

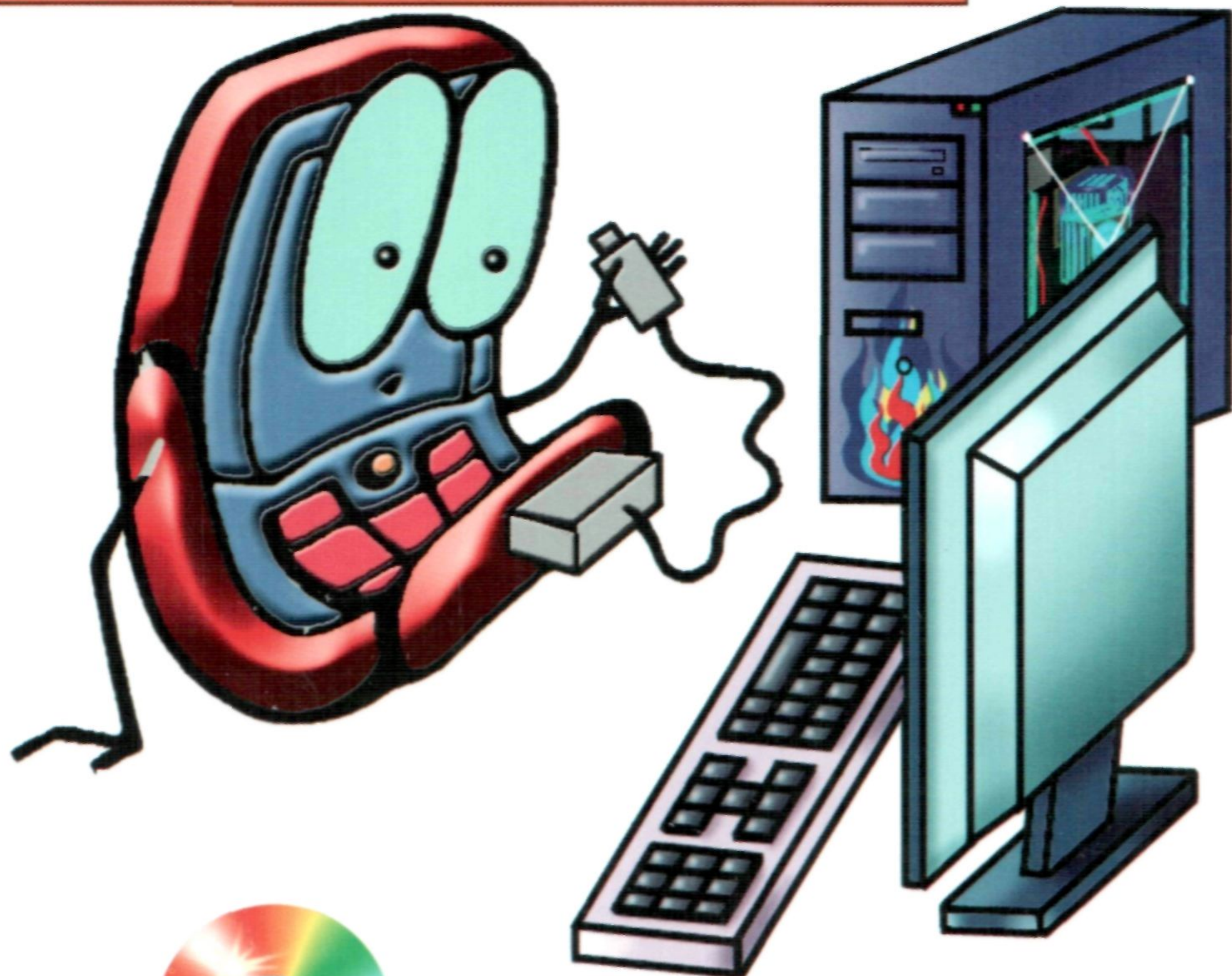


Мобильные технологии

ПРОГРАММИРОВАНИЕ МОБИЛЬНЫХ ТЕЛЕФОНОВ

Бестселлер!

на JAVA 2 ME



Компакт-диск к книге

Горнаков С. Г.

Второе дополненное
и переработанное издание

УДК 004.438
ББК 32.973.26-018.2

Горнаков С. Г.

Г26 Программирование мобильных телефонов на Java 2 Micro Edition. - М.: ДМК Пресс, 2008. - 512 с: ил.

ISBN 5-94074-409-5

Вы держите в руках второе и переработанное издание одной из популярных книг о программировании мобильных телефонов на Java 2 ME. Первое издание книги продавалось огромными тиражами по всему постсоветскому пространству. Автор книги создал уникальное издание, обучившее огромное количество начинающих программистов делать приложения для мобильных телефонов. Спустя три года после выхода первого издания, по многочисленным заявкам читателей была создана новая и переработанная версия книги.

Книга содержит девять новых глав. Часть старого материала первого издания была переработана в соответствии с веяниями времени. Теперь читатель кроме программирования приложений для платформы Java 2 ME, изучит полный процесс создания мобильной игры. В течение книги будет освещен подход в формировании полноценного мобильного игрового движка, освоена работа с графикой, показаны примеры многослойных и анимированных игровых карт. Будут рассмотрены основы искусственного интеллекта, игровые столкновения, создание интерактивного меню игры, подсчет очков и жизненной энергии главного героя, сохранение данных в памяти, работа со звуком и многое другое. Итогом книги станет создание полноценной мобильной игры и знакомство с разработкой пользовательских программ на Java 2 ME.

Компакт-диск содержит средства разработки мобильных приложений NetBeans и J2ME Wireless Toolkit, а также большой набор телефонных эмуляторов от компаний Nokia, BenQ-Siemens, Sony Ericsson, Motorola и Samsung.

УДК 004.438
ББК 32.973.26-018.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 5-94074-409-5

© Горнаков С. Г., 2008
© Оформление, ДМК Пресс, 2008



Содержание

Предисловие	13
Что вы должны знать	14
Какое программное обеспечение используется	14
О чем эта книга	14
Содержание компакт-диска	14
Благодарности	15
Об авторе	15
 Часть I. Введение в Java 2 Micro Edition	16
 Глава 1. Основы языка программирования Java	17
1.1. Введение в программирование	17
1.2. Объектно-ориентированное программирование	18
1.2.1. Классы	19
1.2.2. Методы	20
1.3. Синтаксис и семантика языка Java 2 ME	20
1.3.1. Комментарии	22
1.3.2. Типы данных Java	22
1.3.3. Операторы	24
1.3.4. Метод main	27
1.3.5. Закрытые и открытые члены классов	28
1.4. Конструктор	28
1.5. Объекты классов	29
1.6. Условные операторы	33
1.7. Управляющий оператор	34
1.8. Циклы	35
1.8.1. Оператор while	36
1.8.2. Цикл do/while	37
1.8.3. Цикл for	38
1.9. Массивы	39
1.10. Наследование	40
1.10.1. Конструктор суперкласса	43
1.11. Интерфейсы	45
1.12. Пакеты	46

Глава 2. Платформа Java 2 Micro Edition	48
2.1. Конфигурация CDC	50
2.2. Конфигурация CLDC	50
2.2.1. Свойства языка Java	52
2.2.2. Виртуальная машина	52
2.3. Профиль	52
2.4. Профиль MIDP 2.0 и конфигурация CLDC 1.1	54
2.4.1. Пакет Java.lang	55
2.4.2. Пакет Java.util	57
2.4.3. Пакет Java.io	58
2.4.4. Пакет javax.microedition.io	59
2.4.5. Пакет javax.microedition.lcdui	60
2.4.6. Пакет javax.microedition.lcdui.game	62
2.4.7. Пакет javax.microedition.media	62
2.4.8. Пакет javax.microedition.media.control	63
2.4.9. Пакет javax.microedition.midlet	64
2.4.10. Пакет javax.microedition.pki	64
2.4.11. Пакет javax.microedition.rms	64
Глава 3. Инструментальные средства разработки мобильных приложений	67
3.1. Установка Java2 SDK SE	67
3.2. Инструментарий J2ME Wireless Toolkit	71
3.2.1. Установка J2ME Wireless Toolkit	72
3.2.2. Знакомимся с J2ME Wireless Toolkit	74
3.2.3. Создание проекта в J2ME Wireless Toolkit	75
3.2.4. Компиляция и запуск программы в J2ME Wireless Toolkit	77
3.2.5. Упаковка программ	79
3.3. Инструментарий NetBeans IDE	81
3.3.1. Установка NetBeans IDE	82
3.3.2. Установка Mobility Pack	85
3.3.3. Создание проекта	85
3.3.4. Компиляция и упаковка проекта	90
3.3.5. Добавление в проект новых эмуляторов	91
Глава 4. Телефонные эмуляторы	94
4.1. Motorola	95
4.2. Nokia	97
4.2.1. Сайт компании Nokia	98
4.2.2. Carbide.j	99

4.3. BenQ-Siemens.....	101
4.4. Sony Ericsson.....	104
4.5. Samsung.....	104
4.6. Интеграция эмуляторов в NetBeans IDE.....	105
Часть II. Разработка программ.....	112
Глава 5. Механизм работы приложений.....	113
5.1. Мидлет.....	113
5.1.1. Модель работы мидлета.....	119
5.2. Пользовательский интерфейс.....	120
5.3. Переход с экрана на экран.....	122
5.4. Навигация.....	128
Глава 6. Классы пользовательского интерфейса.....	134
6.1. Класс Form.....	134
6.1.1. Методы класса Form.....	135
6.2. Класс Item.....	137
6.2.1. Класс ChoiceGroup.....	138
6.2.2. Класс DateField.....	145
6.2.3. Класс TextField.....	148
6.2.4. Класс StringItem.....	151
6.2.5. Класс Spacer.....	156
6.2.6. Класс ImageItem.....	158
6.2.7. Класс Gauge.....	162
6.3. Класс Alert.....	165
6.3.1. Методы класса Alert.....	166
6.4. Класс List.....	169
6.4.1. Методы класса List.....	170
6.5. Класс Ticker.....	175
6.5.1. Методы класса Ticker.....	176
6.6. Класс Image.....	178
6.6.1. Методы класса Image.....	179
6.7. Класс Font.....	181
Глава 7. Программирование графики.....	187
7.1. Класс Canvas.....	187
7.1.1. Методы класса Canvas.....	188
7.2. Класс Graphics.....	189
7.2.1. Методы класса Graphics.....	189
7.3. Рисование линий.....	191
7.4. Рисование прямоугольников.....	195

7.5. Рисование дуг.....	198
7.6. Вывод текста.....	201
7.7. Механизм создания игрового цикла.....	203
7.8. Перемещение квадрата.....	204
7.9. Циклическое передвижение объекта по экрану.....	208
7.10. Столкновение.....	211
7.11. Перемещение объекта с помощью клавиш.....	215

Часть III. Пишем свою первую игру.....220

Глава 8. Игровые классы.....221

8.1. Класс GameCanvas.....	222
8.2. Класс Layer.....	223
8.3. Класс TiledLayer.....	224
8.4. Класс LayerManager.....	225
8.5. Класс Sprite.....	226
8.6. Создание фонового изображения.....	228
8.7. Обработка событий с клавиш телефона.....	233
8.8. Анимация в игровом процессе.....	239
8.9. Столкновение объектов.....	244
8.10. Игра «Метеоритный дождь».....	252
8.10.1. Идея игры.....	253
8.10.2. Графика.....	253
8.10.3. Исходные коды.....	254

Глава 9. Формируем каркас игры.....255

9.1. Механизм работы мобильных игр.....	255
9.1.1. Вход в игру.....	256
9.1.2. Инициализация игры.....	256
9.1.3. Игровой цикл.....	256
9.1.4. Обработка ввода пользователя.....	257
9.1.5. Игровая логика.....	257
9.1.6. Синхронизация времени.....	258
9.1.7. Вывод графики на экран.....	258
9.1.8. Пауза в игре.....	258
9.1.9. Выход из игры.....	258
9.2. Как работают шаблоны.....	258
9.2.1. Запуск игры и информационная заставка.....	259
9.2.2. Вступительный ролик.....	259
9.2.3. Меню.....	259
9.2.4 Загрузка и запуск игры.....	260

9.3. Структура классов игры «Метеоритный дождь».....	260
9.4. Как устроен каркас игровых классов?.....	261
9.5. Класс GameMidlet.....	262
9.6. Класс Splash.....	267
9.7. Класс Loading.....	271
9.8. Класс MainGameCanvas.....	275
9.8.1. Глобальные переменные.....	278
9.8.2. Конструктор класса MainGameCanvas.....	279
9.8.3. Метод createGame().....	280
9.8.4. Установка объектов и обновление состояния игры.....	281
9.8.5. Выводим графику на экран телефона.....	282
9.8.6. Обработка клавиш выбора.....	283
9.8.7. Игровой цикл.....	285
Глава 10. Добавляем в игру меню.....	293
10.1. Идея реализации игрового меню.....	293
10.2. Класс Menu.....	294
10.3. Планируем запуск меню.....	303
Глава 11. Создание игровых карт.....	307
11.1. Игровая карта.....	307
11.1.1. Техника компоновки карт.....	307
11.2. Многослойные карты.....	309
11.3. Инструменты создания игровых карт.....	309
11.3.1. Создаем карту.....	310
11.4. Класс Background.....	313
11.5. Загружаем в игру карту.....	316
11.6. Работа с памятью телефона.....	323
11.6.1. Запись данных в память.....	324
11.6.2. Чтение данных.....	325
Глава 12. Создание и перемещение корабля по экрану.....	327
12.1. Класс Sprite.....	327
12.1.1. Конструкторы класса Sprite.....	328
12.1.2. Методы класса Sprite.....	329
12.1.3. Константы класса Sprite.....	329
12.2. Класс Ship.....	329
12.2.1. Создаем корабль.....	330
12.2.2. Исходный код класса Ship.java.....	331
12.2.3. Создание объекта класса Ship.....	338

12.3. Проект Demo.....	345
12.3.1. Класс Background.Java.....	346
12.3.2. Класс Ship.java.....	348
12.3.3. Класс MainGameCanvas.java.....	350
12.3.4. Класс GameMidlet.java.....	354
Глава 13. Основы искусственного интеллекта.....	357
13.1. Структура классов в демонстрационных примерах.....	358
13.1.1. Класс GameMidlet.....	358
13.1.2. Класс Splash.....	360
13.1.3. Класс Loading.....	361
13.1.4. Класс Background.....	362
13.1.5. Класс Ship.....	365
13.1.6. Класс Boll.....	365
13.1.7. Класс MainGameCanvas.....	366
13.2. Движение в заданном направлении.....	370
13.3. Движение объекта за целью.....	371
13.4. Движение объекта от цели.....	372
13.5. Движение в случайном направлении.....	374
13.6. Шаблоны.....	377
13.7. Шаблоны с обработкой событий.....	380
13.8. Модель простой системы смены состояний.....	382
13.9. Распределенная логика смены состояний объекта.....	385
Глава 14. Движение спрайтов в пространстве.....	389
14.1. Метеориты.....	389
14.1.1. Класс Meteorite.....	390
14.1.2. Движение метеоритов.....	391
14.2. Реализуем стрельбу корабля.....	394
14.2.1. Класс Shot.....	395
14.2.2. Класс Ship.....	395
14.2.3. Движение пуль.....	397
Глава 15. Игровые столкновения.....	406
15.1. Пишем код для обработки игровых столкновений.....	406
15.1.1. Столкновения корабля с метеоритом.....	407
15.1.2. Столкновение пуль и метеоритов.....	408
15.2. Рисуем на экране взрывы.....	409
15.3. Добавляем в игру подсчет набранных очков.....	412
15.4. Механизм подсчета жизненной энергии корабля.....	413
15.5. Графическое представление жизненной энергии корабля на экране телефона.....	415

Глава 16. Звуковые эффекты	427
16.1. Пакет javax.microedition.media	428
16.1.1. Интерфейс Control	428
16.1.2. Интерфейс Controllable	428
16.1.3. Интерфейс Player	428
16.1.4. Интерфейс PlayerListener	429
16.1.5. Класс Manager	430
16.2. Пакет javax.microedition.media.control	430
16.2.1. Интерфейс ToneControl	430
16.2.2. Интерфейс VolumeControl	431
16.3. Воспроизведение звуковых файлов	431
16.4. Воспроизведение тональных звуков	432
16.5. Добавляем звук в игру	438
Приложение 1. Обзор компакт-диска	444
Приложение 2. Справочник по Java 2 Micro Edition	445
2.1. Пакет Java.lang	445
2.1.1. Интерфейс Runnable	445
2.1.2. Класс Boolean	445
2.1.3. Класс Byte	446
2.1.4. Класс Character	446
2.1.5. Класс Class	447
2.1.6. Класс Integer	447
2.1.7. Класс Long	448
2.1.8. Класс Math	449
2.1.9. Класс Object	449
2.1.10. Класс Runtime	450
2.1.11. Класс Short	450
2.1.12. Класс String	451
2.1.13. Класс StringBuffer	453
2.1.14. Класс System	454
2.1.15. Класс Thread	457
2.1.16. Класс Throwable	455
2.1.17. Исключения	457
2.1.18. Ошибки	456
2.2. Пакет Java.util	457
2.2.1. Интерфейс Enumeration	457
2.2.2. Класс Calendar	457
2.2.3. Класс Date	458
2.2.4. Класс Hashtable	458

2.2.5. Класс Random.....	459
2.2.6. Класс Stack.....	460
2.2.7. Класс Timer.....	460
2.2.8. Класс TimerTask.....	461
2.2.9. Класс TimeZone.....	461
2.2.10. Класс Vector.....	461
2.2.11. Исключения.....	463
2.3. Пакет Java.io.....	463
2.3.1. Интерфейс DataInput.....	463
2.3.2. Интерфейс DataOutput.....	463
2.3.3. Класс ByteArrayInputStream.....	464
2.3.4. Класс ByteArrayOutputStream.....	465
2.3.5. Класс DataInputStream.....	465
2.3.6. Класс DataOutputStream.....	466
2.3.7. Класс InputStream.....	467
2.3.8. Класс InputStreamReader.....	468
2.3.9. Класс OutputStream.....	468
2.3.10. Класс OutputStreamWriter.....	468
2.3.11. Класс PrintStream.....	469
2.3.12. Класс Reader.....	470
2.3.13. Класс Writer.....	470
2.3.14. Исключения.....	471
2.4. Пакет javax.microedition.io.....	471
2.4.1. Интерфейс CommConnection.....	471
2.4.2. Интерфейс Connection.....	471
2.4.3. Интерфейс ContentConnection.....	471
2.4.4. Интерфейс Datagram.....	472
2.4.5. Интерфейс DatagramConnection.....	472
2.4.6. Интерфейс HttpConnection.....	472
2.4.7. Интерфейс HttpsConnection.....	474
2.4.8. Интерфейс InputConnection.....	474
2.4.9. Интерфейс OutputConnection.....	474
2.4.10. Интерфейс SecureConnection.....	474
2.4.11. Интерфейс SecurityInfo.....	474
2.4.12. Интерфейс ServerSocketConnection.....	475
2.4.13. Интерфейс SocketConnection.....	475
2.4.14. Интерфейс StreamConnection.....	475
2.4.15. Интерфейс StreamConnectionNotifier.....	475
2.4.16. Интерфейс UDPDatagramConnection.....	475
2.4.17. Класс Connector.....	476

2.4.18.	Класс PushRegistry.....	476
2.4.19.	Исключение.....	476
2.5.	Пакет javax.microedition.lcdui.....	477
2.5.1.	Интерфейс Choice.....	477
2.5.2.	Интерфейс CommandListener.....	478
2.5.3.	Интерфейс ItemCommandListener.....	478
2.5.4.	Интерфейс ItemStateListener.....	478
2.5.5.	Класс Alert.....	478
2.5.6.	Класс AlertType.....	479
2.5.7.	Класс Canvas.....	479
2.5.8.	Класс ChoiceGroup.....	481
2.5.9.	Класс Command.....	482
2.5.10.	Класс Custom Item.....	482
2.5.11.	Класс DateField.....	484
2.5.12.	Класс Display.....	484
2.5.13.	Класс Displayable.....	485
2.5.14.	Класс Font.....	486
2.5.15.	Класс Form.....	487
2.5.16.	Класс Gauge.....	487
2.5.17.	Класс Graphics.....	488
2.5.18.	Класс Image.....	490
2.5.19.	Класс ImageItem.....	491
2.5.20.	Класс Item.....	491
2.5.21.	Класс List.....	493
2.5.22.	Класс Screen.....	494
2.5.23.	Класс Spacer.....	494
2.5.24.	Класс StringItem.....	494
2.5.25.	Класс TextBox.....	495
2.5.26.	Класс TextField.....	495
2.5.27.	Класс Ticker.....	497
2.6.	Пакет javax.microedition.lcdui.game.....	497
2.6.1.	Класс GameCanvas.....	497
2.6.2.	Класс Layer.....	498
2.6.3.	Класс LayerManager.....	498
2.6.4.	Класс Sprite.....	498
2.6.5.	Класс TiledLayer.....	499
2.7.	Пакет javax.microedition.media.....	500
2.7.1.	Интерфейс Control.....	500
2.7.2.	Интерфейс Controllable.....	500
2.7.3.	Интерфейс Player.....	501

2.7.4. Интерфейс PlayerListener.....	501
2.7.5. Класс Manager.....	502
2.7.6. Исключения.....	502
2.8. Пакет javax.microedition.media.control.....	502
2.8.1. Интерфейс ToneControl.....	502
2.8.2. Интерфейс VolumeControl.....	503
2.9. Пакет javax.microedition.midlet.....	503
2.9.1. Класс MIDlet.....	503
2.9.2. Исключение.....	504
2.10. Пакет javax.microedition.pki.....	504
2.10.1. Интерфейс Certificate.....	504
2.10.2. Исключение.....	504
2.11. Пакет javax.microedition.rms.....	504
2.11.1. Интерфейс RecordComparator.....	504
2.11.2. Интерфейс RecordEnumeration.....	505
2.11.3. Интерфейс RecordFilter.....	505
2.11.4. Интерфейс RecordListener.....	505
2.11.5. Класс RecordStore.....	506
Алфавитный указатель.....	508

<http://palata-x.narod.ru>

Предисловие

Кто бы мог подумать пять-шесть лет назад, что с помощью мобильного телефона можно будет делать фотографии, играть в игры, смотреть видео, слушать музыку, получать электронную почту, просматривать интернет-страницы, читать книги, создавать и редактировать документы Word и Excel...

Сейчас мобильный телефон - это нечто большее, чем просто телефон. Может быть, мы этого еще и не осознали, но телефон постепенно превращается в мобильный персональный компьютер, небольшого размера, но с огромными возможностями. Телефон становится все больше и больше коммуникационным центром, в котором сосредоточено множество функций персонального компьютера, и при этом телефон легко помещается в карман!

Что послужило такому развитию мобильной индустрии? По всей видимости, в этом есть большие заслуги платформы Java 2 ME, поскольку загрузка дополнительных программ на телефон автоматически расширяет его возможности. Производители телефонов не в силах предусмотреть, создать или установить все необходимые пользователю программы. Поэтому мобильные устройства с возможностью установки дополнительных программ всегда будут пользоваться наибольшим спросом у покупателей.

По некоторым данным, среднестатистический покупатель меняет свой телефон раз в полгода или год и стремится приобрести телефон выше классом. Телефоны с поддержкой платформы Java 2 ME дают возможность пользователю устанавливать мобильные Java-программы и игры, улучшая тем самым потенциал самого телефона. Сейчас на рынке более 98% телефонов поддерживают платформу Java 2 ME MIDP 2.0, и даже корпорация Microsoft в операционной системе Windows Mobile 5.0 реализовала полноценную поддержку Java 2 ME (CLDC 1.1/MIDP 2.0), чего не было ранее.

Стремительное развитие технологий мобильных устройств обязано подразумевать определенный эталон в данном направлении. Немалое количество сторонних разработчиков при строгой стандартизации заинтересованы в развитии рынка программного обеспечения телефонов. Поэтому рынок мобильных устройств по своим масштабам считается наиболее перспективным, а платформа Java 2 ME является стандартом в создании программ для телефонов.

