70×100/16. Усл. п. л. 16,77. Тираж 3000 экз. Заказ № 6647.

ООО «Питер Принт». 194044, Санкт-Петербург, Б. Сампсониевский пр., 29а.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная.

Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького

Федерального агентства по печати и массовым коммуникациям.

197110, Санкт-Петербург, Чкаловский пр., 15.

КЛУБ ПРОФЕССИОНАЛ

В 1997 году по инициативе генерального директора **Издательского дома «Питер»** Валерия Степанова и при поддержке деловых кругов города в Санкт-Петербурге был основан **«Книжный клуб Профессионал»**. Он собрал под флагом клуба профессионалов своего дела, которых объединяет постоянная тяга к знаниям и любовь к книгам. Членами клуба являются лучшие студенты и известные практики из разных сфер деятельности, которые хотят стать или уже стали профессионалами в той или иной области.

Как и все развивающиеся проекты, с течением времени книжный клуб вырос в **«Клуб Профессионал»**. Идею клуба сегодня формируют три основные «клубные» функции:

- неформальное общение и совместный досуг интересных людей;
- участие в подготовке специалистов высокого класса (семинары, пакеты книг по специальной литературе);
- формирование и высказывание мнений современного профессионала (при встречах и на страницах журнала).

КАК ВСТУПИТЬ В КЛУБ?

Для вступления в **«Клуб Профессионал»** вам необходимо:

- ознакомиться с правилами вступления в **«Клуб Профессионал»** на страницах журнала или на сайте **www.piter.com**;
- выразить свое желание вступить в **«Клуб Профессионал»** по электронной почте **postbook@piter.com** или по тел. (812) 103-73-74;
- заказать книги на сумму не менее 500 рублей в течение любого времени или приобрести комплект **«Библиотека профессионала»**.

«БИБЛИОТЕКА ПРОФЕССИОНАЛА»

Мы предлагаем вам получить все необходимые знания, подписавшись на **«Библиотеку профессионала»**. Она для тех, кто экономит не только время, но и деньги. Покупая комплект — книжную полку **«Библиотека профессионала»**, вы получаете:

- скидку 15% от розничной цены издания, без учета почтовых расходов;
- при покупке двух или более комплектов — дополнительную скидку 3%;
- членство в **«Клубе Профессионал»**;
- подарок — журнал **«Клуб Профессионал»**.

Закажите бесплатный журнал
«Клуб Профессионал».

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM



КНИГА-ПОЧТОЙ



**ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:**

- по телефону: (812) 103-73-74;
- по электронному адресу: postbook@piter.com;
- на нашем сервере: www.piter.com;
- по почте: 197198, Санкт-Петербург, а/я 619,
ЗАО «Питер Пост».

**ВЫ МОЖЕТЕ ВЫБРАТЬ ОДИН ИЗ ДВУХ СПОСОБОВ ДОСТАВКИ
И ОПЛАТЫ ИЗДАНИЙ:**

-  Наложенным платежом с оплатой заказа при получении посылки на ближайшем почтовом отделении. Цены на издания приведены ориентировочно и включают в себя стоимость пересылки по почте (**но без учета авиатарифа**). Книги будут высланы нашей службой «Книга-почтой» в течение двух недель после получения заказа или выхода книги из печати.
-  Оплата наличными при курьерской доставке (**для жителей Москвы и Санкт-Петербурга**). Курьер доставит заказ по указанному адресу в удобное для вас время в течение трех дней.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, код, количество заказываемых экземпляров.

**Вы можете заказать бесплатный
журнал «Клуб Профессионал»**

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»

предлагают эксклюзивный ассортимент компьютерной, медицинской,
психологической, экономической и популярной литературы

РОССИЯ

Москва м. «Павелецкая», 1-й Кожевнический переулок, д. 10; тел./факс (095) 234-38-15,
255-70-67, 255-70-68; e-mail: sales@piter.msk.ru

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а;
тел./факс (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Воронеж ул. 25 января, д. 4; тел./факс (0732) 39-43-62, 39-61-70;
e-mail: pitervm@comch.ru

Екатеринбург ул. 8 Марта, д. 2676, офис 203, 205; тел./факс (343) 225-39-94, 225-40-20;
e-mail: piter-ural@isnet.ru

Нижний Новгород ул. Совхозная, д. 13; тел. (8312) 41-27-31;
e-mail: piter@infonet.nnov.ru