Язык программирования Java 2 ME сам по себе не сложен, а обилие готовых библиотек позволяет писать Java-приложения за короткий промежуток времени. И что самое главное, наборы инструментальных средств программирования, предоставляемые компанией Sun Microsystems и производителями телефонов, абсолютно бесплатны! Множество энтузиастов со всего мира в кратчайшие сроки создают как платные, так и бесплатные программы и игры на Java 2 ME. Задача этой книги -

дать исчерпывающую информацию по созданию программ и игр для мобильных телефонов с поддержкой платформы Java 2 ME.

Что вы должны знать

Если вы новичок в программировании и не знакомы ни с одним языком программирования, то в первой главе содержится полная информация непосредственно по языку программирования Java. Изучив эту главу, вы сможете приступить к прочтению книги. Те читатели, которые уже знакомы с основами Java, могут сразу приступить к прочтению второй главы.

Все главы книги написаны в доступной форме и рассчитаны на широкий круг читателей. Множество примеров с подробными комментариями призваны улучшить понимание общей концепции платформы Java 2 ME.

Какое программное обеспечение используется

Вся работа по созданию программ для мобильных телефонов происходит на компьютере, и наличие мобильного телефона совсем не требуется. В качестве операционной системы используется Windows. Большинство программного обеспечения, находящегося на компакт-диске, ориентировано именно на эту операционную систему.

Компакт-диск, идущий в комплекте с этой книгой, имеет просто потрясающий набор инструментальных средств для создания приложений на Java 2 ME. На компакт-диске вы найдете две бесплатные интегрированные среды программирования от компании Sun Microsystems, а также множество бесплатных инструментальных средств от таких известных компаний производителей телефонов, как BenQ-Siemens, Nokia, Sony Ericsson, Motorola и Samsung.

О чем эта книга

Цель этой книги - научить вас создавать программы и игры на Java 2 ME MIDP 2.0. Книга предназначена как для самостоятельного изучения языка Java, так и для создания программ и игр для мобильных телефонов. Подробно рассматриваются все имеющиеся пакеты библиотек Java 2 ME, разбираются классы, методы и интерфейсы, раскрывается механизм работы программ на Java 2 ME. Прочтение всей книги лучше осуществлять в хронологическом порядке, главу за главой. В приложении 2 содержится справочник, дающий исчерпывающую информацию по всем интерфейсам, классам, методам и константам платформы Java 2 Micro Edition.

Содержание компакт-диска

Компакт-диск содержит исходные коды программ и игр из книги. Кроме этого, представлены две бесплатные среды программирования приложений для мобильных

телефонов компании Sun Microsystems, а также большое количество телефонных эмуляторов от компаний Nokia, BenQ-Siemens, Sony Ericsson, Motorola, Samsung.

Благодарности

Прежде всего хочется поблагодарить свою жену Светлану за помощь при создании рисунков к этой книге, игре и обложке. Спасибо, дорогая, без тебя было бы сложно все это нарисовать! За сборку компакт-диска необходимо поблагодарить генерального директора издательства «ДМК-Пресс» Дмитрия Алексеевича Мовчана.

Об авторе

Горнаков Станислав Геннадьевич - профессиональный программист в области создания мобильных и компьютерных игр. Автор книг: «Symbian OS. Программирование мобильных телефонов на C++», «Программирование мобильных телефонов на Java 2 ME», «Самоучитель работы на КПК, коммуникаторе и смартфоне под управлением Windows Mobile», «Инструментальные средства программирования и отладки шейдеров в DirectX и OpenGL», «DirectX 9. Уроки программирования на C++», «Разработка компьютерных игр под Windows в XNA Game Studio Express» и «Разработка компьютерных игр для приставки Xbox 360 в XNA Game Studio Express».

Посетите интернет-страницу автора книги по адресу: www.gornakov.ru

<http://palata-x.narod.ru>



Часть I

ВВЕДЕНИЕ В JAVA 2 MICRO EDITION

Глава 1. Основы языка программирования Java

Глава 2. Платформа Java 2 ME

**Глава 3. Инструментальные средства разработки
мобильных приложений**

Глава 4. Телефонные эмуляторы

[**http://palata-x.narod.ru**](http://palata-x.narod.ru)



Глава 1. Основы языка программирования Java

Эта обзорная глава не претендует на роль полного руководства по языку программирования Java (Ява), но данного материала вам будет вполне достаточно для дальнейшего изучения книги. Предлагаемая в этом разделе информация к рассмотрению содержит основы языка Java и ориентирована на неподготовленного читателя. Я уверен, что после изучения этой главы вы сможете писать и читать исходные коды, встречающиеся в книге. Но нужно иметь в виду, что обучение языку Java будет происходить в соответствии с контекстом книги, а именно всей той части языка, которая необходима для программирования мобильных устройств. Такие «продвинутые» темы, как апплеты, библиотеки AWT, Swing, Graphic, в этой главе мы рассматривать не будем. В Java 2 Micro Edition все перечисленные (и не только) компоненты просто не используются.

1.1. Введение в программирование

Программирование - это написание исходного кода программы на одном из языков программирования. Существует множество различных языков программирования, благодаря которым создаются всевозможные программы, решающие определенный круг задач. *Язык программирования* - это набор зарезервированных слов, с помощью которых пишется исходный код программы. Компьютерные системы не в силах (пока) понимать человеческий язык и уж тем более человеческую логику (особенно женскую), поэтому все программы пишутся на языках программирования, которые впоследствии переводятся на язык компьютера или в машинный код. Системы, переводящие исходный код программы в машинный код, очень сложные, и их, как правило, создают не один десяток месяцев и не один десяток программистов. Такие системы называются *интегрированными средами программирования приложений*, или *инструментальными средствами*.

Система программирования представляет собой огромную продуманную визуальную среду, где можно писать исходный код программы, переводить его в машинный код, тестировать, отлаживать и многое другое. Дополнительно существуют программы, позволяющие производить вышеперечисленные действия при помощи командной строки. Язык Java предоставляет такую возможность, но в книге данная проблематика не освещается.

Вы, наверное, не раз слышали термин «программа написана под Windows или под Linux, Unix». Дело в том, что среды программирования при переводе языка программирования в машинный код могут быть двух видов - это *компиляторы*

и *интерпретаторы*. Компиляция или интерпретация программы задает способ дальнейшего выполнения программы на устройстве. Программы, написанные на языке Java, всегда работают на основе интерпретации, тогда как программы, написанные на C/C++, - компиляции. В чем разница этих двух способов?

Компилятор после написания исходного кода в момент компиляции читает сразу весь исходный код программы и переводит в машинный код. После чего программа существует как одно целое и может выполняться только в той операционной системе, в которой она была написана. Поэтому программы, написанные под Windows, не могут функционировать в среде Linux, и наоборот. Интерпретатор осуществляет пошаговое или построчное выполнение программы каждый раз, когда она выполняется. Во время интерпретации создается не выполняемый код, а виртуальный, который впоследствии выполняется виртуальной Java-машиной. Поэтому на любой платформе - Windows или Linux - Java-программы могут одинаково выполняться при наличии в системе виртуальной Java-машины, которая еще носит название *Системы времени выполнения*. Поскольку все телефоны сейчас оснащаются виртуальной Java-машиной, то и возможность запуска Java-программы на мобильном устройстве не вызовет проблем. Грубо говоря, производители мобильных устройств нашли способы (или разработали технологии), которые позволяют встраивать в телефоны виртуальную Java-машину, а значит, и запускать Java-программы на телефонах.

Итак, от вас сейчас требуется только изучение синтаксиса языка Java, для того чтобы писать и понимать исходные коды программ, написанные на этом языке. Этим мы и займемся в данной главе. В следующей главе вы узнаете больше о спецификациях языка Java 2 Micro Edition, или требованиях, на базе которых пишутся программы для телефонов. Затем мы освоим работу с одной из сред программирования и пакетами разработчика и далее перейдем к самому процессу программирования. А пока займемся языком Java и его объектно-ориентированной направленностью.

1.2. Объектно-ориентированное программирование

Объектно-ориентированное программирование строится на базе объектов, что в какой-то мере аналогично с нашим миром. Если оглянуться вокруг себя, то обязательно можно найти то, что поможет более ярко разобраться в модели такого программирования. Например, я сейчас сижу за столом и печатаю эту главу на компьютере, который состоит из монитора, системного блока, клавиатуры, мыши, колонок и т. д. Все эти части являются объектами, из которых состоит компьютер. Зная это, очень легко сформулировать какую-то обобщенную модель работы всего компьютера. Если не разбираться в тонкостях программных и аппаратных свойств компьютера, то можно сказать, что объект **Системный блок** производит определенные действия, которые показывает объект **Монитор**. В свою очередь, объект **Клавиатура** может корректировать или вовсе задавать действия для объекта

Системный блок, которые влияют на работу объекта Монитор. Представленный процесс очень хорошо характеризует всю систему объектно-ориентированного программирования.

Представьте себе некий мощный программный продукт, содержащий сотни тысяч строк кода. Вся программа выполняется построчно, строка за строкой, и в принципе каждая из последующих строк кода обязательно будет связана с предыдущей строкой кода. Если не использовать объектно-ориентированное программирование, и когда потребуется изменить этот программный код, скажем при необходимости улучшения каких-то элементов, то придется произвести большое количество работы со всем исходным кодом этой программы.

В объектно-ориентированном программировании все куда проще, вернемся к примеру компьютерной системы. Допустим, вас уже не устраивает 17-дюймовый монитор. Вы можете спокойно его обменять, например на 19-дюймовый или 20-дюймовый монитор, конечно же при наличии определенных материальных средств. Сам же процесс обмена не повлечет за собой огромных проблем, разве что драйвер придется сменить да вытереть пыль из-под старого монитора - и все. Примерно на таком принципе работы и строится объектно-ориентированное программирование, где определенная часть кода может представлять класс однородных объектов, которые можно легко модернизировать или заменять.

Объектно-ориентированное программирование очень легко и ясно отражает суть решаемой проблемы и, что самое главное, дает возможность без ущерба для всей программы убирать ненужные объекты, заменяя эти объекты на более новые. Соответственно, общая читабельность исходного кода всей программы становится намного проще. Существенно и то, что один и тот же код можно использовать в абсолютно разных программах.

1.2, 1. Классы

Стержнем всех программ Java являются *классы*, на которых основывается объектно-ориентированное программирование. Вы по сути уже знаете, что такое классы, но пока об этом не догадываетесь. В предыдущем разделе мы говорили об объектах, ставя в пример устройство всего компьютера. Каждый объект, из которых собран компьютер, является представителем своего класса. Например, класс Мониторов объединяет все мониторы вне зависимости от их типов, размеров и возможностей, а один какой-то конкретный монитор, стоящий на вашем столе, и есть объект класса мониторов.

Такой подход позволяет очень легко моделировать всевозможные процессы в программировании, облегчая решение поставленных задач. Например, имеются четыре объекта четырех разных классов: монитор, системный блок, клавиатура и колонки. Чтобы воспроизвести звуковой файл, необходимо при помощи клавиатуры дать команду системному блоку, само же действие по даче команды вы будете наблюдать визуально на мониторе, и в итоге колонки воспроизведут звуковой файл. То есть любой объект является частью определенного класса и содержит в себе все имеющиеся у этого класса средства и возможности. Объектов одного класса может быть столько, сколько это необходимо для решения поставленной задачи.

1.2.2. Методы

Когда приводился пример воспроизведения звукового файла, то было упомянуто о даче команды или сообщения, на основе которого и выполнялись определенные действия. Задача по выполнению действий решается с помощью методов, которые имеет каждый объект. *Методы* - это набор команд, с помощью которых можно производить те или иные действия с объектом.

Каждый объект имеет свое назначение и призван решать определенный круг задач с помощью методов. Какой толк был бы, например, в объекте Клавиатура, если нельзя было бы нажимать на клавиши, получая при этом возможность отдавать команды? Объект Клавиатура имеет некое количество клавиш, с помощью которых пользователь приобретает контроль над устройством ввода и может отдавать необходимые команды. Обработка таких команд в целом происходит с помощью методов.

Например, вы нажимаете клавишу Esc для отмены каких-либо действий и тем самым даете команду методу, закрепленному за этой клавишей, который на программном уровне решает эту задачу. Сразу же возникает вопрос о количестве методов объекта Клавиатура, но здесь может быть различная реализация - как от определения методов для каждой из клавиш (что, вообще-то, неразумно), так и до создания одного метода, который будет следить за общим состоянием клавиатуры. То есть этот метод следит за тем, была ли нажата клавиша, а потом в зависимости от того, какая из клавиш задействована, решает, что ему делать.

Итак, мы видим, что каждый из объектов может иметь в своем распоряжении набор методов для решения различных задач. А поскольку каждый объект является объектом определенного класса, то получается, что класс содержит набор методов, которыми и пользуются различные объекты одного класса. В языке Java все созданные вами методы должны принадлежать или являться частью какого-то конкретного класса.

1.3. Синтаксис и семантика языка Java 2 ME

Для того чтобы говорить и читать на любом иностранном языке, необходимо изучить алфавит и грамматику этого языка. Подобное условие наблюдается и при изучении языков программирования, с той лишь разницей, как мне кажется, что этот процесс несколько легче. Но прежде чем начинать писать исходный код программы, необходимо сначала решить поставленную перед вами задачу в любом удобном для себя виде.

Давайте создадим некий класс, отвечающий, например, за телефон, который будет иметь всего два метода: включающий и выключающий этот самый телефон. Поскольку мы сейчас не знаем синтаксиса языка Java, то напомним класс Телефон на абстрактном языке.

```
Класс Телефон
{
    Метод Включить ( )
    {
```

```
// операции по включению телефона
}
Метод Выключить ( )
{
// операции по выключению телефона
}
}
```

Примерно так может выглядеть класс Телефон. Заметьте, что *фигурные скобки* обозначают соответственно начало и конец тела класса, метода либо всякой последовательности данных. То есть скобки указывают на принадлежность к методу или классу. На каждую открывающуюся скобку обязательно должна быть закрывающаяся скобка. Чтобы не запутаться, их обычно ставят на одном уровне в коде, как в приведенном выше примере.

А теперь давайте запишем тот же самый класс, только уже на языке Java.

```
class Telefon
{
    void on ( )
    {
        // тело метода on ( )
    }
    void off ( )
    {
        // тело метода off ( )
    }
}
```

Ключевое слово `class` в языке Java объявляет класс, далее идет название самого класса. В нашем случае это `Telefon`. Сразу пару слов касательно регистра записи. Почти во всех языках программирования важно сохранять запись названий в том регистре, в котором она была сделана. Если вы написали `Telefon`, то уже такое написание, как `telefon` или `TELEfoN`, выдаст ошибку при компиляции. Как написали первоначально, так и надо писать дальше.

Зарезервированные или ключевые слова записываются в своем определенном регистре, и вы не можете их использовать, давая их названия методам, классам, объектам и т. д. Пробелы между словами не имеют значения, поскольку компилятор их просто игнорирует, но для читабельности кода они важны.

В теле класса `Telefon` имеются два метода: `on ()` - включающий телефон и `off ()` - выключающий телефон. Оба метода имеют свои тела, и в них по идее должен быть какой-то исходный код, описывающий необходимые действия обоих методов. Для нас сейчас не важно, как происходит реализация этих методов, главное - это синтаксис языка Java.

Оба метода имеют круглые скобки `on ()`, внутри которых могут быть записаны параметры, например `on(int time)` или `on(int time, int time1)`. С помощью параметров происходит своего рода связь методов с внешним миром. Говорят,

что метод `on (int time)` принимает параметр `time`. Для чего это нужно? Например, вы хотите, чтобы телефон включился в определенное время. Тогда целочисленное значение в параметре `time` будет передано в тело метода, и на основе полученных данных произойдет включение телефона. Если скобки пусты, то метод не принимает никаких параметров.

1.3.1. Комментарии

В классе `Telefon` в телах обоих методов имеется запись после двух слэшей: `//`. Такая запись обозначает *комментарии*, которые будут игнорироваться компилятором, но нужны для читабельности кода. Чем больше информации вы прокомментируете по ходу написания программы, тем больше у вас будет шансов вспомнить через год, над чем же все это время трудились.

Комментарии в Java могут быть трех видов, это: `//`, `/*...*/` и `/**...*/`. Комментарии, записанные с помощью оператора `//`, должны располагаться в одной строке:

```
// Одна строка
!!! Ошибка! На вторую строку переносить нельзя!
// Первая строка
// Вторая строка
// ...
// Последняя строка
```

Комментарии, использующие операторы `/*...*/`, могут располагаться на нескольких строках. В начале вашего комментария поставьте `/*`, а в конце, когда закончите комментировать код, поставьте оператор `*/`.

Последний вид комментария `/**...*/` используется при документировании кода и также может располагаться на любом количестве строк.

1.3.2. Типы данных Java

Чтобы задать произвольное значение, в Java существуют *типы данных*. В классе `Telefon` мы создали два метода. Оба метода не имели параметров, но когда приводился пример метода `on (int time)` с параметром `time`, говорилось о передаче значения в метод. Данное значение указывало на время, с помощью которого якобы должен включиться телефон. Спецификатор `int` как раз и определяет тип значения `time`. В Java 2 ME шесть типов данных, которые перечислены в табл. 1.1:

Таблица 1.1

Тип	Назначение	Размер в байтах
byte	Маленькое целое	1
short	Короткое целое	2
int	Целое	4
long	Длинное целое	8
char	Символ	2
boolean	Логический тип	

- `byte` - маленькое целочисленное значение от -128 до 128;
- `short` — короткое целое значение в диапазоне от -32 768 до 32 767;
- `int` - содержит любое целочисленное значение от -2 147 483 648 до 2 147 483 647;
- `long` - очень большое целочисленное значение от -922 337 203 685 475 808 до 9 223 372 036 854 775 807;
- `char` - это символьная константа в формате Unicode. Диапазон данного формата - от 0 до 65 536, что равно 256 символам. Любой символ этого типа должен записываться в одинарных кавычках, например: 'G';
- `boolean` - логический тип, имеет всего два значения: `false` -ложь и `true` - истина. Этот тип часто используется в циклах, о которых чуть позже. Смысл очень прост: если у вас в кармане есть деньги, предположительно это `true`, а если нет - то `false`. Таким образом, если деньги имеются - идем в магазин за хлебом или пивом (нужное подчеркнуть), если нет денег - остаемся дома. То есть это такая логическая величина, которая способствует выбору дальнейших действий вашей программы.

Чтобы объявить какое-то необходимое значение, используется запись:

```
int    time;
long   BigTime;
char   word;
```

Оператор точка с запятой необходим после записей и ставится в конце строки. Можно совместить несколько одинаковых по типу объявлений через запятую:

```
mt    time, time1, time2;
```

Теперь давайте усовершенствуем наш класс `Telefon`, добавив в него несколько значений. Методы `on()` и `off()` нам больше не нужны, добавим новые методы, которые действительно могут решать определенные задачи.

```
class Telefon
{
//S — площадь дисплея
//w — ширина дисплея
//h — высота дисплея
int w, h, S;
//метод, вычисляющий площадь дисплея
void Area()
{
    S = w*h;
}
}
```

Итак, мы имеем три переменные `S`, `w` и `h`, отвечающие соответственно за площадь, ширину и высоту дисплея в пикселях. Метод `Area()` вычисляет площадь

экрана телефона в пикселях. Операция бесполезная, но очень показательная и простая в понимании. Тело метода Area () обрело себя и имеет вид $S = w * h$. В этом методе мы просто перемножаем ширину на высоту и присваиваем, или, как еще говорят, сохраняем результат в переменной S. Эта переменная будет содержать значения площади дисплея данного телефона. Сейчас мы подошли вплотную к операторам языка Java, с помощью которых можно совершать всевозможные операции.

1.3.3. Операторы

Операторы языка Java имеют различные назначения. Существуют арифметические операторы, операторы инкремента и декремента, логические операторы и операторы отношения.

Арифметические операторы очень просты и аналогичны операторам умножения «*», деления «/», сложения «+» и вычитания «-», используемым в математике. Существует оператор деления по модулю «%» и слегка запутанная на первый взгляд ситуация с оператором равно «=», Оператор «равно» в языках программирования называется оператором присваивания:

```
int x = 3
```

Здесь вы присваиваете переменной x значение 3. А оператор «равно» в языках программирования соответствует записи двух подряд операторов «равно»: «==». Рассмотрим на примере, что могут делать различные арифметические операторы.

```
int x, y, z;
x = 5;
y = 3;
z = 0;
z = x + y;
```

В данном случае z будет иметь значение уже суммы x и y, то есть 8.

```
x = z * x;
```

Переменная x имела значение 5, но после такой записи предыдущее значение теряется, и записывается произведение $z * x$ ($8 * 5$), что равно 40. Теперь если мы продолжим дальше наш код, то переменные будут иметь такой вид:

```
// x = 40;
// y = 3;
// z = 8;
```

Что касается оператора деления, то поскольку Java 2 ME и конфигурация CLDC 1.0 не поддерживают дробных чисел, то результат такого деления:

```
x = z / y;
```

что равносильно записи:

```
x = 8 / 3;
```

будет равен 2. Дробная часть просто отбрасывается, то же самое происходит при использовании оператора деления по модулю «%». В новой версии CLDC 1.1 реализована поддержка дробных чисел.

Операторы сложения и вычитания имеют те же назначения, что и в математике. Отрицательные числа также родственны.

Операторы декремента «—» и инкремента «++» весьма специфичны, но очень просты. В программировании часто встречаются моменты, когда требуется увеличить или уменьшить значение на единицу. Часто это встречается в циклах. Операция инкремента увеличивает переменную на единицу.

```
int x = 5;
x++;
// Здесь x уже равен 6
```

Операция декремента уменьшает переменную на единицу.

```
int x = 5;
x--;
// x равен 4
```

Операции инкремента и декремента могут быть пост- и префиксными:

```
int x = 5;
int y = 0;
y = x++;
```

В последней строке кода сначала значение x присваивается y, это значение 5, и только потом переменная x увеличивается на единицу. Получается, что:

```
x = 6, y = 5
```

Префиксный инкремент имеет вид:

```
int x = 3;
int y = 0;
y = ++x;
```

И в этом случае сначала переменная x увеличивается на один, а потом присваивает уже увеличенное значение y.

```
y = 4, x = 4
```

Операторы отношения

Операторы отношения позволяют проверить равенство обеих частей выражения. Имеются оператор равенства «==», операторы меньше «<» и больше «>», меньше или равно «<=» и больше или равно «>=», а также оператор отрицания «!=».

```
9 == 10;
```

Это выражение не верно, девять не равно десяти, поэтому значение этого выражения равно false.

9 != 10;

Здесь же, наоборот, оператор отрицания указывает на неравенство выражения, и значение будет равно `true`.

Операторы больше, меньше, больше или равно и меньше или равно аналогичны соответствующим операторам из математики.

Логические операторы

Существует два *логических оператора*. Оператор «И», обозначаемый знаками «&&», и оператор «ИЛИ», обозначенный в виде двух прямых слэшей «||». Например, имеется выражение:

`A*B && B*C;`

В том случае если только обе части выражения истинны, значение выражения считается истинным. Если одна из частей неверна, то значение всего выражения будет ложным.

В противовес оператору «&&» имеется оператор «||», не напрасно имеющий название «ИЛИ».

`A*B || B*C;`

Если любая из частей выражения истинна, то и все выражение считается истинным. Оба оператора можно комбинировать в одном выражении, например:

`A*B || B*C && C*D || B*A;`

С помощью этого выражения я вас ввел, как мне кажется, в затруднение, не правда ли? Дело в том, что в Java, как и в математике, существует приоритет, или так называемая *иерархия операторов*, с помощью которой определяется, какой из операторов главнее, а следовательно, и проверяется первым. Рассмотрим с помощью списка приоритет всех имеющихся операторов языка Java:

[], ., (),
!, ~, ++, --, + (унарный), - (унарный), new,
*, /, %, ,
+, -,
<<, >>, >>>,
<, <=, >, >=,
==, !=,
&, ^, |,

||,
?:,
=, +=, -=, *=, /=, %=, |=, ^=, <<=, >>=, >>>=.

Не со всеми операторами вы еще знакомы, поэтому пугаться не стоит. Ассоциативность операторов в списке следует слева направо и сверху вниз. То есть все, что находится левее и выше, старше по званию и главнее.

1.3.4. Метод *main*

Класс `Telefon`, который мы описывали в предыдущем разделе, имел один метод, с помощью которого вычислялась площадь дисплея. Созданная спецификация класса `Telefon` может быть описана как угодно. Можно добавить методы, реагирующие на обработку событий с клавиатуры телефона, и любые другие методы, которые вы сочтете нужными для описания класса `Telefon`. Таких и подобных классов может быть любое количество. Каждый из классов принято хранить в своем отдельном файле с расширением `*.java` (например: `Telefon.java`).

Все методы, находящиеся в классе `Telefon`, которые вы опишете для данного класса, обязаны производить определенные действия с объектом этого класса, иначе зачем тогда нужны все эти методы. Реализация методов, как уже говорилось, происходит непосредственно в создаваемом классе. Но возникает вопрос: где и как происходит вызов необходимых по ситуации методов или создание объектов используемого класса. В языке Java для этих целей существует *метод* `main()`, который подобно строительной площадке собирает на своей платформе по частям объекты, методы, строя при этом функциональность всей программы.

```
public class RunTelefon
{
    public static void main( String[] args)
    {
        // Работа программы
    }
}
```

Пока не обращайтесь внимания на ключевые слова `public` и `static` - о них чуть позже. Создав класс `RunTelefon` и определив в его теле метод `main()`, мы теперь имеем возможность пользоваться классом `Telefon` и его не особо богатой функциональностью. Объединив эти два класса в один файл либо записав каждый по отдельности, вы получите работоспособную программу. Класс `Telefon` содержит основные данные и методы, а класс `RunTelefon` берет на себя роль мотора. Внутри класса `RunTelefon` происходят создание объектов класса, в данном случае класса `Telefon`, и вызовы соответствующих методов этого класса.

```
class Telefon
{
    // переменные
    int w, h, s;
    // метод
    void Area ()
    {
        S = w*h;
    }
}
```

```
class RunTelefon
{
    public static void main (String args[])
    {
        /* создание объекта/ов класса Telefon и вызовы метода
        Area() */
    }
}
```

Поскольку вы пока не умеете создавать объекты классов и вызывать методы, тело класса `RunTelefon` пустое. Прежде чем идти дальше, необходимо познакомиться с ключевыми словами `public`, `private` и `protected`, а также научиться создавать конструкторы классов.

1.3.5. Закрытые и открытые члены классов

Ключевое слово `public`, объявленное перед методом `main()`, показывает на то, что метод `main()` считается открытым, или, как говорят в любом классе, метод `main()` виден, и к нему можно обратиться. Ключевое слово `public` может назначаться не только методам, но и объектам класса, переменным, любым членам созданного класса. Все объявленные переменные с ключевым словом `public` будут доступны всем другим существующим в программе классам, а это может иногда навредить программе. Например, у вас есть какие-то данные, которые не должны быть доступны другим классам, что тогда? Для этого в языке Java существует еще пара ключевых слов: `private` и `protected`, - благодаря которым вы защищаете переменные или члены классов от общего доступа.

По умолчанию, если вы не используете никаких ключевых слов при объявлении объектов, методов или переменных, язык Java назначает всем членам класса спецификатор `public`. Метод `main()` всегда должен вызываться с ключевым словом `public`, чтобы для всех классов программы метод `main()` был доступен. Как только программа начнет работать, первым делом она ищет метод `main()` и постепенно, шаг за шагом, а точнее, строка за строкой, выполняет все предписанные действия в этом методе.

1.4. Конструктор

Каждый класс обязан содержать конструктор. *Конструктор* - это тот же самый метод, но имеющий название класса, например:

```
class Telefon
{
    Telefon();    // конструктор
    int w, h, s; // переменные
    void Area();  // метод
}
```

Конструктор позволяет инициализировать или создает объекты данного класса с заданными значениями. Каждый класс имеет конструктор, и если вы явно не

записали строку кода (как в нашем случае `Telefon()`), Java автоматически создаст его за вас, и такой конструктор носит название конструктора *по умолчанию*.

Конструкторы в программировании очень полезны, и ни одна профессиональная программа не обходится без конструкторов. Чтобы действительно ощутить мощь конструктора, надо создать конструктор с аргументами, благодаря которым можно инициализировать данные класса.

```
class Telefon
{
    // переменные
    int w, h, s;

    // конструктор класса
    Telefon(int a, int b)
    {
        w = a;
        h = b;
    }

    // метод, вычисляющий площадь дисплея телефона
    void Area ()
    {
        s = w*h;
    }
}
```

При создании объекта (об этом чуть позже) вы можете указать необходимые значения для параметров `a` и `b`, например: `a = 70, b = 100`. Эти заданные числа автоматически присвоятся переменным `w` и `h` при создании объекта класса `Telefon`. Тем самым произойдет инициализация объекта с необходимыми значениями.

Количество конструкторов в классе ограничивается только вашей фантазией и здравым смыслом. Например, можно создать два конструктора классу `Telefon`:

```
Telefon (int a, int b);
Telefon (char a, char b);
```

В этом случае при создании объекта по переданным параметрам конструктору класса компилятор сам выберет необходимый конструктор и создаст заданный объект.

1.5. Объекты классов

Объекты представляют класс, наследуя от своего класса все возможности. Объявить объект очень просто, необходимо вначале указать класс, а потом - объект этого класса.

```
Telefon object;
```

Точно так же, как создается переменная `int`, создается и объект класса. Такая запись создаст пустой объект класса, инициализирующийся значением `null`. Конечно, это еще не полноценный объект, а всего лишь ссылка на данный класс. Чтобы создать полноценный класс, нужно воспользоваться *ключевым словом* `new`, с помощью которого происходит выделение области памяти для содержания создаваемого объекта.

```
Telefon object;  
object = new Telefon ();
```

ИЛИ

```
Telefon object = new Telefon();
```

Обе записи создания объекта класса `Telefon` одинаковы. Чаще всего используется вторая запись, но это можно оставить на усмотрение программиста. Сначала создается пустая ссылка `object`, а потом с помощью ключевого слова `new` выделяется память для объекта, и он инициализируется конструктором класса `Telefon()`. Именно тут и необходим хорошо продуманный конструктор с параметрами, инициализирующий объект с заданными значениями.

Теперь давайте рассмотрим пример сложнее. Создадим класс с конструктором для инициализации высоты и ширины дисплея и в методе `main()` создадим объект класса `Telefon`, вычислив при этом площадь экрана абстрактного телефона.

```
class Telefon  
{  
    int w, h, s;  
  
    // конструктор класса Telefon  
    Telefon (int a, int b)  
    {  
        w = a;  
        h = b;  
    }  
  
    // метод, вычисляющий площадь экрана  
    void Area()  
    {  
        s = w*h;  
    }  
}  
  
// класс RunTelefon может находиться в файле  
RunTelefon.Java  
class RunTelefon
```

```
{
    public static void main (String args[])
    {
        // создаем объект класса Telefon
        Telefon object = new Telefon (70, 90);

        // вычисляем площадь дисплея
        object.Area();
    }
}
```

Разберем весь пример более подробно. Итак, мы создаем класс `Telefon` с тремя переменными: `w`, `h` и `s`, в которых впоследствии будут храниться соответственно ширина, высота и площадь дисплея телефона. Затем мы создаем конструктор класса `Telefon` для инициализации высоты и ширины дисплея. Метод `Area ()` класса `Telefon` вычисляет площадь экрана, сохраняя полученный результат в переменной `s`. После этого идет класс `RunTelefon` с методом `main ()`, являющийся входной точкой для всей программы. Как только написанное приложение начнет работать, программа найдет метод `main()` и начнет выполнять содержимое именно этого метода.

В первой строке кода метода `main()` происходит создание объекта класса `Telefon` со значениями ширины - 70 и высоты - 90 для экрана телефона. Вторая строка кода вызывает метод `Area ()`, вычисляющий площадь дисплея. Метод `Area ()` нельзя просто так вызвать в программе, он должен ссылаться на объект класса `Telefon`. Запись `object.Area ()` как раз и указывает на то, что объект класса `Telefon` вызывает метод `Area ()`. Если имеются несколько объектов класса `Telefon` или, скажем, несколько различных телефонов, тогда вызов метода `Area ()` должен происходить для каждого из объектов класса `Telefon`, например:

```
object1.Area();
object2.Area();
object.area();
```

При условии, что все перечисленные объекты были ранее объявлены и созданы. Для каждого из таких объектов класса `Telefon` будет вычисляться только своя площадь экрана. Давайте рассмотрим еще более сложную программу, создав несколько объектов класса `Telefon`, а заодно используем другую схему работы метода `Area ()`, воспользовавшись *ключевым словом* `return`.

```
class Telefon
{
    int w, h, s, n;

    // конструктор
    Telefon (int a, int b)
    {
        w = a;
```



```
        h = b;
    }

    // вычисляет площадь дисплея
    int Area()
    {
        return w*h;
    }
}

// class RunTelefon может находиться в файле
RunTelefon.java
class RunTelefon
{
    public static void main (String[] args)
    {
        // создадим объект Siemens
        Telefon Siemens = new Telefon (101, 80);

        // создадим объект nokia
        Telefon nokia = new Telefon (128, 128);

        // сохраним полученную площадь в s
        s = Siemens.Area();

        // сохраним полученную площадь в п
        n = nokia.Area();
    }
}
```

В реализации класса `Telefon` изменился только метод `Area()`, использовалось ключевое слово `return`. С его помощью результат умножения высоты на ширину, то есть результат работы всего метода, возвращается для дальнейшего использования в коде программы. В рабочем цикле программы в методе `main()` этот результат сохраняется в двух переменных `s` и `п` для каждого из объектов класса `Telefon`. В данном случае площадь экрана для объекта `siemens` равна $101 \times 80 = 8080$, а для объекта `nokia` $128 \times 128 = 16384$, и оба результата хранятся в различных переменных.

У вас, наверное, сразу возникает вопрос: а как увидеть получившийся результат? Действительно, просчитать значение не составляет труда, но хотелось бы увидеть полученный результат на дисплее. Для этого в Java существует встроенный метод `println()`. Чтобы увидеть на экране результат работы метода `Area()`, нужно добавить вызов метода `println()`.

```
s = Siemens.Area();
System.out.println("Площадь экрана Siemens равна: " + s );
n = nokia.Area();
System.out.println("Площадь экрана nokia равна: " + n );
```

Метод `print()`, как уже говорилось, встроенный и принадлежит классу `System`, поэтому такая запись, `out` - это поток ввода, связывающий программу с консолью. Реально в программировании телефонов метод `println()` используется, как правило, в диагностических целях, но как логическое завершение примера подойдет. С помощью кавычек ограничивается количество выводимого на экран текста, это необходимое условие. Запись `+ s` применяет операцию конкатенации на основе оператора `+`, с помощью которого на экран выводится значение переменной `s`, то есть целочисленное значение, равное площади экрана.

Резюмируя объектно-ориентированное программирование, можно с уверенностью заявить: это не сложно, не страшно, достаточно понятно и, что самое главное, просто в использовании. Создав необходимый класс, объявив объект и воспользовавшись методами созданного класса, вы получаете неограниченные возможности в программировании приложений. Но главное - большое количество необходимых классов и методов уже созданы за вас и собраны в специальные библиотеки, о которых мы поговорим в конце этой главы. Вам же остается только воспользоваться этими готовыми классами, создавая свои объекты этих классов.

1.6. Условные операторы

Когда мы создавали и использовали класс `RunTelefon` с методом `main()`, я говорил, что программа выполняет прописанный код строка за строкой, и это действительно так. В небольших программах, таких, как мы создали, этого вполне достаточно, но в действительно огромных программах подобный подход нелогичен. Нельзя «прыгать» по программе, надо идти от строки к строке. Но если все же требуется перейти в программе к какому-то методу или месту программы, что тогда? Для этого в языке Java существуют *условные операторы*.

Рассмотрим ситуацию, которая более наглядно объяснит смысл операторов. Допустим, вы дома, у вас нет хлеба и надо сбегать в булочную за углом. Прежде чем идти в булочную, вы обязательно проверите, есть ли у вас деньги на покупку хлеба, и только потом отправитесь в магазин за хлебом. Здесь имеет место предусловие наличия денег, выполнение которого приводит вас либо к походу за хлебом, либо к соседу за деньгами. В том случае если не рассматривать это предусловие, то вы бы вышли и пошли в магазин и, придя в магазин, стали бы искать по карманам деньги, которых в карманах как всегда нет. Вот для этого и существуют условные операторы.

В языке Java имеется несколько условных операторов. Первый из них: `if/else` - представляет собой такую сдвоенную конструкцию. Синтаксический шаблон условного оператора `if/else` выглядит следующим образом:

```
if (условие)
{
// действие № 1
}
else
{
// действие № 2
}
```

Переводя эту конструкцию на русский язык, можно сказать: если (if) некое условие верно, то перейти к действию № 1, а иначе (else) выполнить действие № 2. Такая сдвоенная конструкция if/else может быть не обязательна, иногда достаточно лишь записи оператора if. Можно использовать вложенные операторы if/else, например:

```
if (условие 1)
{
    if (условие 2)
    {
        ...
    }
}
```

Здесь проверяется первое условие, и если оно верное, то происходит переход к следующему оператору if, если и это условие верное, то выполняются действия и во втором, и в первом условных операторах. Если же условие во втором операторе if не верно, то выполняются действия только первого оператора if. Условные операторы в программировании на Java используются постоянно, и важно понимать их общий принцип работы.

1.7. Управляющий оператор

В языке Java еще существует *управляющий оператор* switch, который можно в принципе отнести к условным операторам. Он тоже основан на неких условиях, но предоставляет многовариантное ветвление кода для выбора нескольких путей работы программы. Посмотрите на синтаксический шаблон этого оператора:

```
switch (условие)
{
    case 1:
        // действие 1
        break;

    case 2:
        // действие 2
```


Циклы

```
break;
...
case 20:
    // действие 20
    break;

default:
    // действие по умолчанию
    break;
}
```

Вся конструкция оператора `switch` основана на метках `case`, имеющих целочисленные значения. Когда условие, проверяющееся оператором `switch`, совпадает с одним из значений метки `case`, то последующие за меткой действия будут выполнены. Если ни одно из условий не совпало с меткой `case`, то будут выполнены действия, следующие за меткой `default`.

Оператор `break`, использующийся в конце всех действий, для каждой метки очень важен. Когда условие совпадет с одной из меток и начнется выполнение указанных действий для этой метки, выходной точкой в этом блоке кода как раз и служит оператор `break`. Если вы случайно забудете поставить оператор `break`, то ваша программа начнет выполнять следующую метку, сведя тем самым ваши усилия по выбору заданных действий к нулю. Никогда не забывайте ставить оператор `break` для каждой из меток!

1.8. Циклы

Вы задумывались когда-нибудь над тем, зачем вообще существуют программы? Если упростить и обобщить ответ на этот вопрос, то выяснится, что программы необходимы для автоматизации рабочего процесса, увеличения скорости выполняемой работы, избавления человека от тысяч монотонных действий и т. д. Сейчас мы как раз и остановимся на повторяющихся друг за другом действиях.

Цикл - это последовательное выполнение повторяющихся действий на основе заданного предусловия. Например, нужно переставить 100 ящиков из одного угла в другой. Если это каким-то образом записать на языке Java (к сожалению, переставить за вас ящики Java не сможет), то получится запись в 100 строк наподобие этим:

1. взять ящик № 1 и поставить в другой угол;
2. взять ящик № 2 и поставить в другой угол;
3. взять ящик № 3 и поставить в другой угол;
- ...
100. взять ящик № 100 и поставить в другой угол;

Сто строк кода - это уже много, но бывает тысяча, две, три и т. д. Для этих целей, а именно - упрощения записи повторяющихся действий, и служат циклы.

Существуют три оператора, представляющих циклы в языке Java, - это `while`, `do/while` и `for`. Каждый из операторов необходим в своей ситуации, но все же чаще всего используется оператор `for`. Рассмотрим по порядку каждый из операторов.

1.8. f. Оператор *while*

Синтаксическая запись оператора `while` выглядит следующим образом:

```
while(условие)
{
    // действия
}
```

Также имеет место выполнение определенного предусловия, но в отличие от оператора `if/else` данная конструкция построена на циклах проверки условия. Когда программа доходит до оператора `while`, если предложенное условие истинно, происходит выполнение всех действий в блоке из фигурных скобок `{...}`. После выполнения этих действий программа снова делает очередную проверку условия после оператора `while`, и если условие опять истинно, происходит повторное выполнение действий в блоке. Действия в блоке выполняются до тех пор, пока условие не станет ложным, и только тогда происходит выход из цикла `while`.

Для выхода из циклов чаще всего используются так называемые счетчики. Рассмотрим небольшой пример:

```
int i = 0;
while (i < 10)
{
    // действия
    i++;
}
```

Сначала переменной `i` присваивается значение 0, далее происходит проверка условия `i < 10`, если переменная `i` меньше цифры 10, выполняются все действия в блоке цикла. В конце блока переменная `i` увеличивается на 1, и вновь проверяется условие в цикле. И так ровно десять раз для нашего примера. Переменная `i` служит счетчиком для цикла `while`. Отсутствие операции по увеличению переменной `i` на единицу приведет к бесконечному циклу, поскольку 0 всегда будет меньше десяти.

В циклах также можно использовать оператор декремента, например:

```
int i=10;
while (i>0)
{
    // действия
    i--;
}
```

Те же действия, но уже в обратную сторону.

Дополнительно в цикле `while` (да и вовсе других циклах) имеется возможность использования булевых переменных, содержащих значения `false` или `true`. В этом случае происходит проверка определенного предусловия.

```
boolean i = true;
while (i)
{
    // действия.
}
```

Переменная `i` истинна, ей присвоено значение `true`, поэтому происходит выполнение цикла `while`, до тех пор пока переменной `i` не будет присвоено значение `false`. Поэтому необходимо позаботиться о выходе из такого цикла, иначе цикл `while` будет выполняться бесконечно, такие циклы носят названия бесконечных циклов.

Напоследок хочу еще обратить ваше внимание на оператор «равно» `==`. Если записать цикл таким образом:

```
int i = 0
while (i == 5)
{
    // действия
    i++;
}
```

то получится вполне работоспособный цикл, а вот если вы ошибетесь или по привычке воспользуетесь классическим вариантом оператора «равно» `=`, использующимся в математике, то у вас будет проблема в виде бесконечного цикла.

```
int i = 0
while (i = 5)
{
    // действия
    i++;
}
```

В предусловии происходит присвоение переменной `i` значения 5, а это действие не запрещено, и что. мы имеем в итоге? Начнется выполнение этого блока цикла, в конце которого значение `i` увеличится на один, но в предусловии после оператора `while` переменной `i` вновь будет присвоено значение пять, и цикл продолжит свою работу до бесконечности. Это пример простого бесконечного цикла и как следствие классическая ошибка, случающаяся очень часто с начинающими программистами.

1.8.2. Цикл `do/while`

Только что рассмотренный нами цикл `while` при определенных условиях может и не заработать. Например, если условие будет изначально ложно, то цикл не выполнится ни разу. Программа, дойдя до строки кода с оператором `while`, проверит

условие, и если оно будет равно `false`, проигнорирует весь цикл и перейдет к коду, следующему сразу за циклом `while`. Но иногда возникает необходимость в выполнении цикла, по крайней мере, один раз. Для этих целей в Java существует цикл `do/while`. Запись и создание цикла `do/while` осуществляются следующим образом:

```
do
{
// действия
}while (условие)
```

Между операторами `do` и `while` существует тело цикла, которое будет выполняться до тех пор, пока постусловие, следующее за оператором `while`, не будет ложно. Тело цикла выполнится, по меньшей мере, один раз, после чего будет произведена проверка условия. Цикл `do/while` используется нечасто, но порой оказывается незаменим.

1.8.3. Цикл *for*

Это самый распространенный цикл в программировании. Работа цикла `for` основана на управлении счетчиком. Смысл работы этого цикла схож с рассмотренными выше циклами `while` и `do/while`. Посмотрите, как выглядит синтаксическая запись цикла `for`:

```
for (int i = 0; i < 10; i++)
{
// действие
}
```

После ключевого слова `for` следует условие выполнения цикла. Само же условие объединяет в себе три этапа. Сначала следует инициализация счетчика `i = 0`, затем проверка условия `i < 10` и в конце увеличение переменной `i` на единицу.

Работает цикл `for` следующим образом. Когда программа доходит до цикла, то происходит инициализация счетчика `i = 0` и проверяется условие `i < 10`. Далее программа переходит в тело цикла. По окончании всех действий в цикле `for` происходит обращение к третьему этапу цикла: `i++`, увеличивая счетчик на единицу. После чего сразу же происходит переход ко второму этапу - проверке переменной `i < 10` - и повторный выход в тело цикла. Весь цикл продолжается до тех пор, пока условие `i < 10` не станет ложным. Цикл `for` используется постоянно при инициализации массива данных, где без него очень сложно, а порой и невозможно обойтись.

Так же как и в цикле `while`, возможно использование оператора декремента, например:

```
for (int i = 10; i > 0; i--)
{
```

```
//действие  
}
```

Используя любую выбранную конструкцию цикла, вы получаете очень мощный инструмент программирования, избавляющий вас от написания большого количества строк кода.

1.9. Массивы

Очень часто в программировании встречается большое количество однотипных данных. Для того чтобы не объявлять, скажем, 100 переменных, существуют массивы данных. *Массив данных* - это набор однотипных значений, записанных по определенной методике. В языке Java для создания массивов предусмотрена специальная запись.

```
int[] M;
```

С помощью такой записи создается пустая переменная, содержащая неопределенный массив данных. То есть мы создали массив данных, но его размер еще не определен. Инициализация массива происходит так же, как и объявление объекта класса. Необходимо воспользоваться *ключевым словом* new, чтобы выделить память массиву и явно инициализировать его с заданным количеством элементов.

```
int[] M = new int[20]
```

В этой строке кода был создан целочисленный массив, состоящий из 20 элементов. К каждому из элементов массива можно обратиться при помощи индекса M [n] для сохранения либо изъятия значения заданного элемента.

```
M[0] = 3;  
M[1] = 5;  
M[2] = 20;
```

Индекс любого созданного массива всегда начинается с 0, об этом нужно помнить. Чтобы инициализировать массив данных, например, из 50 элементов, вам может потребоваться 50 строк кода, но в нашем распоряжении имеется цикл for, прекрасно справляющийся с этой задачей буквально в несколько строк.

```
int[] M = new int[50];  
for (int i = 0; i<50; i++)  
{  
    M[i] = i;  
}
```

В этом примере происходит инициализация каждого элемента массива от 0 до 49 целочисленными значениями от 0 до 49. Можно инициализировать массив другим способом, без ключевого слова new.

```
int[] M = {0, 1, 2, 3, 4}
```

В этом случае каждый элемент массива инициализируется пятью числами от 0 до 4. Все вышеперечисленные примеры создавали простой одномерный массив данных, но иногда приходится представлять данные в виде парных значений. Например, координаты по X и по Y в системе координат. Для этого в Java существуют *многомерные массивы* данных.

```
int stroka = 10;
int stolbec = 10;
int[][]M = new[stroka][stolbec];
```

Многомерные массивы представлены в виде таблицы. Чтобы получить доступ к заданному элементу массива, нужно указать, в какой строке и в каком столбце находится элемент массива. Точно так же в высшей математике происходит работа с матрицами. Многомерный массив данных можно инициализировать с помощью фигурных скобок:

```
int[][]M =
{
{5, 3, 8},
{7, 12, 16},
{9, 12, 14}
}
```

Инициализация массива данных при помощи цикла `for` немного сложнее, но очень эффективна и используется постоянно. Например, у вас есть большой массив данных, который требуется обнулить. Вот как будет выглядеть запись для этого массива данных:

```
int[][]M = new int[100][50];
for(int i = 0; i < 100; i++)
    for(int a = 0; a < 50; a++)
        M[i][a] = 0;
```

Многомерные массивы позволяют хранить большое количество данных при минимуме записанного кода, сохраняя при этом его понятность и читабельность.