Новосибирск ул. Немировича-Данченко, д. 104, офис 502;
тел./факс (383) 354-13-09, 211-27-18; e-mail: piter-sib@risp.ru

Ростов-на-Дону ул. Ульяновская, д. 26; тел. (863) 269-91-22, 269-91-30;
e-mail: jupiter@rost.ru

Самара ул. Молодогвардейская, д. 33, литер А2, ком. 225; тел. (846) 77-89-79;
e-mail: pitvolga@samtel.ru

УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10-11; тел./факс (10-38-057) 712-27-05,
751-10-02, (0572) 58-41-45; e-mail: piter@kharkov.piter.com

Киев пр. Московский, д. 6, кор. 1, офис 33; тел./факс (10-38-044) 490-35-68, 490-35-69;
e-mail: office@piter-press.kiev.ua

БЕЛАРУСЬ

Минск ул. Бобруйская, д. 21, офис 3; тел./факс (10-375-17) 226-19-53;
e-mail: office@minsk.piter.com



Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 703-73-73.**

E-mail: grigorjan@piter.com



Издательский дом «Питер» приглашает к сотрудничеству авторов.
Обращайтесь по телефонам: **Санкт-Петербург — (812) 703-73-72,**
Москва — (095) 974-34-50.



Заказ книг для вузов и библиотек: **(812) 703-73-73.**

Специальное предложение — e-mail: kozin@piter.com

Башкортостан

Уфа, «Азия», ул. Гоголя, д. 36, офис 5,
тел./факс (3472) 50-39-00, 51-85-44.
E-mail: asiаufa@ufanet.ru

Дальний Восток

Владивосток, «Приморский торговый дом книги»,
тел./факс (4232) 23-82-12.
E-mail: bookbase@mail.primorye.ru

Хабаровск, «Мирс»,
тел. (4212) 30-54-47, факс 22-73-30.
E-mail: sale_book@bookmirs.khv.ru

Хабаровск, «Книжный мир»,
тел. (4212) 32-85-51, факс 32-82-50.
E-mail: postmaster@worldbooks.kht.ru

Европейские регионы России

Архангельск, «Дом книги»,
тел. (8182) 65-41-34, факс 65-41-34.
E-mail: book@atnet.ru

Калининград, «Вестер»,
тел./факс (0112) 21-56-28, 21-62-07.
E-mail: nshibkova@vester.ru
<http://www.vester.ru>

Северный Кавказ

Ессентуки, «Россы», ул. Октябрьская, 424,
тел./факс (87934) 6-93-09.
E-mail: rossy@kmw.ru

Сибирь

Иркутск, «ПродаЛитъ»,
тел. (3952) 59-13-70, факс 51-30-70.
E-mail: prodalit@irk.ru
<http://www.prodalit.irk.ru>

Иркутск, «Антей-книга»,
тел./факс (3952) 33-42-47.
E-mail: antey@irk.ru

Красноярск, «Книжный мир»,
тел./факс (3912) 27-39-71.
E-mail: book-world@public.krasnet.ru

Нижневартовск, «Дом книги»,
тел. (3466) 23-27-14, факс 23-59-50.
E-mail: book@nvtovsk.wsnet.ru

Новосибирск, «Топ-книга»,
тел. (3832) 36-10-26, факс 36-10-27.
E-mail: office@top-kniga.ru
<http://www.top-kniga.ru>

Тюмень, «Друг»,
тел./факс (3452) 21-34-82.
E-mail: drug@tyumen.ru

Тюмень, «Фолиант»,
тел. (3452) 27-36-06, факс 27-36-11.
E-mail: foliant@tyumen.ru

Челябинск, ТД «Эврика», ул. Барбюса, д. 61,
тел./факс (3512) 52-49-23.
E-mail: evrika@chel.surnet.ru

Татарстан

Казань, «Таис»,
тел. (8432) 72-34-55, факс 72-27-82.
E-mail: tais@bancorp.ru

Урал

Екатеринбург, магазин № 14,
ул. Челюскинцев, д. 23,
тел./факс (3432) 53-24-90.
E-mail: gvardia@mail.ur.ru

Екатеринбург, «Валео-книга»,
ул. Ключевская, д. 5,
тел./факс (3432) 42-56-00.
E-mail: valeo@etel.ru