1.10. Наследование

В языке Java имеется действенный инструмент под названием наследование. Это очень мощный инструмент, без которого не обходится ни одна профессионально написанная программа. Каждый из вас, читающий эти строки, обязательно хочет стать профессиональным программистом, поэтому стоит подробно рассмотреть механизм наследования.

Наследование - это механизм, позволяющий наследовать от вышестоящего в иерархии класса все его возможности. Что значит класс, стоящий выше в иерархии? В языке Java существует термин суперкласс и подкласс. Например, имеется некий класс `Siemens`:


```
class Siemens
{
    int w, h;
    int Area()
    {
        return w*h;
    }
}
```

Чтобы создать подкласс класса `Siemens`, необходимо воспользоваться ключевым словом `extends`:

```
class SiemensM55 extends Siemens
{
    // члены класса
}
```

Класс `SiemensM55` является подклассом класса `Siemens`, в свою очередь, класс `Siemens` является суперклассом для класса `SiemensM55`. Класс `SiemensM55` наследует все члены своего суперкласса `Siemens` и имеет возможность доступа к ним при одном условии: все члены суперкласса `Siemens`, к которым вы впоследствии захотите получить доступ, должны быть объявлены со спецификатором `public`. Поскольку мы не определяли вообще никакого спецификатора, а это по умолчанию равно ключевому слову `public`, то подкласс `SiemensM55` будет иметь доступ ко всем членам своего суперкласса `Siemens`.

Например, в рассмотренных ранее примерах, где мы вычисляли площадь дисплея, вы можете воспользоваться методом `Area()` и переменными `w` и `h`. Вам не нужно переопределять переменные и метод суперкласса `Siemens`, но вы можете добавлять дополнительные переменные и методы для подкласса `SiemensM55`, реализующие свои специфические действия для этого подкласса.

Давайте рассмотрим небольшой пример по созданию суперкласса и подкласса. Как вы уже, наверное, догадались, я воспользовался маркой телефона `Siemens M55` для названия подкласса. Чтобы не обижать других производителей телефонов, рассмотрим пример для телефонов компании `Nokia`.

```
//суперкласс Nokia
class Nokia
{
    // высота, ширина и площадь дисплея
    int dh, dw, ds;

    // метод, вычисляющий площадь дисплея
    int Area()
    {
        return dw * dh;
    }
}
```

```
    }
}

// подкласс NokiaE93
class Nokia310 extends Nokia
{
    // высота, ширина и площадь всего телефона
    int th, tw, ts;

    // метод, вычисляющий площадь телефона
    int tArea()
    {
        return tw * th;
    }
}

// входная точка приложения
class RunNokia
{
    public static void main (String args[])
    {
        // при создании объекта используется конструктор
        // по умолчанию
        NokiaE93 nokiaE93 = new NokiaE93();
        // задаем значения для переменных
        nokiaE93.dh = 320
        nokiaE93.dw = 240
        nokiaE93.th = 118
        nokiaE93.tw = 55;
        // вычисляем площадь дисплея
        ds = nokia.dArea();
        // вычисляем площадь телефона
        dt = nokia.tArea();
    }
}
```

Создав подкласс `NokiaE93`, вы получите доступ ко всем членам суперкласса `Nokia`, определенным как `public`. Используя оператор «.» (точка), вы открываете доступ к переменным и методам суперкласса. При создании объекта суперкласса, например `Nokia nokiasuper = new Nokia()`, объект созданного класса может пользоваться своими членами класса, но уже члены подкласса ему не доступны.

Иначе говоря, подкласс расширяет свой суперкласс, добавляя новые члены в подкласс, тем самым подстраивая под себя новые, необходимые только подклассу члены класса, оставляя за собой возможность использования членов суперкласса.

Суперкласс вправе иметь сколько угодно подклассов, например в рассмотренном примере можно создать еще десяток-другой подклассов: NokiaE95, NokiaN80, NokiaN-Gage и т. д. Но каждый из подклассов может иметь только один суперкласс. Множественные наследования в языке Java не предусмотрены. Но каждый из подклассов может иметь свой подкласс, и такая цепочка наследования может длиться до разумной бесконечности.

1.10. 1. Конструктор суперкласса

В рассмотренном примере по созданию суперкласса и подкласса Nokia умышленно использовался конструктор по умолчанию. Ситуация с конструкторами суперклассов требует отдельного внимания.

Когда в программе происходит вызов конструктора, то вызов осуществляется согласно всей иерархии наследования. Первым вызывается конструктор самого первого суперкласса и опускается ниже по всей иерархии наследования. Но иногда необходимо обратиться к ближайшему конструктору суперкласса, и тогда используется ключевое слово `super`. Рассмотрим небольшой пример.

```
// суперкласс Nokia
class Nokia
{
    // конструктор суперкласса Nokia
    // тело класса Nokia
}
// подкласс NokiaSeries60
class NokiaSeries60 extends Nokia
{
    // конструктор подкласса NokiaSeries60
    NokiaSeries60 (int a, int b) ;
    // тело подкласса NokiaSeries60
}
// подкласс Nokia6600 для суперкласса NokiaeSerie.s60
class Nokia6600 extends NokiaSeries60
{
    // конструктор подкласса Nokia6600
    Nokia6600 (short f, short b, short c)
    {
        super (a, b)          // тело конструктора
    }
}
// подкласс Nokia6100 для суперкласса NokiaeSeries60
class Nokia6100 extends NokiaeSeries60
{
    // конструктор подкласса Nokia6100
```

```
Nokia6100 (char a, char b)
{
    super (a, b)
    // тело конструктора
}
// тело подкласса Nokia6100
}
```

Если вы желаете воспользоваться конструктором ближайшего суперкласса, в конструкторе подкласса первым должно идти ключевое слово `super` с параметрами для необходимого конструктора суперкласса. Ключевое слово `super` имеет двойную смысловую нагрузку и может использоваться не только для вызова конструктора суперкласса. Вторым вариантом использования ключевого слова `super` заключается в доступе к невидимому члену суперкласса. Сейчас мы коснулись так называемого вопроса видимости. Вы не задумывались над тем, что будет, если методы или переменные в суперклассе и подклассе будут совпадать по именам?

Дело в том, что если подкласс будет иметь одинаковые названия любых членов класса, то он будет использовать только свои члены класса. Все данные суперкласса окажутся недоступными или невидимыми в силу совпадения названий. Для этих целей может быть использовано слово `super`. Посмотрите на пример, иллюстрирующий способ его использования:

```
// суперкласс Siemens
class Siemens
{
    int a, b;
}
// подкласс SiemensMC62
class SiemensMC62 extends Siemens
{
    int a, b;
    // конструктор подкласса SiemensMC62
    SiemensMC62 (int c, int d)
    {
        super.a = c;           // для a в суперклассе
        super.b = d;           // для b в суперклассе
        a = c;                  // для a подкласса
        b = d;                  // для b подкласса
    }
}
```

Использование ключевого слова `super` очень похоже на обращение объекта класса к методам и переменным. Когда необходимо получить доступ к членам суперкласса, имеющим одинаковые названия, воспользуйтесь ключевым словом `super` либо поступайте еще проще - просто давайте различные имена для членов суперкласса и подкласса.

В языке Java существует еще одно ключевое слово `this`, выполняющее похожие действия. При помощи этого ключевого слова можно произвести вызов любого другого конструктора того же класса либо использовать для указания ссылки на параметр, отсутствующий в конструкторе класса.

Подводя итог темы наследования, необходимо упомянуть о классе `Object`. В языке Java все классы наследуются от большого суперкласса `Object`. Это происходит автоматически, и беспокоиться о явном наследовании при помощи ключевого слова `extends` не нужно.

1.11. Интерфейсы

В программировании мобильных телефонов на Java 2 ME очень часто используются интерфейсы. *Интерфейс* задает классу, что именно должен делать этот класс, но при этом не говоря, каким именно образом это должно быть реализовано. Интерфейс - это некая спецификация, в рамках которой происходит реализация необходимых действий. Создание интерфейса происходит при помощи *ключевого слова* `interface`, а для реализации возможностей интерфейса одним из классов используется *ключевое слово* `implements`. Чтобы более четко разобраться в работе с интерфейсами, рассмотрим небольшой пример:

```
public interface MyInterface
{
    int Inkriment();
}
class MyOne implements MyInterface
{
    int a;
    // реализация метода Inkriment() для класса MyOne
    public int Inkriment()
    {
        a = 9++;
        return a;
    }
}
class MyTwo implements MyInterface
{
    int a;
    // реализация метода Inkriment() для класса MyTwo
    public int Inkriment()
    {
        a = 2++;
        return a;
    }
}
```

Интерфейс задает, что именно надо сделать, а класс, реализующий данный интерфейс, решает, как ему это сделать. Все методы, заключенные в интерфейс, обязательно должны быть созданы в классе, реализующем этот интерфейс. Интерфейсы не являются классами, поэтому создать интерфейс при помощи ключевого слова `new` невозможно, но создавать переменные интерфейса можно, в случае если они сохраняются на объекты класса. Любой интерфейс может наследовать другой интерфейс при помощи *ключевого слова* `extends`. Интерфейсы очень ярко отражают полифонизм языка Java.

1.12. Пакеты

При наличии большого количества своих классов можно создавать *пакеты* для этих классов, как бы классифицируя их в собственную коллекцию или библиотеку классов. В языке Java с помощью *ключевого слова* `package` можно создать пакет для одного и более классов. Для этого в самом начале исходного кода класса сделайте запись `package`. Например, для ранее созданного примера класса `Nokia` это может выглядеть следующим образом:

```
package Nokia;
```

Сохранив такой класс, вы впоследствии можете получить доступ к этому созданному пакету при помощи *спецификатора* `import`.

```
import Nokia.*
```

Такая запись делается в начале исходного кода файла, где вам необходимо использовать пакет с классом `Nokia`. Оператор звездочка в конце импорта класса `Nokia` обозначает доступ ко всем классам этого пакета. Но возможно обращение и к отдельному классу всего пакета, например:

```
import Nokia.NokiaeSeries60.Nokia6600;
```

При условии, конечно, что все эти классы существуют в пакете `Nokia`. Если вы собираетесь использовать много классов из пакета, то лучше воспользоваться оператором звездочка, чем перечислять каждый класс в отдельной строке кода.

Кроме этого, пакеты имеют еще одну ценную возможность - при создании своих классов существует вероятность того, что какой-нибудь программист из Новой Зеландии возьмет и назовет свой созданный класс точно таким же именем. В этом случае возникает конфликт имен, вызывающий исключительную ситуацию. Но если пользоваться возможностью создания пакетов, такая вероятность повторения снижается.

Даже если названия созданных классов будут одинаковыми, содержаться они будут в разных пакетах. Конфликта не возникнет. Слышу провокационный вопрос: а если названия пакетов совпадут? Да, вы правы, есть же сказка про Буратино от двух разных авторов... В таком случае можно воспользоваться рекомендуемой компанией Sun Microsystems, создавшей язык Java, схемой записи по зарезервированному домену в Интернете, например:

```
package ru.gornakov;
```

Или как в случае упомянутой сказки о деревянном человечке:

```
package ru.Buratino;  
package it.Buratino;
```

Язык Java имеет огромное количество предопределенных классов, существующих в виде библиотек, собранных в отдельные пакеты. Каждый из таких пакетов содержит множество классов с различной областью применения. Имеются математические классы, классы, отвечающие за работу с сетью, классы ввода-вывода, классы утилит и т. д. В итоге платформа Java 2 ME состоит из 11 следующих пакетов:

- java.io;
- java.lang;
- java.util;
- javax.microedition.lcdui;
- javax.microedition.lcdui.game;
- javax.microedition.io;
- javax.microedition.media;
- javax.microedition.mediacontrol;
- javax.microedition.pki;
- javax.microedition.midlet;
- javax.microedition.rms.

Более подробно каждый из пакетов будет рассмотрен в *главе 2*. Для того чтобы воспользоваться имеющимися классами и методами, необходимо в начале исходного кода импортировать нужный для работы пакет:

```
import java.lang.*
```

Затем можно воспользоваться, к примеру, методом `abs (int a)` класса `Math`, возвращающим целочисленное значение переменной.

```
int a = 9;  
int x = Math.abs(a);
```

Импортируя пакеты в программу, вы упрощаете разработку приложения, поскольку имеется огромное количество готовых классов и достаточно просто создать объект импортированного класса и пользоваться всеми его методами и возможностями.

<http://palata-x.narod.ru>

Глава 2. Платформа Java 2 Micro Edition

История языка программирования Java насчитывает уже более десятка лет. Вначале, при создании Java, планировалось использовать этот язык для программирования микроконтроллеров бытовых устройств. Поэтому Java (тогда этот язык, правда, имел другое название) изначально создавался независимым от архитектуры, компактным и безопасным, что впоследствии сыграло решающую роль в его широком распространении. Однако в тот момент язык программирования Java не смог обрести популярности. Более того, он оказался совершенно невостребованным. И только благодаря появлению сети World Wide Web язык Java получил настоящее признание, но уже в сфере интернет-программирования.

За этот десяток лет компанией Sun Microsystems было создано несколько платформ для различных сфер деятельности:

- Java 2 Enterprise Edition - эта платформа необходима при создании серверных приложений;
- Java 2 Standard Edition - используется для работы на простых компьютерных системах;
- Java 2 Micro Edition - эта платформа ориентирована на работу с портативными устройствами.

Язык программирования Java независим от архитектуры, в силу того что используется интерпретатор, переводящий байт-код, сгенерированный компилятором, в машинно-независимый код. Интерпретация кода осуществляется под управлением системы выполнения, носящей название *виртуальная Java-машина*. Такой механизм образует среду исполнения приложений. Среда исполнения, в свою очередь, предъявляет определенные требования к свойствам языка программирования Java, построенным на основе спецификации Java Language Specification, разработанной компанией Sun Microsystems. При написании программ на Java активно используется библиотека Java API, без которой написать приложение практически невозможно. Библиотека Java API содержит огромное количество предопределенных интерфейсов, классов, методов, констант, помогающих программисту в минимальные сроки создавать рабочее приложение.

Подобный механизм создания и выполнения программ характерен для всех трех имеющихся платформ Java 2 EE, Java 2 SE и Java 2 ME. Платформы Java 2 EE и Java 2 SE можно признать почти одинаковыми, однако Java 2 EE несколько мощнее и содержит ряд библиотек, позволяющих производить разработку программного обеспечения для серверов, а вот платформа Java 2 ME ориентирована именно на работу с портативными устройствами. В связи с этим Java 2 ME предъявляет уже свои требования к виртуальной Java-машине, свойствам языка Java и библиотекам,

поскольку системные ресурсы портативных устройств ограничены в силу своей спецификации аппаратного обеспечения.

Небольшие размеры портативных устройств накладывают значительные ограничения на процессор, память, дисплей, устройство. Учитывая эти обстоятельства, Java 2 ME была специально разработана для создания программ, которые будут работать именно на мобильных устройствах.

Среда исполнения приложений Java должна находиться внутри портативных устройств или телефона. За это отвечает их производитель, и это как раз и характеризует конкретное устройство как устройство, поддерживающее технологию Java. Количество портативных устройств, поддерживающих Java, растет с каждым днем, но мощность и соответственно возможности устройств различны. Поэтому платформа Java 2 ME разработана в виде блочной модели настраиваемых модулей, конфигураций и профилей.

Конфигурация определяет свойства языка Java и виртуальной Java-машины, а также набор доступных библиотек Java API. Профиль, в свою очередь, предъявляет требования к аппаратной части устройства и может содержать ряд дополнительных библиотек Java, направленных на работу с конкретным портативным устройством.

Платформа Java 2 ME состоит из двух конфигураций: CDC (Connected Device Configuration - конфигурация подключаемых устройств) и CLDC (Connected Limited Device Configuration - конфигурация подключаемых устройств с ограничениями). Каждая из конфигураций определяет свое семейство портативных устройств. При программировании мобильных телефонов используется конфигурация CLDC.

Каждая из конфигураций содержит свои профили, которые настраиваются над своей конкретной конфигурацией. Когда программист создает программное обеспечение для портативного устройства, он обязан четко осознавать, под какой профиль и конфигурацию он пишет программу, а производитель устройства должен осуществить поддержку того или иного профиля в связке со своей конфигурацией.

Как программист вы не можете повлиять на сущность профилей и конфигураций - это жестко заданные спецификации, используемые в программировании портативных устройств. При создании программ вы будете ориентироваться на профиль, устанавливающий требования к аппаратной части устройства. На рис. 2.1 представлена общая схема модульного построения платформы Java 2 ME.

Специально следить за свойствами конфигураций и профиля вам не придется. При создании проекта в любой среде программирования появляется одно или несколько диалоговых окон, в которых посредством галочек или кнопок (в зависимости от реализации) избирается профиль и конфигурация. После чего вам автоматически будет предоставлен доступный набор свойств и средств избранного для разработки профиля и конфигураций.

Цель этой книги - научить вас создавать приложения для мобильных телефонов с применением конфигурации CLDC и профиля MIDP. Вся концепция книги построена на использовании связки MIDP/CLDC, но несколько слов стоит сказать о конфигурации CDC и доступных для этой конфигурации профилях.

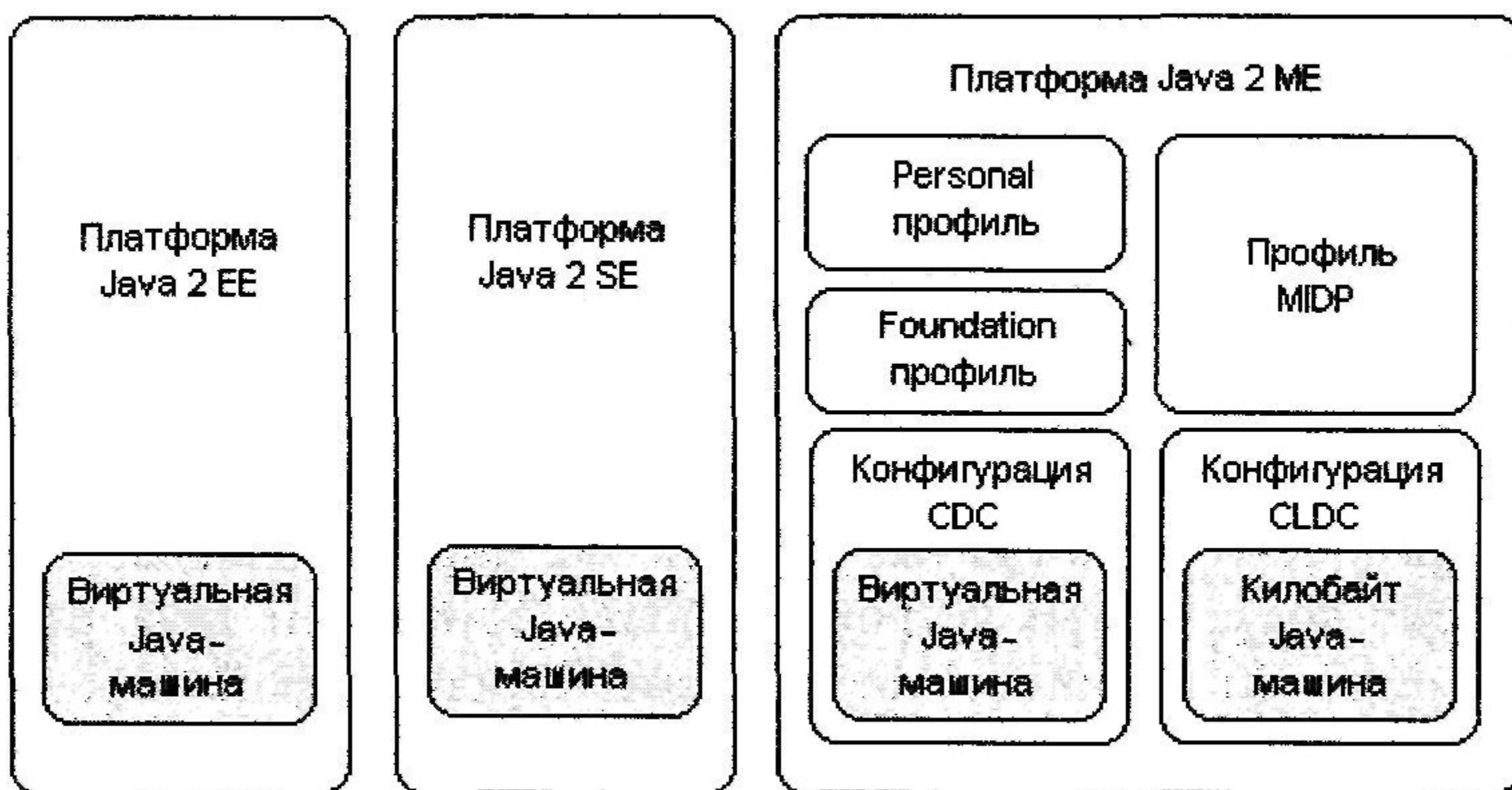


Рис. 2.1. Блочная схема построения платформы Java 2 ME

2.1. Конфигурация CDC

Конфигурация CDC объединяет в себе ряд устройств, имеющих постоянное сетевое соединение, таких как двунаправленные пейджеры, телевизионные приставки, автомобильные системы навигации, интеллектуальные коммуникаторы, КПК и даже ряд мобильных телефонов. Данные устройства характеризуются более мощными системными ресурсами, они имеют обычно 32-разрядные процессоры и как минимум 2 мегабайта памяти. В этой связи обе конфигурации CDC и CLDC имеют свой набор свойств, определяющих поддержку библиотек Java API, виртуальную машину, свойства самого языка Java. Эти свойства и отличают эти две конфигурации между собой.

Как вы уже знаете, каждая из конфигураций построена на основе блочной схемы в виде настраиваемых профилей. Профиль ставит определенные требования к аппаратной части устройства и содержит свой набор API, позволяющий создавать на основе имеющихся библиотек различные приложения. Профили созданы для определенной конфигурации, а приложения пишутся под конкретный профиль. Такая блочная модель позволяет любому приложению работать на портативных устройствах, поддерживающих данный профиль.

Конфигурация может содержать несколько профилей. Конфигурация CDC имеет два профиля - это Foundation Profile и Personal Profile и PDA. Смысл и устройство этих профилей мы рассматривать не будем, они не имеют никакого отношения к теме этой книги и были приведены лишь для понимания общей идеи конфигурации. При желании в документации по Java 2 ME вы сможете найти необходимую информацию и изучить ее самостоятельно.

2.2. Конфигурация CLDC

Конфигурация CLDC рассчитана на семейство мобильных устройств, таких как телефоны, органайзеры, КПК. Мобильные устройства, для которых предназначена конфигурация CLDC, характеризуются следующими параметрами:

- процессор 16- или 32-разрядный;
- память от 160 килобайт, конкретно выделяется под платформу Java 2 ME;
- беспроводное сетевое соединение;
- питание от аккумуляторов.

Все перечисленные характеристики, несомненно, накладывают определенные ограничения на создаваемое приложение. Конфигурации CDC и CLDC независимы друг от друга и не могут использоваться вместе. Вся концепция конфигурации CLDC была разработана дочерней группой Java Community Process компании Sun Microsystems, которая включает в себя множество известных компаний:

- America Online;
- Bull;
- Ericsson;
- Fujitsu;
- Matsushita;
- Mitsubishi;
- Motorola;
- Nokia;
- NTT DoCoMo;
- Oracle;
- Palm Computing;
- RIM;
- Samsung;
- Sharp;
- Siemens;
- Sony;
- Sun Microsystems;
- Symbian.

Конфигурация CLDC содержит ряд классов, интерфейсов, методов платформы Java 2 SE, но в урезанном виде. Это и не мудрено, компьютерная платформа превосходит по мощности мобильные телефоны во много раз. С другой стороны, та простота, с которой можно за несколько дней создать среднее по сложности приложение, подкупает и даже возвращает нас во времена 16-битных приставок. На самом деле общая масса игр, написанных на Java 2 ME, по своему игровому процессу напоминает именно те старые добрые времена.

Конфигурация CLDC также определяется своим набором свойств, состоящих из языка Java, виртуальной Java-машины и библиотек API. В данный момент имеются две версии этой конфигурации- CLDC 1.0 и CLDC 1.1. Конфигурация CLDC 1.1 имеет больше возможностей, например поддержку чисел с плавающей точкой, что соответственно предъявляет более серьезные требования к аппаратной части телефона. Эта конфигурация построена на базе версии конфигурации CLDC 1.0 и просто имеет в своем составе ряд улучшений.

2.2.1. Свойства языка Java

Все свойства языка Java в конфигурации CLDC должны, насколько это возможно, соответствовать спецификации языка Java, но в силу ограниченности системных ресурсов мобильных устройств не поддерживаются следующие свойства, доступные в платформах Java 2 EE и Java 2 SE;

- 1) отсутствует финализация (finalization);
- 2) отсутствует восстановление ошибок после сбоя (error handling).

2.2.2. Виртуальная машина

Виртуальная машина, используемая в конфигурации CLDC, несколько отличается от обычной виртуальной машины, задействованной в Java для компьютерных систем, но обязана оставаться совместимой со спецификацией этой виртуальной машины (Java Virtual Machine Specification). Виртуальная машина находится непосредственно в телефоне, и за совместимостью с общепринятой спецификацией обязаны следить производители мобильных телефонов. Сама же виртуальная машина носит название Kilobyte Virtual Machine (KVM) из-за своей компактности и также имеет ряд недоступных свойств:

- 1) нельзя создать класс-загрузчик (class loader);
- 2) отсутствует механизм отражения (reflection);
- 3) не реализован интерфейс Java Nativ (Java Native Interface);
- 4) не поддерживается финализация (finalization);
- 5) отсутствует восстановление ошибок после сбоя (error handling);
- 6) не поддерживается работа с групповыми потоками (Thread group).

2.3. Профиль

Как уже не раз отмечалось, профиль содержит predetermined требования к аппаратной части устройства, а также включает в себя минимальный набор API, используемый в программировании мобильных устройств. Единственно доступный рабочий профиль в конфигурации **CLDC** имеет название MIDP (Mobile Information Device Profile - информационный профиль мобильных устройств). Спецификация профиля разработана экспертной группой MIDP Expert Group, в состав которой входят следующие компании:

- America Online;
- DDI;
- Ericsson;
- Espial Group, Inc.;
- Fujitsu;
- Hitachi;
- J-Phone;
- Matsushita;
- Mitsubishi;
- Motorola, Inc.;

- NEC;
- Nokia;
- NTT DoCoMo;
- Palm;
- ResearchIn Motion;
- Samsung;
- Sharp;
- Siemens;
- Sony;
- Sun Microsystems, Inc.;
- Symbian;
- Telcordia Technologies.

Профиль MIDP был создан специально для поддержки мобильных устройств и задает следующие технические характеристики для мобильных устройств:

- разрешение экрана как минимум 96 x 54 пикселя с глубиной экрана как минимум 1 бит;
- устройством ввода может быть клавиатура или сенсорный экран;
- как минимум 32 килобайта выделяемой динамической памяти;
- как минимум 128 килобайт под компоненты MIDP;
- как минимум 8 килобайт для хранения постоянных данных;
- беспроводная сеть;
- питание от аккумулятора.

Такое сочетание конфигурации и профиля CLDC/MIDP используется в программировании мобильных телефонов и будет основным сочетанием при рассмотрении примеров из книги.

На данный момент профиль MIDP имеет две версии: MIDP 1.0 и MIDP 2.0. До последнего времени первая версия MIDP была основным профилем при создании приложений для телефонов. Все телефоны, поддерживающие Java, имеют совместимость с *профилем MIDP 1.0*. Этот профиль был сформирован при начальном создании платформы Java 2 ME и имеет в своем составе определенный набор API.

С выходом профиля MIDP 2.0 добавился ряд новых библиотек, значительно улучшающих создание приложений для мобильных телефонов. Но самое главное - это то, что у него имеется полная совместимость с профилем MIDP 1.0. Профиль MIDP 2.0 содержит большое количество новых дополнительных библиотек, отсутствующих в составе MIDP 1.0, но при создании приложений под профиль MIDP 2.0 можно пользоваться библиотеками профиля MIDP 1.0. Если же вы пишете программу под профиль MIDP 1.0, то библиотеки профиля MIDP 2.0 вам будут недоступны. В книге будут рассмотрены оба профиля как единое целое.

Подытожив все вышесказанное о профилях и конфигурациях, необходимых для программирования мобильных телефонов, можно резюмировать: программный продукт, создаваемый разработчиками, ориентирован на конкретный профиль, который является спецификацией, устанавливающей определенные требования к аппаратной части телефона, а также содержит дополнительные библиотеки. Каждый конкретный профиль надстраивается над своей и только ему доступной конфигурацией.

Конфигурация предъявляет требования к виртуальной Java-машине и свойствам языка Java, используемым в этой конфигурации. Далее идет плотное взаимодействие с аппаратным обеспечением телефона через имеющиеся сервисы, которые предоставляются операционной системой либо прошивкой телефона. Благодаря такой цепочке взаимодействий любое программное обеспечение, написанное на языке Java под конкретный профиль, будет работать на телефоне с поддержкой Java. На рис. 2.2 хорошо прослеживается общая схема взаимодействия приложения с мобильным телефоном.

Такая модульность в построении Java 2 ME дает неограниченную возможность в модернизации всей платформы и написания действительно аппаратно-независимого кода программы. Люди, знакомые с моей книгой «DirectX 9. Уроки программирования на C++», обязательно найдут много общего в подходе реализации двух платформ DirectX и Java 2 ME. Оставшаяся часть этой главы целиком посвящена рассмотрению пакетов и классов, доступных в MIDP 2.0/CLDC 1.1. Будут затронуты практически все имеющиеся компоненты данного профиля и конфигурации.

2.4. Профиль MIDP 2.0 и конфигурация CLDC 1.1

Язык Java - самый «библиотечный язык», такого количества продуманных классов, наверное, нет ни в одном языке программирования. С другой стороны, простота в использовании Java, по всей видимости, поспособствовала определенной популярности этого языка. С точки зрения программиста, обилие готовых классов гораздо упрощает разработку программного продукта и, что самое главное, уменьшает сроки создания программ.

Профиль MIDP2.0 и конфигурация CLDC 1.1 содержат большое количество интерфейсов и классов, использование которых в программировании приложений, пожалуй, сможет удовлетворить любого разработчика. Часть классов была взята из Java 2 SE с некоторыми усечениями, а часть была специально написана под Java 2 ME. Вся библиотека, доступная для профиля MIDP 2.0 и конфигурации CLDC 1.1, состоит из 11 пакетов. По традиции, каждый отвечает за свою определенную область:

- java.lang;
- java.util;
- java.io;
- javax.microedition.lcdui;
- javax.microedition.lcdui.game;
- javax.microedition.io;

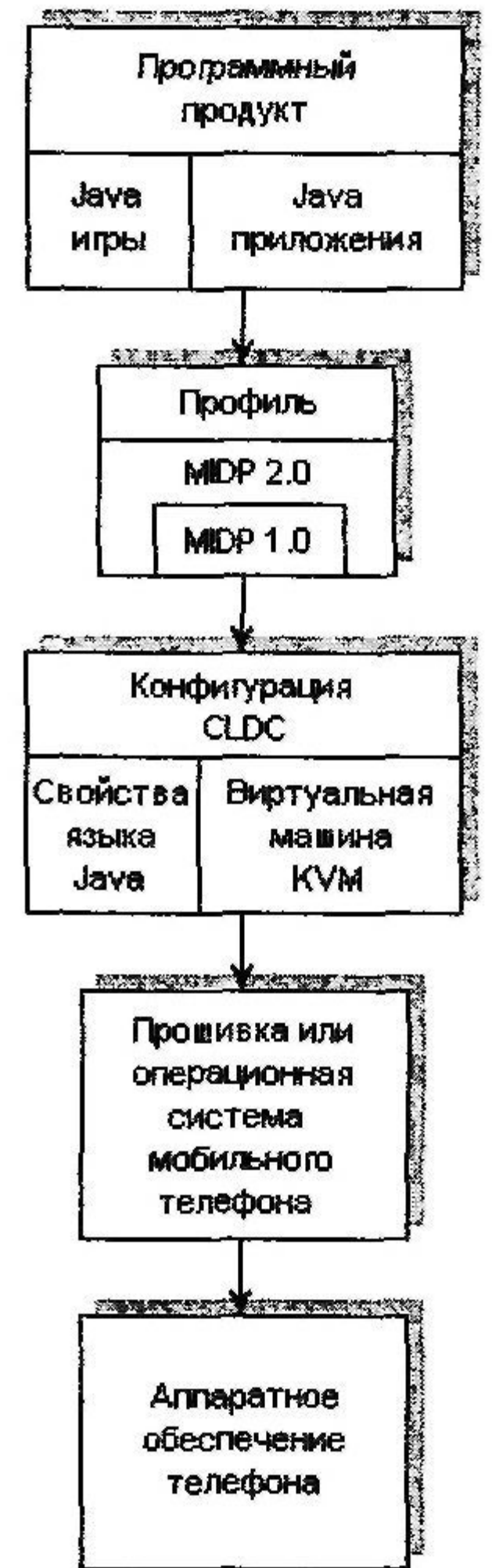


Рис. 2.2. Схема взаимодействия приложений с аппаратным обеспечением телефона

- javax.microedition.media;
- javax.microedition.media.control;
- javax.microedition.pki;
- javax.microedition.midlet;
- javax.microedition.rms.

Все пакеты с префиксами javax.microedition.* написаны специально для Java 2 ME профиля MIDP 2.0. Пакеты с префиксом Java.* взяты из Java 2 SE версии 1.4 в урезанном виде, имеют полную совместимость с оригиналом и определены в конфигурации CLDC 1.1.

2.4.7. Пакет Java .lang

Этот пакет содержит системные классы, или основы языка Java, и исключения. Имеется также один-единственный интерфейс Runnable. На рис. 2.3 изображена иерархия классов пакета java.lang.

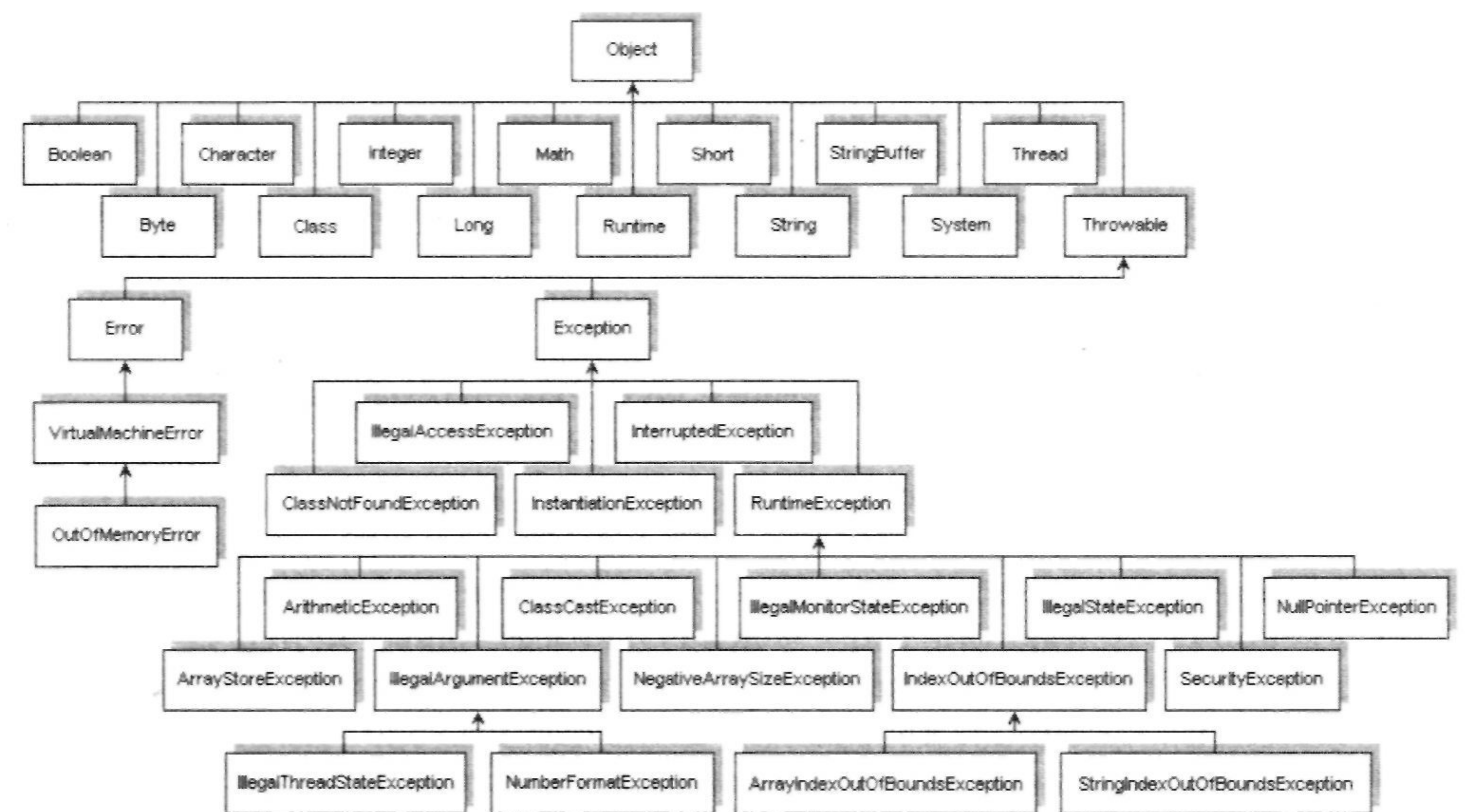


Рис. 2.3. Иерархия классов пакета java.lang

Рассмотрим имеющиеся компоненты пакета java.lang и дадим краткую характеристику каждому из них.

Интерфейс

- Runnable - создает поток в приложении.

Классы

- Boolean - объектно-ориентированный класс, оболочка, или, как еще говорят, «обертка», для простого типа Boolean;
- Byte - объектно-ориентированный класс для простого типа Byte;
- Character - объектно-ориентированный класс для простого типа Char;

- `Class` - виртуальная машина создает объекты этого класса, которые представляют интерфейсы и классы языка Java;
- `Integer` - объектно-ориентированный класс для простого типа `int`;
- `Long` - объектно-ориентированный класс, оболочка для простого типа;
- `Math` - класс, содержащий математические методы;
- `Object` - суперкласс для всех классов Java. Все классы наследуются от класса `Object` и являются его подклассами;
- `Runtime` - класс времени исполнения;
- `Short` - объектно-ориентированный класс, оболочка для простого типа `Short`;
- `String` - создает строки символов;
- `StringBuffer` - содержит строку символов любого размера;
- `System` - содержит ряд системных методов;
- `Thread` - создает поток в работе приложения;
- `Throwable` - суперкласс для всех подклассов, предназначенных для работы с ошибками и исключениями.

Исключения

- `Exceptions` - исключения для классов и подклассов;
- `ArithmeticException` - арифметическое исключение;
- `ArrayIndexOutOfBoundsException` - исключение, обрабатывающее неправильный индекс в массиве данных;
- `ArrayStoreException` - исключение, обрабатывающее неправильно заданный тип объекта в массиве объектов;
- `ClassCastException` - неправильно указан подкласс объекта;
- `ClassNotFoundException` - класс не найден;
- `IllegalAccessException` - нет доступа к классу;
- `IllegalArgumentException` - указан неправильный аргумент;
- `IllegalMonitorStateException` - мониторинг объектов;
- `IllegalStateException` - неправильно вызванный метод;
- `IllegalThreadStateException` - неправильные установки потока;
- `IndexOutOfBoundsException` - исключает неверно указанный индекс;
- `InstantiationException` - исключает ситуацию в создании или вызове членов абстрактного класса;
- `InterruptedException` - исключает прерывание потока, находящегося в состоянии ожидания;
- `NegativeArraySizeException` - исключает ситуацию в создании большего размера массива данных, чем было указано при инициализации;
- `NumberFormatException` - неправильное преобразование строки в целочисленный тип данных;
- `RuntimeException` - суперкласс исключений времени исполнения виртуальной машины Java;
- `SecurityException` - менеджер безопасности;
- `StringIndexOutOfBoundsException` - выход индекса за пределы строки.

Ошибки

- `Error` - обобщенная модель ошибок;
- `OutOfMemoryError` - ошибки, связанные с выходом за пределы памяти;
- `VirtualMachineError` - ошибка времени исполнения.

2.4.2. Пакет *Java, util*

В этом пакете содержатся классы стандартных утилит, упрощающих работу программиста. Пакет сильно урезан по сравнению со стандартным пакетом `Java 2 SE`. На рис. 2.4 представлена иерархия классов пакета *java.util*.

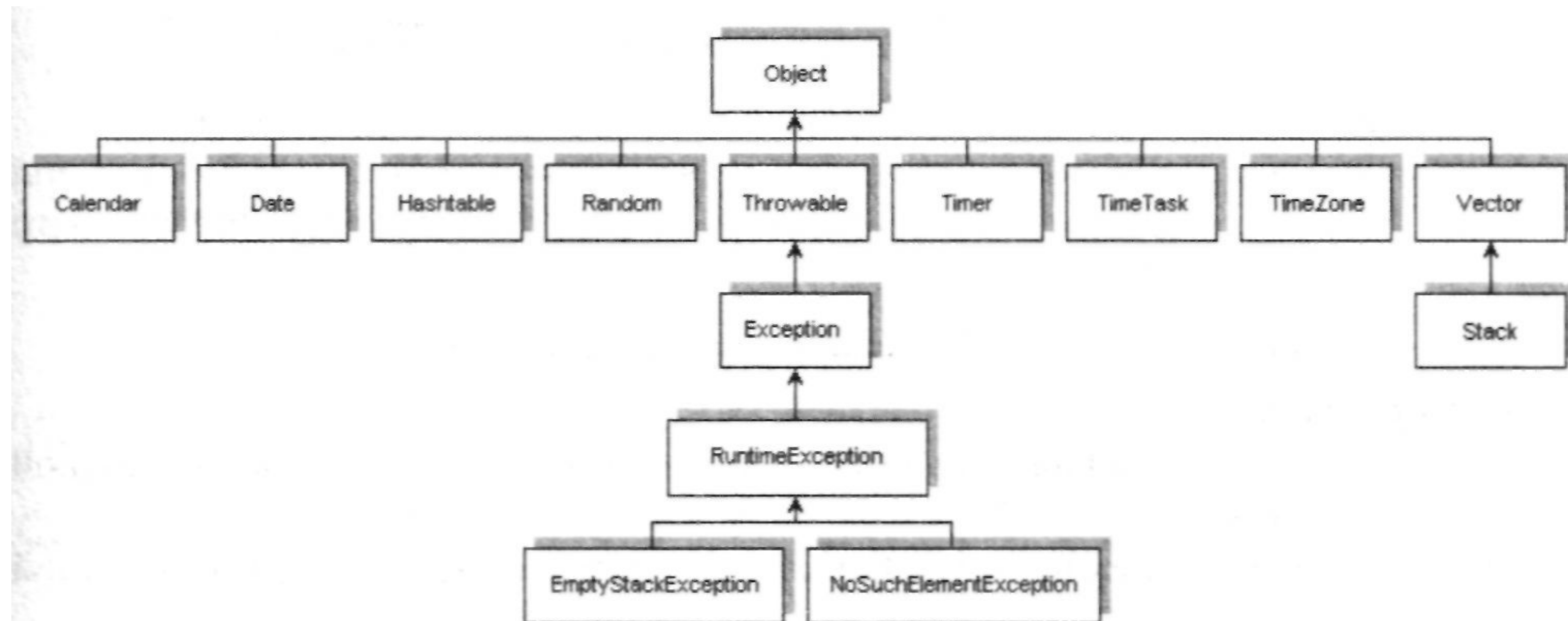


РИС. 2.4. Иерархия классов пакета `java.util`

Интерфейс

- `Enumeration` - декларирует возможность доступа к элементам.

Классы

- `Calendar` - выполняет функции обыкновенного календаря;
- `Date` - реализует возможность работы с датой и временем;
- `Hashtable` - имеет возможность в сохранении объектов с доступом к ним по определенно заданному ключу;
- `Random` - генератор случайных чисел;
- `Stack` - реализует функциональность стека;
- `Timer` - реализует возможность работы со временем;
- `TimerTask` - планировщик задач;
- `TimeZone` - дает возможность в определении временного пояса;
- `Vector` - класс для создания и содержания массивов любого размера. Имеет возможность изменять размер заданного массива.

Исключения

- `EmptyStackException` - указывает на пустой стек;
- `NoSuchElementException` - исключение указывает на отсутствие элементов в определенном перечислении.

2.4.3. Пакет *Java.io*

Классы этого пакета отвечают за работу с входными и выходными потоками данных. На рис. 2.5 показана иерархия наследования классов *пакета Java.io*.

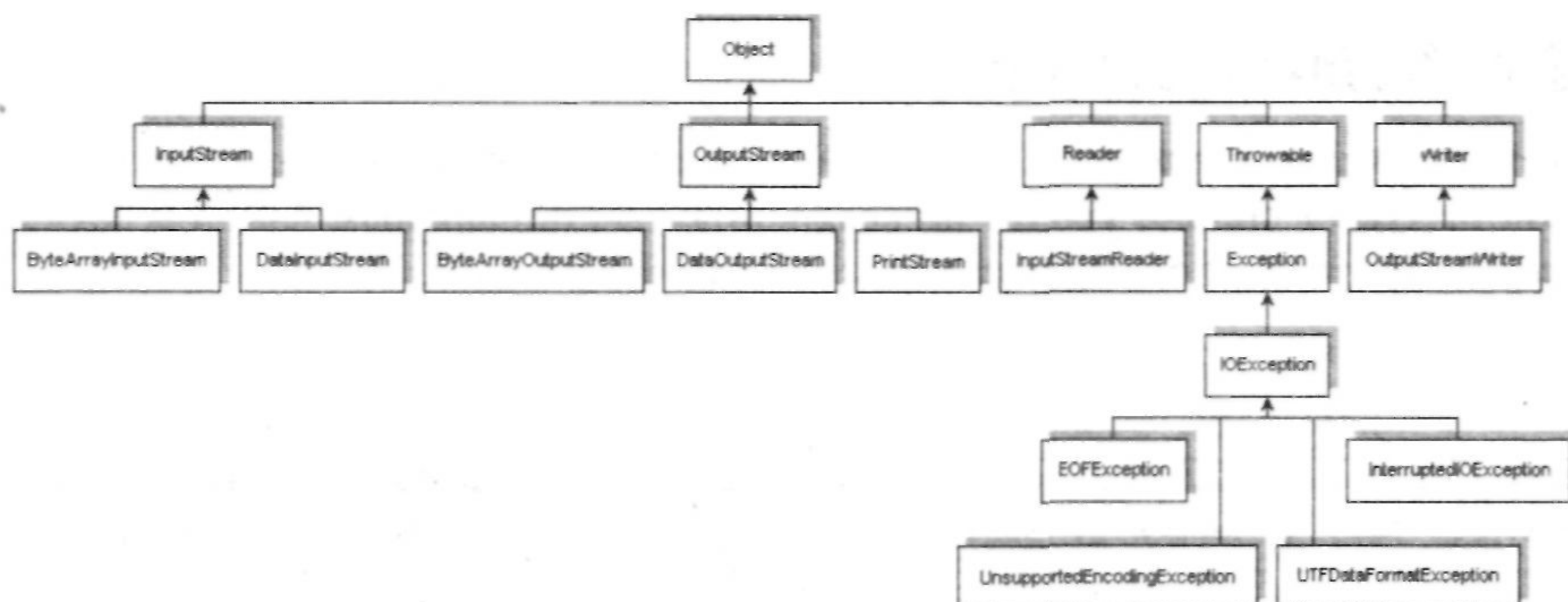


Рис. 2.5. Иерархия пакета *Java.io*

Интерфейсы

- *DataInput* - декларирует методы для чтения простых типов во входной поток данных;
- *DataOutput* - декларирует методы для записи простых типов в выходной поток данных.

Классы

- *ByteArrayInputStream* - необходим при чтении входного потока байт из массива данных для дальнейшего размещения их в памяти;
- *ByteArrayOutputStream* - необходим при записи потока байт из памяти в массив выходных данных;
- *DataInputStream* - этот класс должен наследоваться от интерфейса *DataInput*, реализуя при этом все его методы;
- *DataOutputStream* - класс должен наследоваться от интерфейса *DataOutput*, реализуя при этом все его методы;
- *InputStream* - абстрактный класс, предназначенный для работы с входным потоком байтов;
- *InputStreamReader* - наследуется от класса *Reader*, реализуя методы для чтения символьных данных входного потока с перекодировкой;
- *OutputStream* - абстрактный класс, предназначенный для работы с выходным потоком байт;
- *OutputStreamWriter* - наследуется от класса *Writer*, реализуя методы для записи символьных данных в выходной поток с перекодировкой;
- *PrintStream* - расширяет выходной поток способностью печати данных;
- *Reader* - абстрактный класс, предназначенный для чтения символьных данных входного потока;
- *Writer* - абстрактный класс, предназначенный для записи символьных данных в выходной поток.

Исключения

- EOFException - сигнализирует о конце файла;
- InterruptedIOException - сигнализирует о прерванном действии по вводу-выводу;
- IOException - указывает на исключение ввода-вывода;
- UnsupportedEncodingException - указывает на невозможность перекодировки;
- UTFDataFormatException - сигнализирует о прочтении строки формата UTF-8.

2.4.4. Пакет javax.microedition.io

Этот пакет содержит множество интерфейсов и всего два класса, обеспечивающих связь с сетью. На рис. 2.6 и рис. 2.7 приводится общая схема наследования соответственно интерфейсов и классов пакета javax.microedition.io.

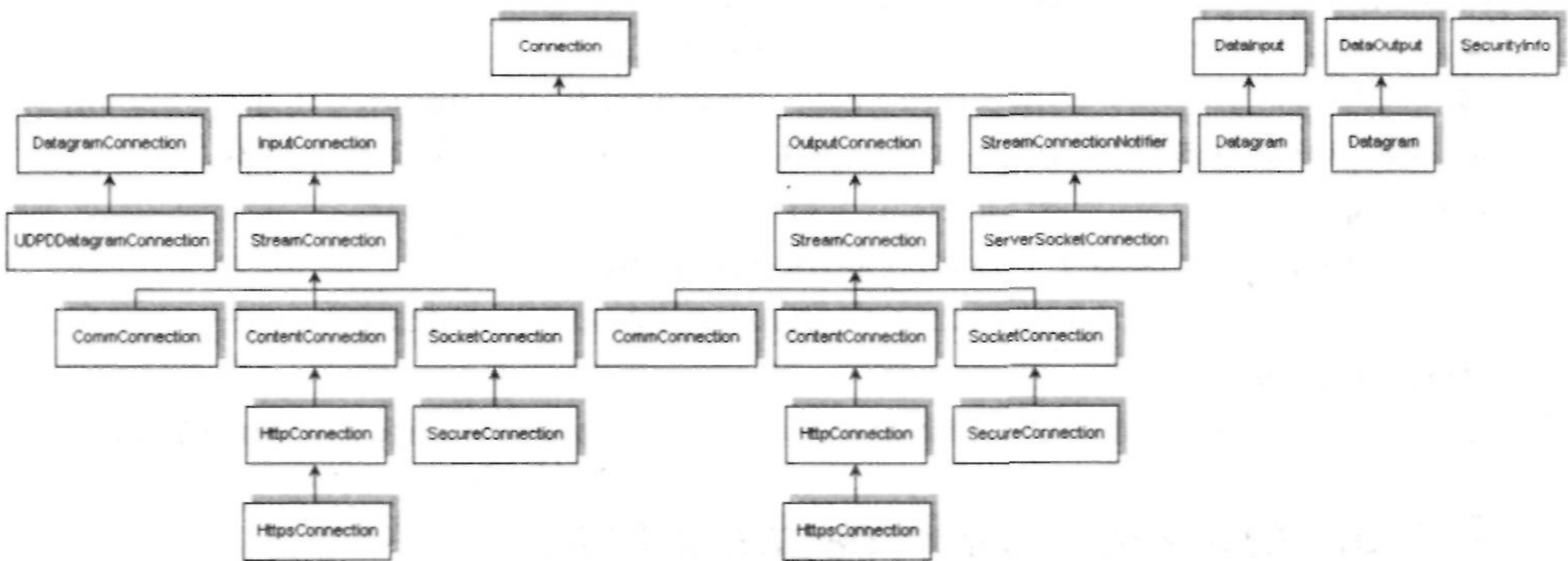


РИС. 2.6. Иерархия интерфейсов пакета javax.microedition.io

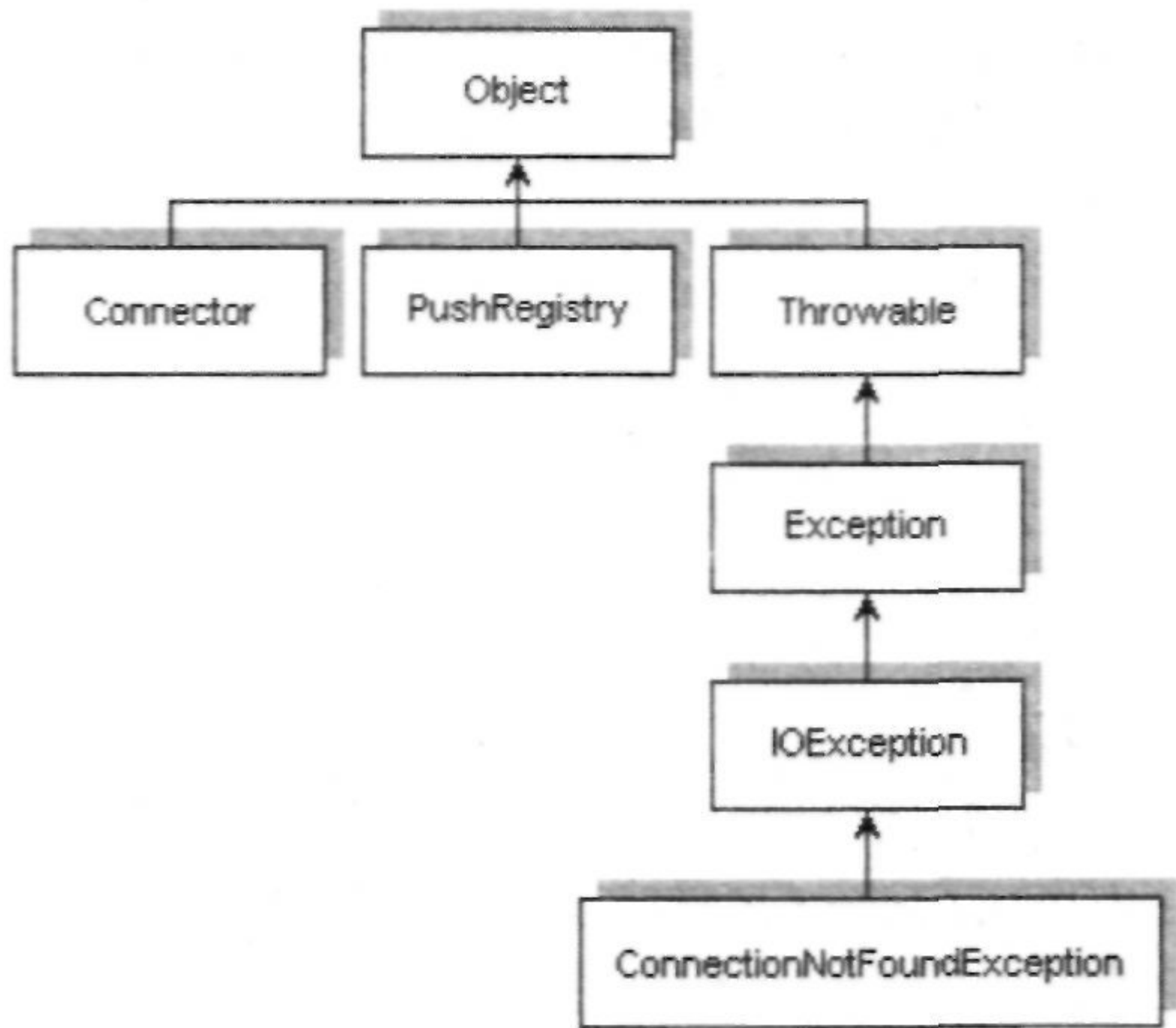


РИС. 2.7. Иерархия классов пакета javax.microedition.io

Интерфейсы

- `CorranConnection` - находит последовательный порт;
- `Connection` - **ОБЩИЙ** тип всей связи сети;
- `ContentConnection` - находит связь с потоком;
- `Datagram` - **ОБЩИЙ** интерфейс дейтограммы;
- `DatagramConnection` - определяет возможность связи дейтограммы;
- `HttpConnection` - декларирует методы константы для http-соединения;
- `HttpsConnection` - декларирует методы константы для безопасного http-соединения;
- `InputConnection` - интерфейс для создания входной связи с сетью;
- `OutputConnection` - интерфейс для создания выходной связи с сетью;
- `SecureConnection` - определяет безопасную связь с сетью;
- `SecurityInfo` - располагает методами для получения информации сетевой связи;
- `ServerSocketConnection` - реализует связь с сервером;
- `SocketConnection` - находит socket (сокет) для потока связи;
- `StreamConnection` - **СВЯЗЬ** с потоком;
- `StreamConnectionNotifier` - определяет возможность всей связи;
- `UDPDatagramConnection` - реализует связь с дейтограммой.

Классы

- `Connector` - класс для создания объектов связи;
- `PushRegistry` - класс для поддержания списков связей.

Исключение

- `ConnectionNotFoundException` - указывает на отсутствие связи.

2.4.5. Пакет *javax.microedition.lcdui*

Данный пакет имеет разнообразные классы для реализации пользовательского интерфейса в мобильных приложениях. Существует большое количество классов, благодаря которым можно создать действительно красивое интерактивное приложение. На рис. 2.8 изображена иерархия классов пакета *javax.microedition.lcdui*.

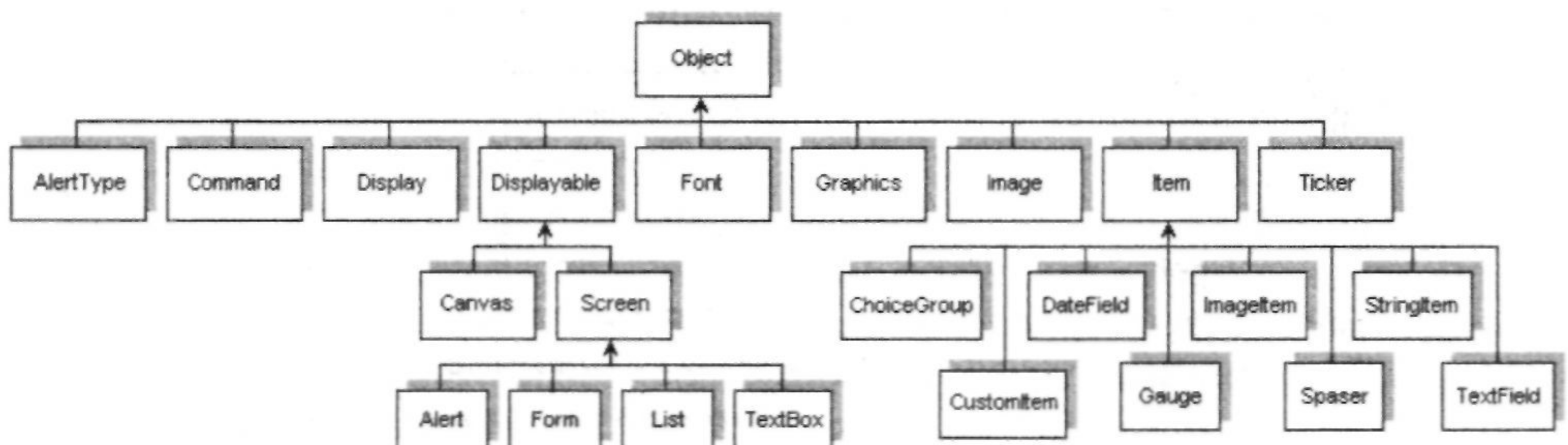


РИС. 2.8. Иерархия классов пакета *javax.microedition.lcdui*

Интерфейсы

- `Choice` - содержит набор библиотек, создающих возможность выбора заданных элементов;
- `CommandListener` - реализует возможность получения событий;
- `ItemCommandListener` - реализует возможность получения событий от объектов класса `Item`;
- `ItemStateListener` - используется при получении событий о состоянии объектов класса `Item`, встроенных в `Form`.

Классы

- `Alert` - этот класс необходим при создании уведомлений об ошибках либо информационных сообщений;
- `AlertType` - отображает тип ошибки;
- `Canvas` - абстрактный класс, обеспечивает графическую прорисовку различных элементов на экране телефона;
- `ChoiceGroup` - встраиваемая группа выбираемых элементов. Интегрируется в класс `Form`, наследуется от класса `Item` и реализует интерфейс `Choice`;
- `Command` - инкапсулирует командные действия, при этом не определяя фактических действий команды, а лишь содержит информацию;
- `CustomItem` - создает возможность в отображении новых графических элементов, встроенных в класс `Form`;
- `DateField` - класс, представляющий работу с датой и временем. Интегрируется в класс `Form`, наследуется от класса `Item`;
- `Display` - это класс-диспетчер, отвечающий за экран телефона;
- `Displayable` - абстрактный класс, содержит иерархию классов пользовательского интерфейса;
- `Font` - класс шрифтов;
- `Form` - этот класс создает пустую форму, в которую впоследствии можно встраивать ряд классов, задающих пользовательский интерфейс всего приложения;
- `Gauge` - показывает графическое течение процесса;
- `Graphics` - предоставляет возможность в рисовании на экране телефона;
- `Image` - класс, отвечающий за загрузку и отображение любых видов изображений формата PNG;
- `ImageItem` - контейнер для загруженных в приложение изображений;
- `Item` - суперкласс, содержащий ряд классов для их дальнейшей интеграции в класс `Form`;
- `List` - создает список группы элементов;
- `Screen` - суперкласс для всех высокоуровневых классов, определяющих пользовательский интерфейс приложения;
- `Spacer` - создает заданное пространство на экране;
- `StringItem` - дает возможность в создании массивов строк;
- `TextBox` - создает редактируемый текстовый контейнер;
- `TextField` - создает редактируемый текстовый контейнер, который встраивается в класс `Form`;
- `Ticker` - создает в приложении бегущую строку текста.

2.4.6. Пакет *javax.microedition.lcdui.game*

Этот новый игровой пакет добавлен в профиль MIDP 2.0. В состав пакета входят пять мощных и хорошо продуманных классов, с помощью которых можно достаточно легко создавать игры для мобильных устройств. На рис. 2.9 показана иерархия классов пакета *javax.microedition.lcdui.game*.

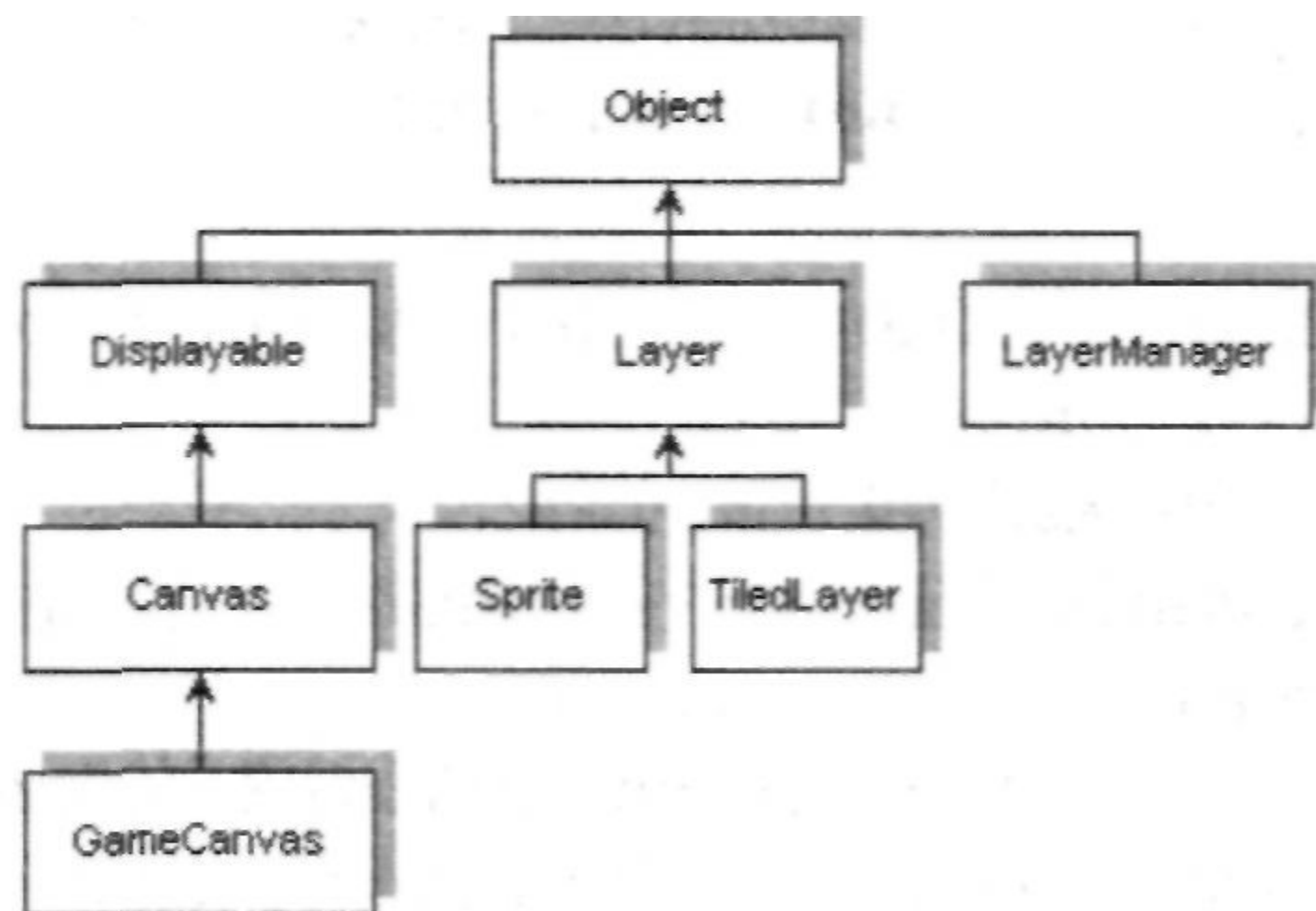


Рис. 2.9. Иерархия классов пакета *javax.microedition.lcdui.game*

Классы

- *GameCanvas* - абстрактный класс, содержащий основные элементы игрового интерфейса;
- *Layer* - абстрактный класс, отвечающий за уровни, представляемые в игре;
- *LayerManager* - менеджер уровней;
- *Sprite* - создает анимационные фреймы;
- *TiledLayer* - отвечает за создание фоновых изображений.

2.4.7. Пакет *javax.microedition.media*

Пакет добавлен в профиль MIDP 2.0 и служит для создания звукового сопровождения в приложении. Пакет разработан специальной экспертной группой (MMAPI Expert Group), в состав которой входят такие известные компании, как:

- Nokia (Specification Lead);
- Aplix Corporation;
- Beatnik, Inc.;
- France Telecom;
- Insignia Solutions;
- Mitsubishi Electric Corp.;
- Motorola;
- Netdecisions Holdings United;
- NTT DoCoMo, Inc.;
- Openwave Systems Inc.;
- Packet Video Corporation;

- Philips;
- Siemens AG ICM MP TI;
- Smart Fusion;
- Sun Microsystems, Inc.;
- SymbianLtd;
- Texas Instruments Inc.;
- Vodafone;
- Yamaha Corporation;
- Zucotto Wireless.

В профиле MIDP 1.0 отсутствует возможность полноценной работы со звуком, и каждый из производителей предоставлял свои библиотеки для этих целей. В профиле MIDP 2.0 такой необходимости уже нет и можно воспользоваться любым необходимым классом и интерфейсом из пакета *javax.microedition.media*. На рис. 2.10 приводится наследование интерфейсов этого пакета.

Интерфейсы

- Control - осуществляет контроль над процессами;
- Controllable - осуществляет контроль над объектами;
- Player - реализует контроль над воспроизведением;
- PlayerListener - необходим для получения асинхронных событий, принятых от проигрывателя.

Классы

- Manager - менеджер системных ресурсов.

Исключение

- MediaException - исключает ошибки в работе методов этого пакета.

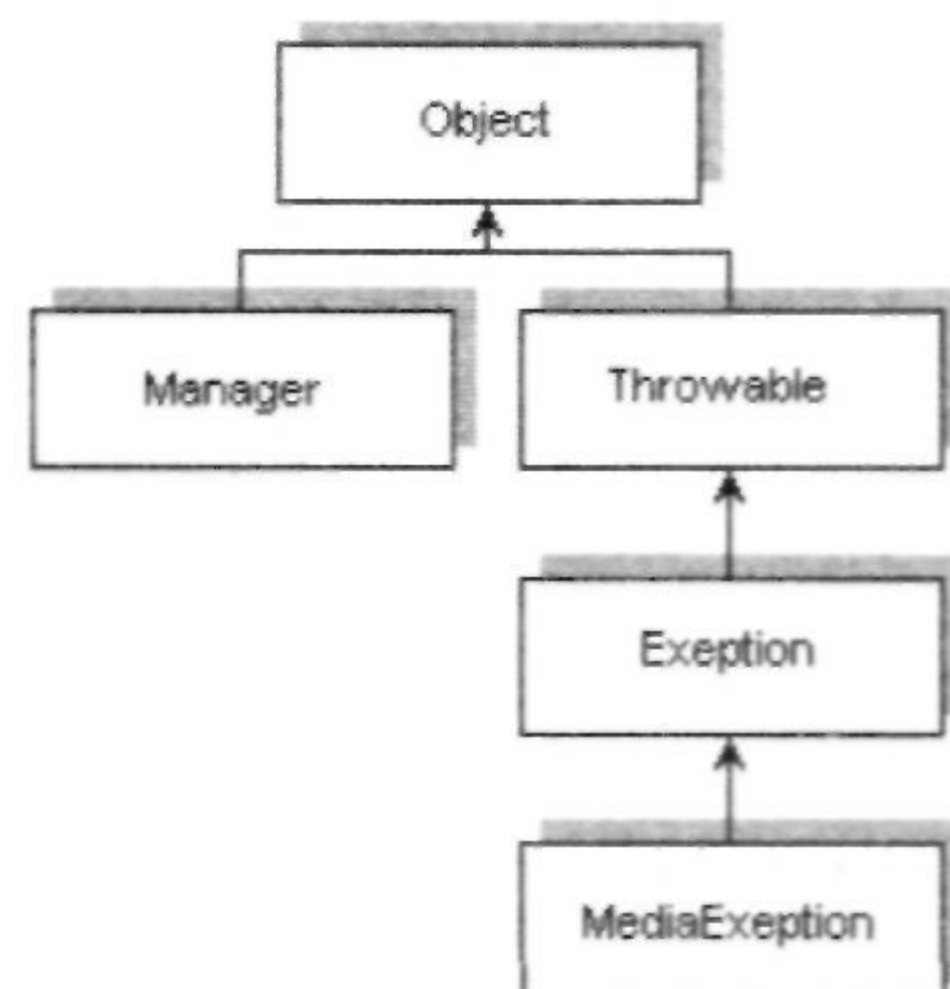


Рис. 2.10. Иерархия интерфейсов из пакета *javax.microedition.media*

2 А Я. Пакет *javax.microedition.media.control*

С помощью пакета *javax.microedition.media.control* определяется контроль над воспроизведением заданных звуковых данных. Это небольшой пакет, имеющий в своем составе всего два интерфейса, а на рис. 2.11 дается схема наследования интерфейсов.

Интерфейсы

- ToneControl - воспроизведение однотоновых звуков;
- VolumeControl - регулирует громкость воспроизведения.

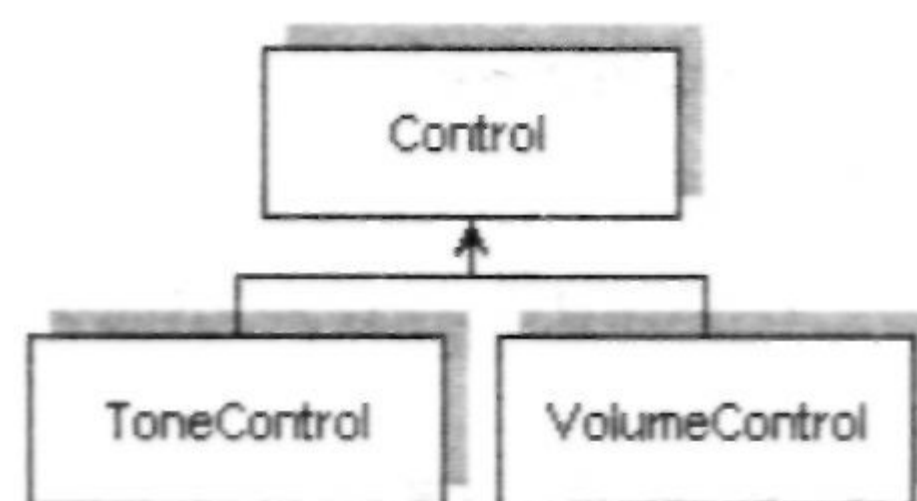


Рис. 2.11. Иерархия интерфейсов пакета *javax.microedition.media.control*

2.4.9. Пакет *javax.microedition.midlet*

Сам по себе пакет небольшой, но он играет ключевую роль при создании приложений на Java 2 ME. С помощью этого пакета происходит связь между приложением и мобильным информационным профилем устройства (MIDP). Рисунок 2.12 отражает полную иерархию пакета *javax.microedition.midlet*

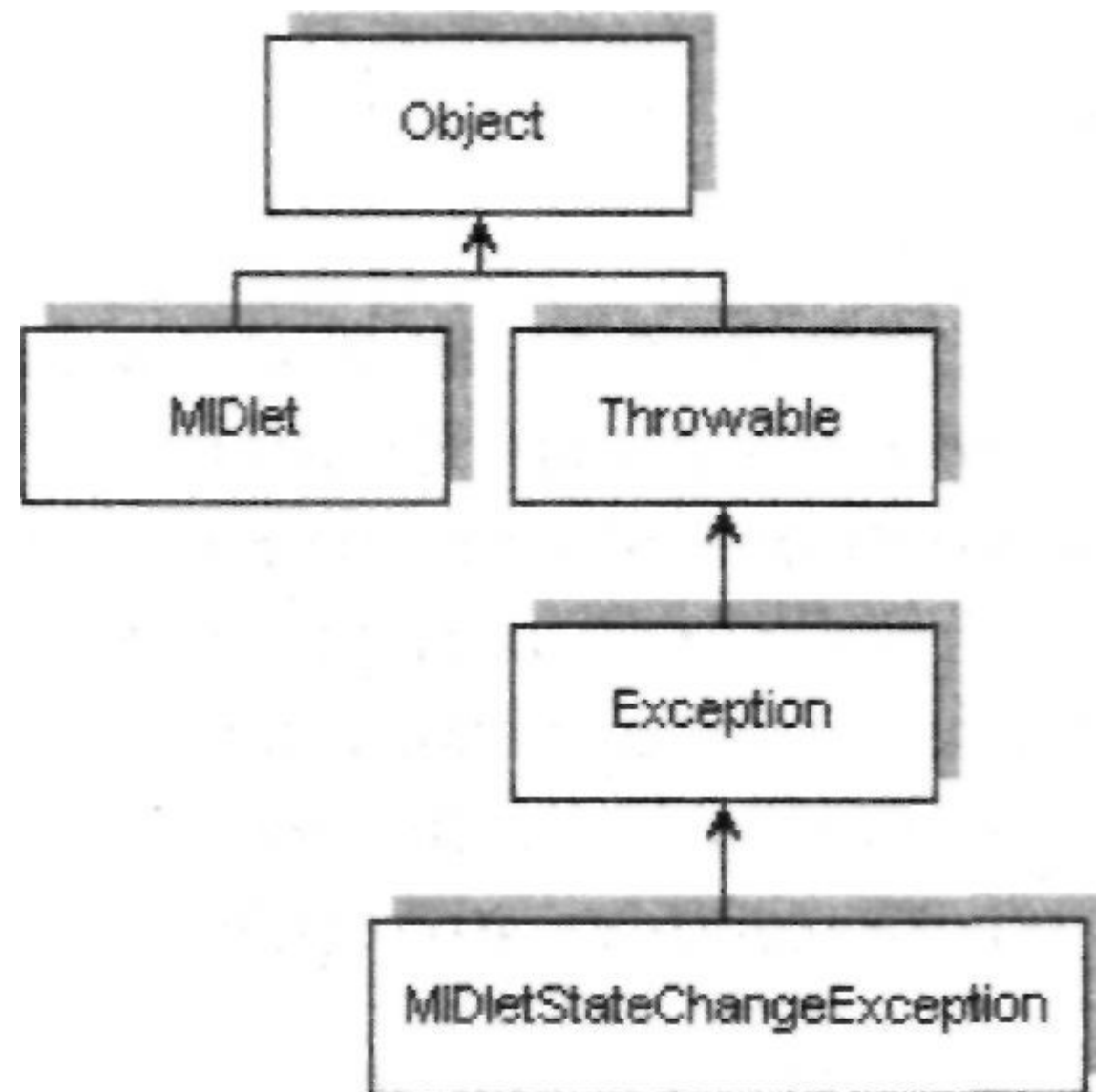


Рис. 2.12. Иерархия пакета *javax.microedition.midlet*

Класс

- MIDlet - основной класс программы должен наследовать класс MIDlet для управления работой приложения.

Исключение

- MIDletStateChangeException - исключает неправильную работу с ----COMMIDlet.

2.4.10. Пакет *javax.microedition.pki*

Пакет *javax.microedition.pki* сертифицирует информацию для безопасной связи. Рисунок 2.13 содержит иерархию этого пакета.

Интерфейс

- Certificate - общий сертификат.

Исключение

- CertificateExceptio - обобщенный вид ошибок, возникший при использовании данного сертификата.

2.4.11. Пакет *javax.microedition.rms*

Этот пакет предназначен для создания механизма хранения и извлечения данных из памяти устройства. Хранение и запись

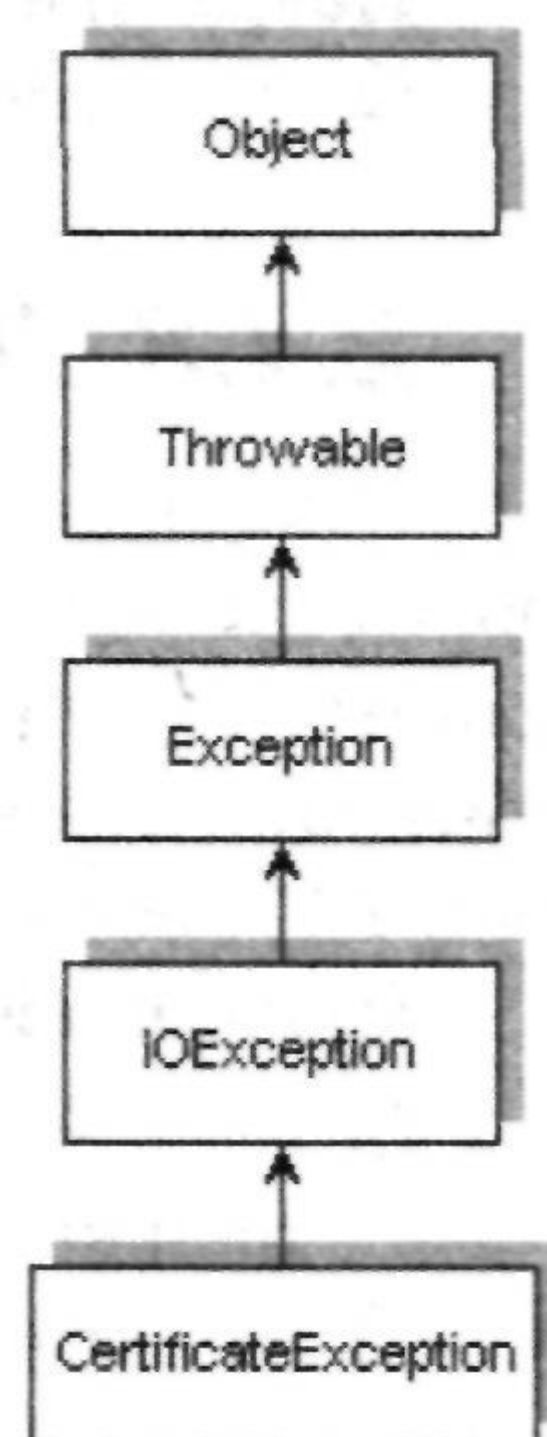


РИС. 2.13. Иерархия пакета *javax.microedition.pki*

данных происходят на основе менеджера системной записи (Record Management System), что дает возможность удалять, добавлять, просматривать, изменять или составлять список всех имеющихся записей. Имеются один класс и несколько интерфейсов, реализующих механизм сохранения и извлечения данных. На рис. 2.14 представлена иерархия пакета *javax.microedition.rms*.

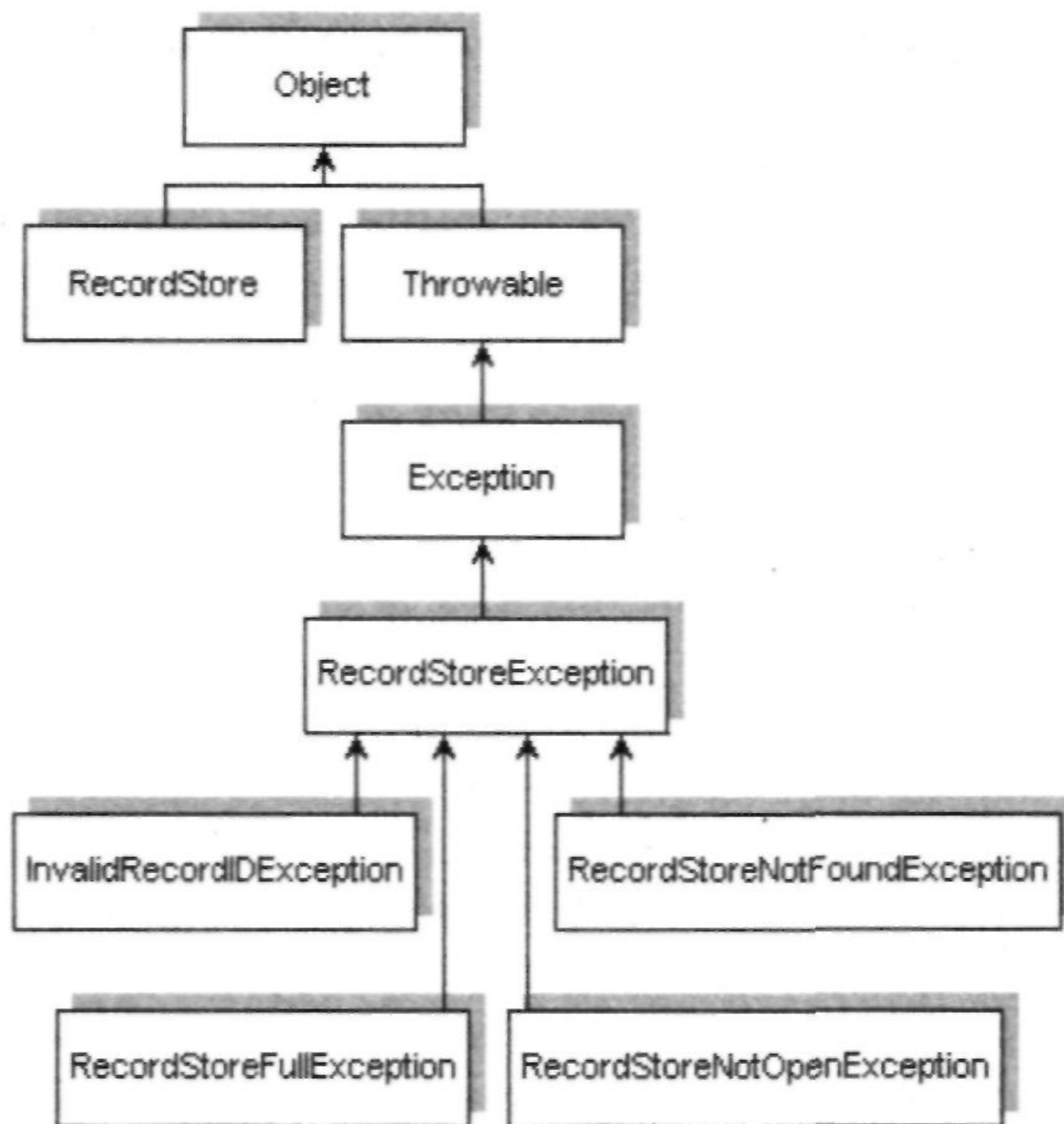


Рис. 2.14. Иерархия пакета *javax.microedition.rms*

Интерфейсы

- `RecordComparator` - осуществляет сравнение двух записей;
- `RecordEnumeration` - реализует двунаправленный список записи;
- `RecordFilter` - определяет различные совпадения в записях;
- `RecordListener` - отслеживает события записи данных.

Класс

- `RecordStore` - производит запись данных.

Исключения

- `InvalidRecordIDException` - исключает запись данных в неправильно указанный адрес;
- `RecordStoreException` - индикатор ошибки записи данных;
- `RecordStoreFullException` - указывает на переполнение системных ресурсов для записи данных;
- `RecordStoreNotFoundException` - показывает, что указанное место для записи данных не было обнаружено;
- `RecordStoreNotOpenException` - указывает на невозможность записи.

В этой главе мы рассмотрели состав 11 пакетов, давая краткую характеристику имеющимся интерфейсам, классам и исключениям. В конце книги в *приложении 2* вы найдете справочник по Java 2 ME, где рассматриваются более подробно все составляющие CDLC/MIDP. В следующей главе будут изучены интегрированные средства разработки приложений, бесплатно предоставляемые компанией Sun Microsystems.

<http://palata-x.narod.ru>



<http://palata-x.narod.ru>

Глава 3. Инструментальные средства разработки мобильных приложений

Для создания программ под мобильные устройства на компьютере используются так называемые интегрированные среды программирования IDE (Integrated Development Environment), или инструментальные средства программирования (инструментарии). Эти средства представляют собой целый набор программных компонентов, которые включают в себя текстовый редактор, компилятор, отладчик, эмулятор телефона и другие различные программные утилиты. В дальнейшем в книге все эти средства мы будем называть просто инструментальными средствами, или инструментариями, для простоты восприятия материала.

Кроме инструментариев, еще существуют так называемые программные пакеты разработчика SDK (Software Developer Kit), которые создаются производителями телефонов. Такие пакеты разработчика представляют один или более эмуляторов реальных телефонов определенной модели, с помощью которых вы можете тестировать создаваемые программы прямо на компьютере. Почти все пакеты разработчиков от разных производителей обладают возможностью интегрироваться в IDE. В следующей главе мы более детально изучим принципы работы с SDK от различных производителей телефонов, сейчас главное, чтобы вы понимали разницу между IDE и SDK.

В данный момент на рынке представлено много разных IDE для создания мобильных программ. Большинство инструментариев распространяются на платной основе, но что самое интересное, все самые значимые и перспективные средства поставляются как раз на бесплатной основе. В основном это относится к инструментам от компании SUN и дополнительным SDK от производителей телефонов. В этой главе мы рассмотрим два широко распространенных и, главное, бесплатных инструментария J2ME Wireless Toolkit и NetBeans IDE + Mobility Pack от компании Sun Microsystems. Оба инструментария вы найдете на компакт-диске к книге в папке \ШЕ.

3.1. Установка Java 2 SDK SE

Перед тем как устанавливать на свой компьютер любые инструменты, связанные с созданием приложений на языке Java, вам обязательно необходимо установить так называемый Java SDK. Этот комплект разработчика содержит в себе ряд полезных и необходимых для работы с Java-компонентами утилит, и прежде всего виртуальную Java-машину. Без Java SDK вам не удастся создать ни одной программы для мобильного устройства, поскольку вы даже не сможете установить некоторые средства программирования на свой компьютер.

На компакт-диске в папке \IDE находится файл j2sdk-1_4_2_05-windows-i586-p, который и представляет установочный пакет Java SDK. В данный момент на сайте компании Sun Microsystems присутствует и более поздний пакет SDK под цифровым обозначением Java SDK v1.5beta. Но, к сожалению, пока еще не все инструменты поддерживают эту версию SDK, поэтому мы остановимся на Java SDK версии 1.4, в этом случае нам удастся избежать массы проблем в создании программ для телефонов. Естественно, со временем новая версия SDK будет поддержана всеми производителями телефонов.

Дальнейшие этапы инсталляции Java SDK на компьютер рассматриваются в виде пошаговой инструкции.

1. Начнем установку пакета Java SDK на ваш компьютер и щелкнем по названию файла j2sdk-1_4_2_05-windows-i586-p два раза левой кнопкой мыши. После запуска процесса инсталляции Java SDK на экране монитора появится диалоговое окно с лицензионным соглашением **Java SDK, SE v1.4.2_05 - License**, изображенное на рис. 3.1. Это стандартное лицензионное соглашение, где вам необходимо избрать флажок **I accept the terms in the license agreement** и нажать кнопку **Next**. В противном случае установка Java SDK на ваш компьютер будет невозможна.

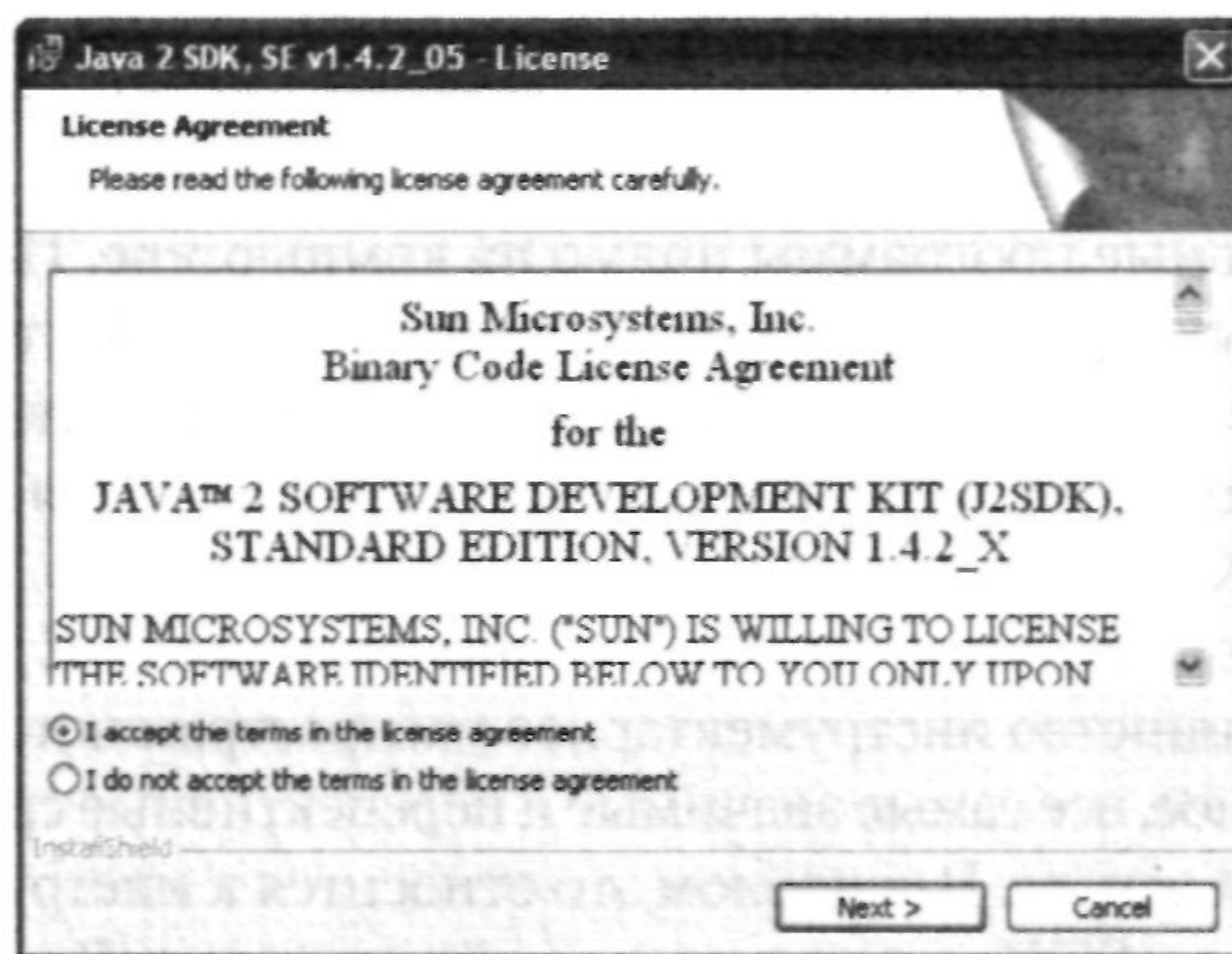


Рис. 3.1. Диалоговое окно Java SDK, SE v1.4.2_05 - License

2. Следующее открывшееся диалоговое окно будет носить название **Java SDK, SE v1.4.2_05 - Custom Setup** (рис. 3.2). В этом окне вам будет представлен полный список устанавливаемых на компьютер компонентов Java SDK. Выделяя из списка один из компонентов, в правой части этого диалогового окна в списке **Feature Description** вы найдете описания каждого из устанавливаемых на компьютер компонентов. Нам понадобятся все компоненты. Дополнительно в этом окне можно изменить директорию и папку, в которую будет устанавливаться Java SDK. Так вот делать этого ни в коем случае нельзя!!! Для установки Java SDK должно выполняться обязательное условие - это корневой каталог диска «С» и папка по умолчанию, предлагаемая инсталлятором. Вы даже не представляете,

какое количество писем я получил по предыдущему изданию этой книги, где меня как только ни обзывали и ругали за то, что якобы ни один код на диске не работает или при компиляции возникают ошибки и т. д.! И все это только потому, что люди ставили как Java SDK, так и другие инструменты (о которых я еще обязательно вспомню) не так, как описывалось в книге! Куда только начинающие программисты ни устанавливали Java SDK, IDE и SDK! Начинаю переписываться с читателем, и выясняется, что IDE стоит на одном виртуальном диске, эмулятор - на другом, а Java SDK не понятно где находится, и человек даже не помнит, куда он его «засунул»! При этом в книге я четко акцентировал свое внимание на том, куда и как нужно устанавливать все необходимые для работы средства (теперь буду еще и выделять жирным шрифтом). Я все понимаю, все мы когда-то начинали изучать программирование, но, пожалуйста, будьте внимательны: если я вам говорю, что это нужно ставить исключительно в корневой каталог диска «С», то это туда и нужно ставить. Это не я придумал, это такие правила, которые необходимо соблюдать, иначе у вас будет масса проблем, а я буду плохим дядей, который все плохо объяснил.

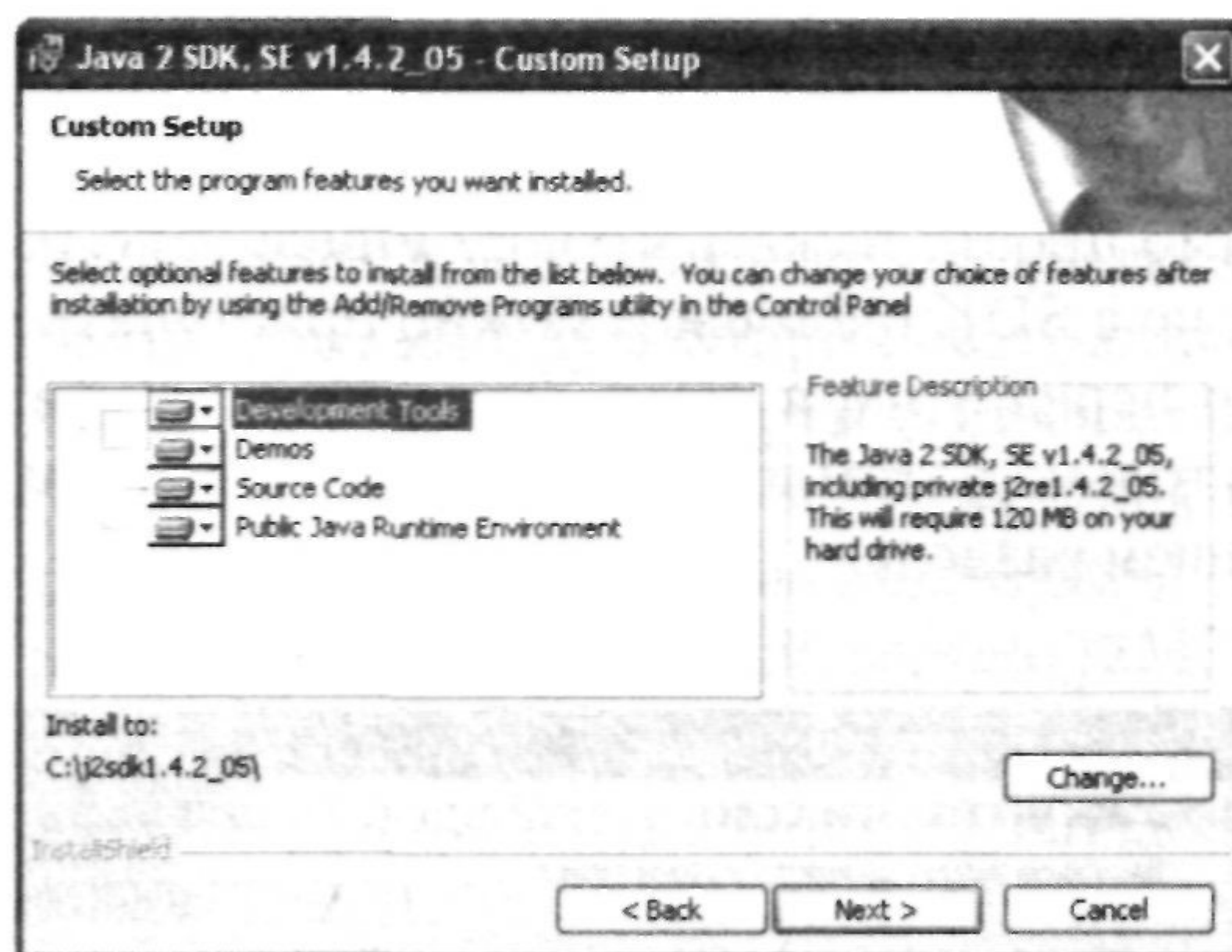


Рис. 3.2. Диалоговое окно Java SDK, SE v1.4.2_05 - Custom Setup

3. В третьем диалоговом окне **Java SDK, SE v1.4.2_05 - Browser Registration**, изображенном на рис. 3.3, необходимо выбрать флажок **Internet Explorer**, дабы виртуальная Java-машина интегрировала в браузер свой модуль. Кстати, если у вас на компьютере ранее уже была установлена виртуальная Java-машина (поздней или ранней версии от той, что устанавливаем мы), то ничего страшного нет. Дело в том, что Java SDK устанавливается специальным образом, и все компоненты будут работать непосредственно с IDE и SDK. Но не выбирать виртуальную Java-машину в предыдущем диалоговом окне нельзя, уж лучше перед установкой или после установки SDK удалить со своего компьютера вашу версию виртуальной Java-машины. Для продолжения установки нажмите кнопку **Install**.

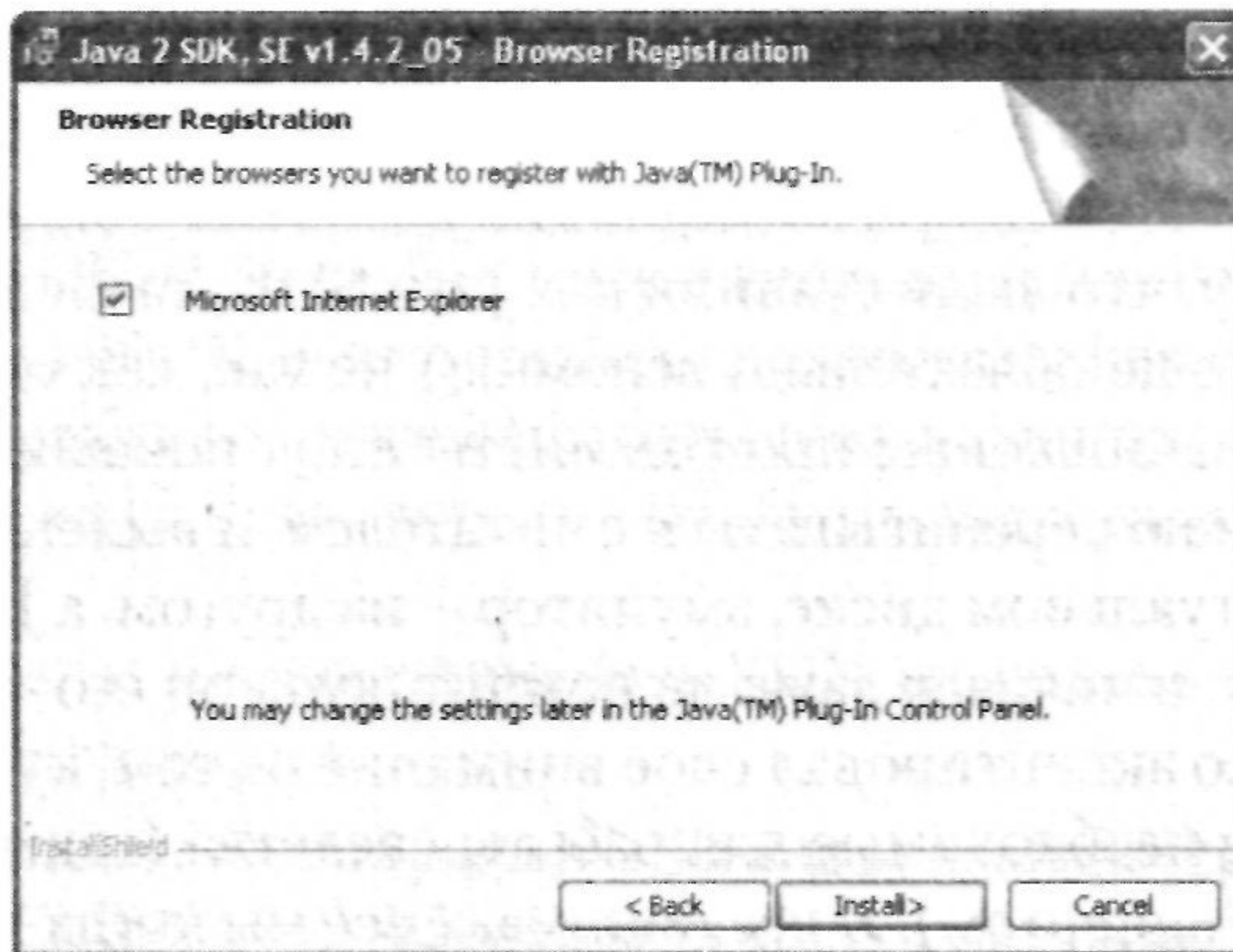


Рис. 3.3. Диалоговое окно Java SDK, SE v1.4.2_05 - Browser Registration

4. После всех настроек опций по установке Java SDK на экране монитора появится предпоследнее диалоговое окно **Java SDK, SE v1.4.2_05 - Progress**, в котором вы сможете наблюдать процесс инсталляции всех компонентов Java SDK на ваш компьютер (рис. 3.4). На весь процесс инсталляции потребуется несколько минут. По окончании процесса установки Java SDK откроется последнее диалоговое окно **Java SDK, SE v1.4.2_05 - Complete**, где вам необходимо просто нажать кнопку **Finish** (рис. 3.5). На этом процесс инсталляции Java SDK окончен и можно приступать к установке инструментариев, но перед этим я вам рекомендую все же перезагрузить свой компьютер и только потом приступать к дальнейшим действиям, описанным в следующем разделе.

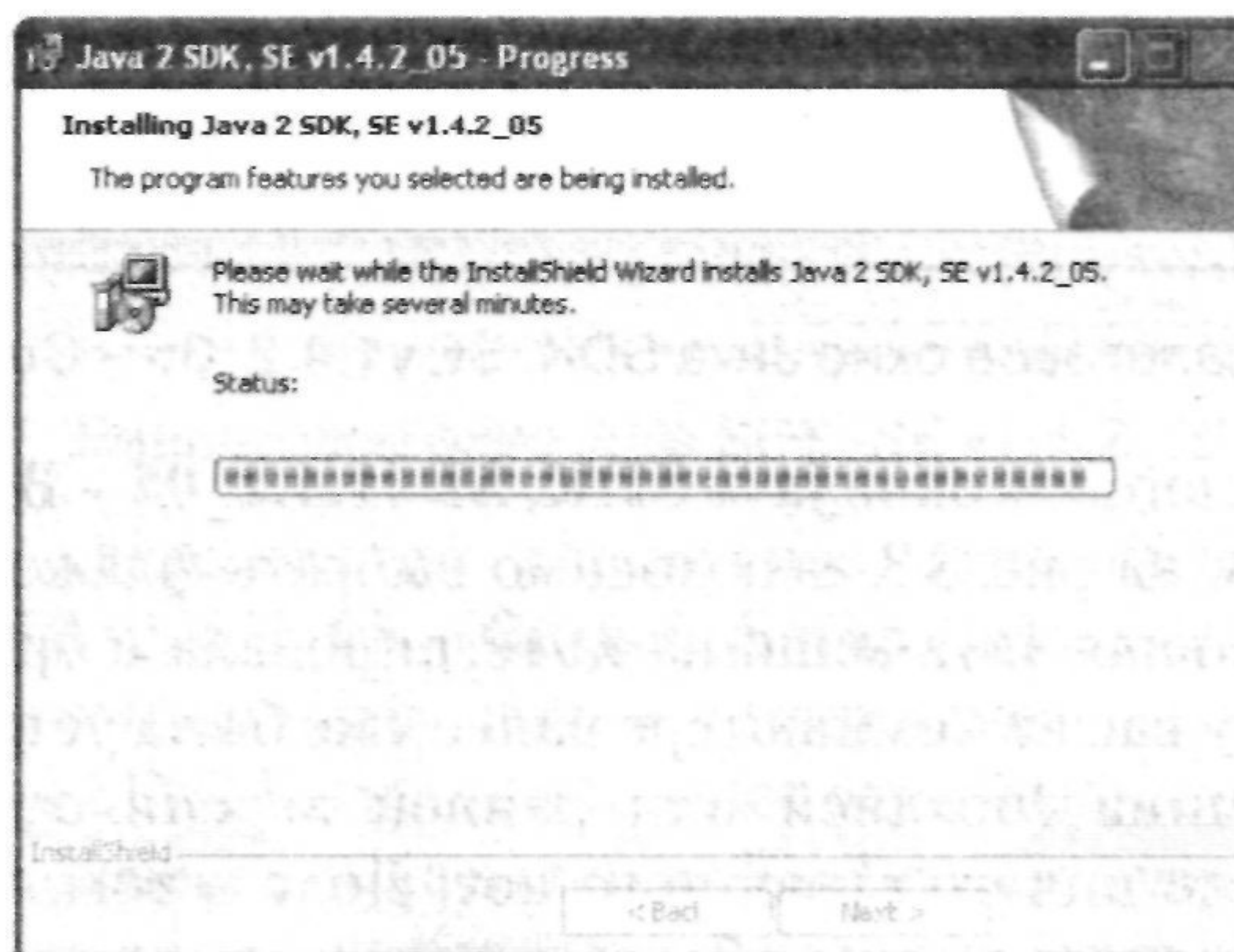


Рис. 3.4. Диалоговое окно Java SDK, SE v1.4.2_05 - Progress



Рис. 3.5. Диалоговое окно Java SDK, SE v1.4.2_05 - Complete

3.2. Инструментарий J2ME Wireless Toolkit

Инструментарий J2ME Wireless Toolkit компании Sun Microsystems распространяется на бесплатной основе и находится на компакт-диске в папке \IDE. Этот инструментарий представляет собой такую хитрую модель инструментария, в котором, к сожалению, отсутствует текстовый редактор. То есть исходный код программы придется писать при помощи других средств. Подобный подход объясняется очень просто. Этот инструментарий был одним из первых и создавался на заре восхождения Java 2 ME, поэтому (может, еще по каким-то неизвестным причинам) текстовый редактор у J2ME Wireless Toolkit отсутствует.

На сегодняшний день на сайте компании Sun Microsystems (<http://java.sun.com>) можно скачать две версии J2ME Wireless Toolkit: версию J2ME Wireless Toolkit 2.3, которую мы и будем изучать, и версию J2ME Wireless Toolkit 2.5. Более поздняя версия отличается только тем, что имеет поддержку виртуальной Java-машины (и Java SDK) в версии 1.5. Как мы уже выяснили, пока не все SDK различных производителей могут работать с этой версией виртуальной машины, поэтому в книге мы используем J2ME Wireless Toolkit 2.3 в связке с Java SDK 1.4. Если вы все же желаете работать с J2ME Wireless Toolkit 2.5, то вам понадобится Java SDK 1.5, который можно скачать с сайта компании Sun Microsystems (<http://java.sun.com>).

Перейдем к установке инструментария J2ME Wireless Toolkit на ваш компьютер. Здесь, как и в случае с инсталляцией Java SDK, есть свои нюансы, которые необходимо беспрекословно соблюсти, иначе затем в работе вас будут поджидать различные непонятные ошибки, возникающие при компиляции проектов. Сама установка J2ME Wireless Toolkit состоит из череды последовательных диалоговых окон. Весь процесс установки инструментария в диалоговых окнах мы рассматривать не будем, а лишь сконцентрируемся на ключевых моментах.

3.2.1. Установка J2ME Wireless Toolkit

1. На компакт-диске в папке \IDE найдите файл j2me_wireless_toolkit-2_3. Двойной щелчок на этом файле запустит программу установки J2ME Wireless Toolkit. В первых двух диалоговых окнах инсталлятора J2ME Wireless Toolkit вас поприветствуют с началом установки программы и предложат прочитать лицензионное соглашение. Затем откроется новое диалоговое окно **J2ME Wireless Toolkit - Java Virtual Machine Location**. В момент открытия этого окна программа установки протестирует вашу компьютерную систему и отыщет место установки виртуальной Java-машины. Как вы помните, мы договорились, что устанавливаем Java SDK в корневой каталог, поэтому на рис. 3.6 инсталлятор обозначил для себя именно этот путь. Для продолжения нажмите кнопку **Next**.

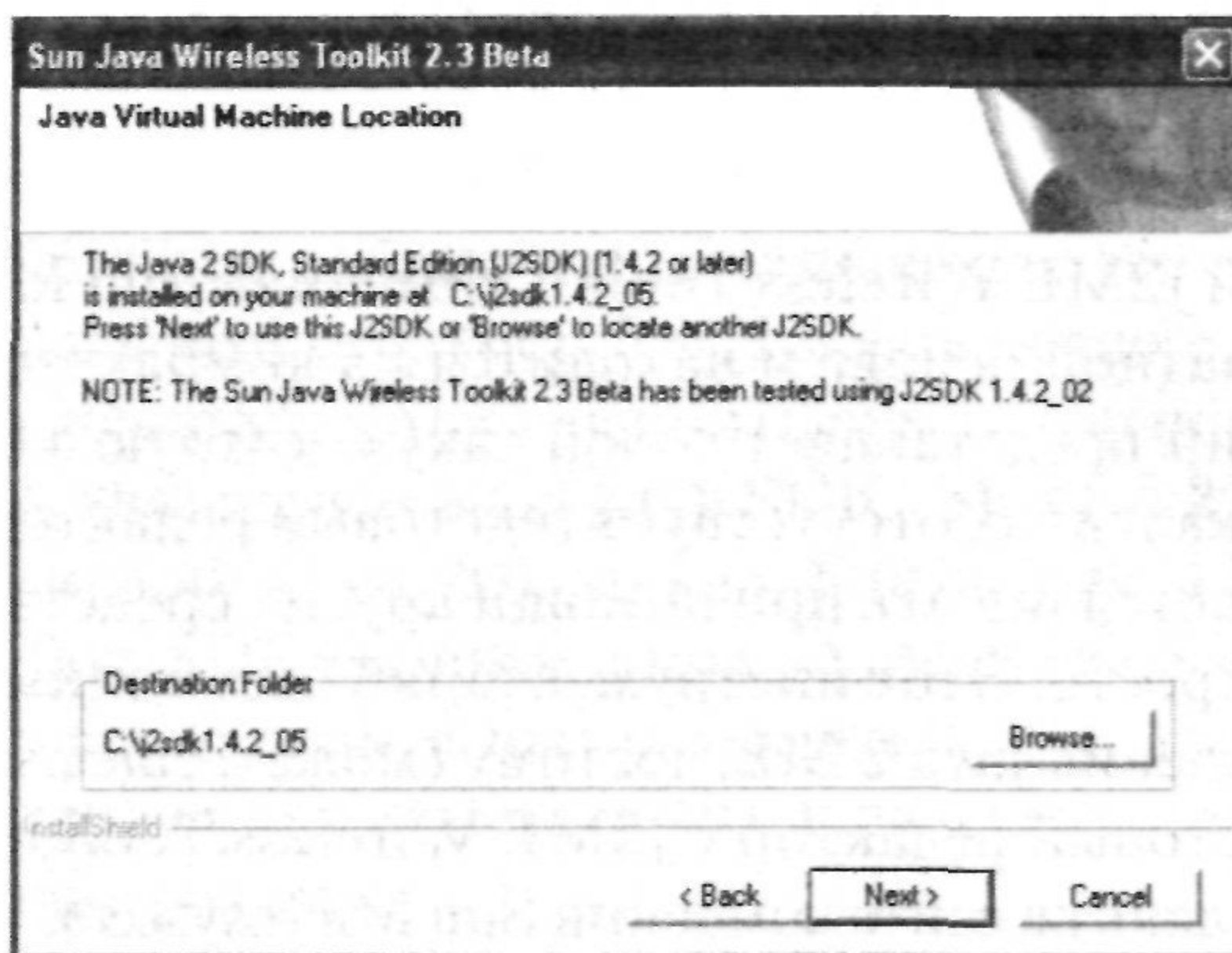


Рис. 3.6. Окно J2ME Wireless Toolkit - Java Virtual Machine Location

2. В следующем диалоговом окне **J2ME Wireless Toolkit - Choose Destination Location** для установки J2ME Wireless Toolkit предлагается корневой каталог диска «С» (рис. 3.7). **Никаких изменений директории в этом диалоговом окне вы делать не должны!** Инсталлируем J2ME Wireless Toolkit в предложенный по умолчанию каталог и не изменяем название папки. Если все-таки название папки вам не нравится, то новое имя **не должно содержать пробелов и русских символов**, но, как мне кажется, значительно проще оставить все как есть. Для продолжения нажмите кнопку **Next**.
3. В итоге правильная директория для установки J2ME Wireless Toolkit будет выглядеть, как показано на рис. 3.8. По появлении этого окна нажмите кнопку **Next**, запустив тем самым установку программы на ваш компьютер. По окончании инсталляции в диалоговом окне **J2ME Wireless Toolkit - Install Shield Wizard Complete** нажмите кнопку **Finish** (рис. 3.9). На этом установка инструментария окончена, перезагрузки компьютера не требуется, но можно сделать на всякий случай.

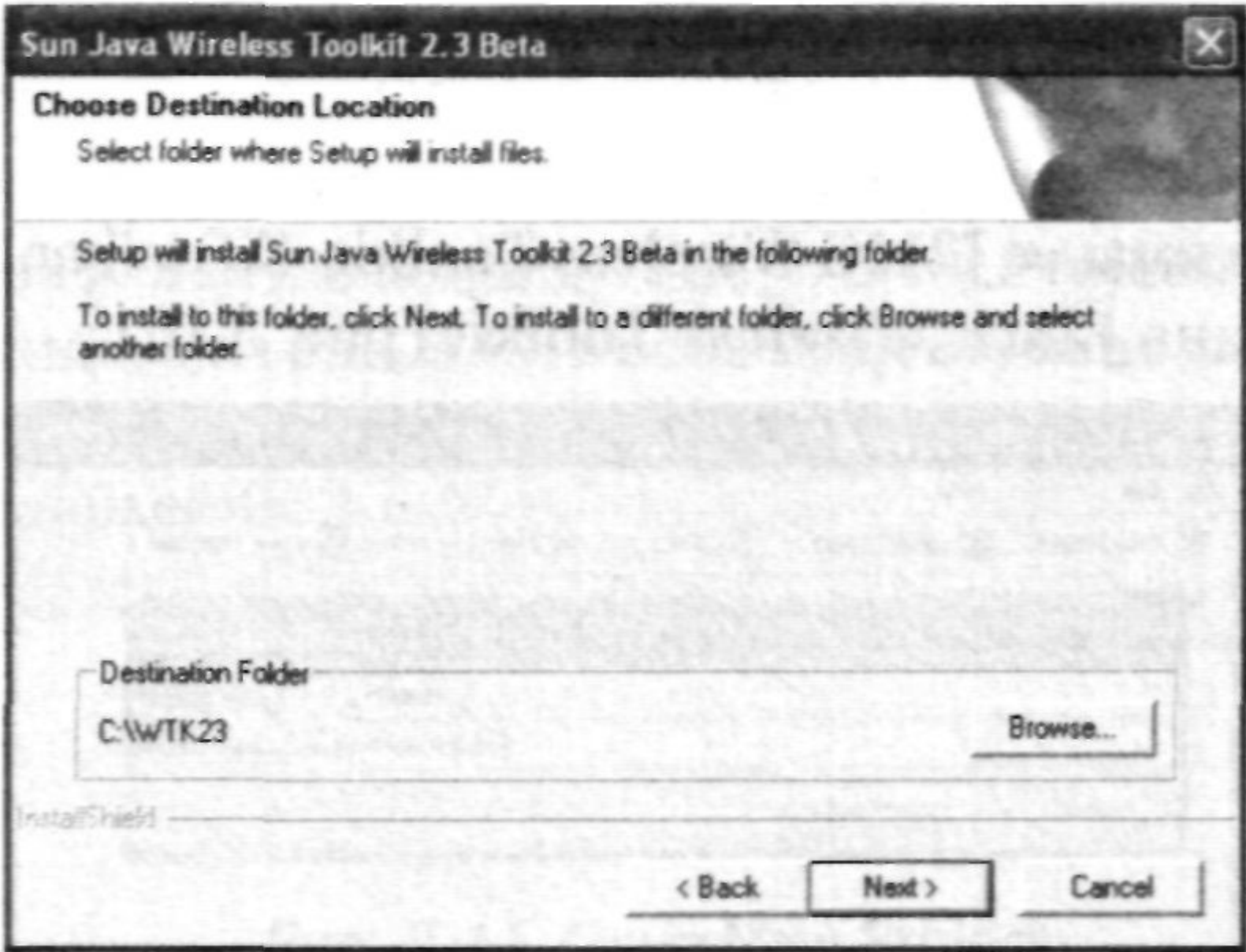


Рис. 3.7. Диалоговое окно J2ME Wireless Toolkit - Choose Destination Location

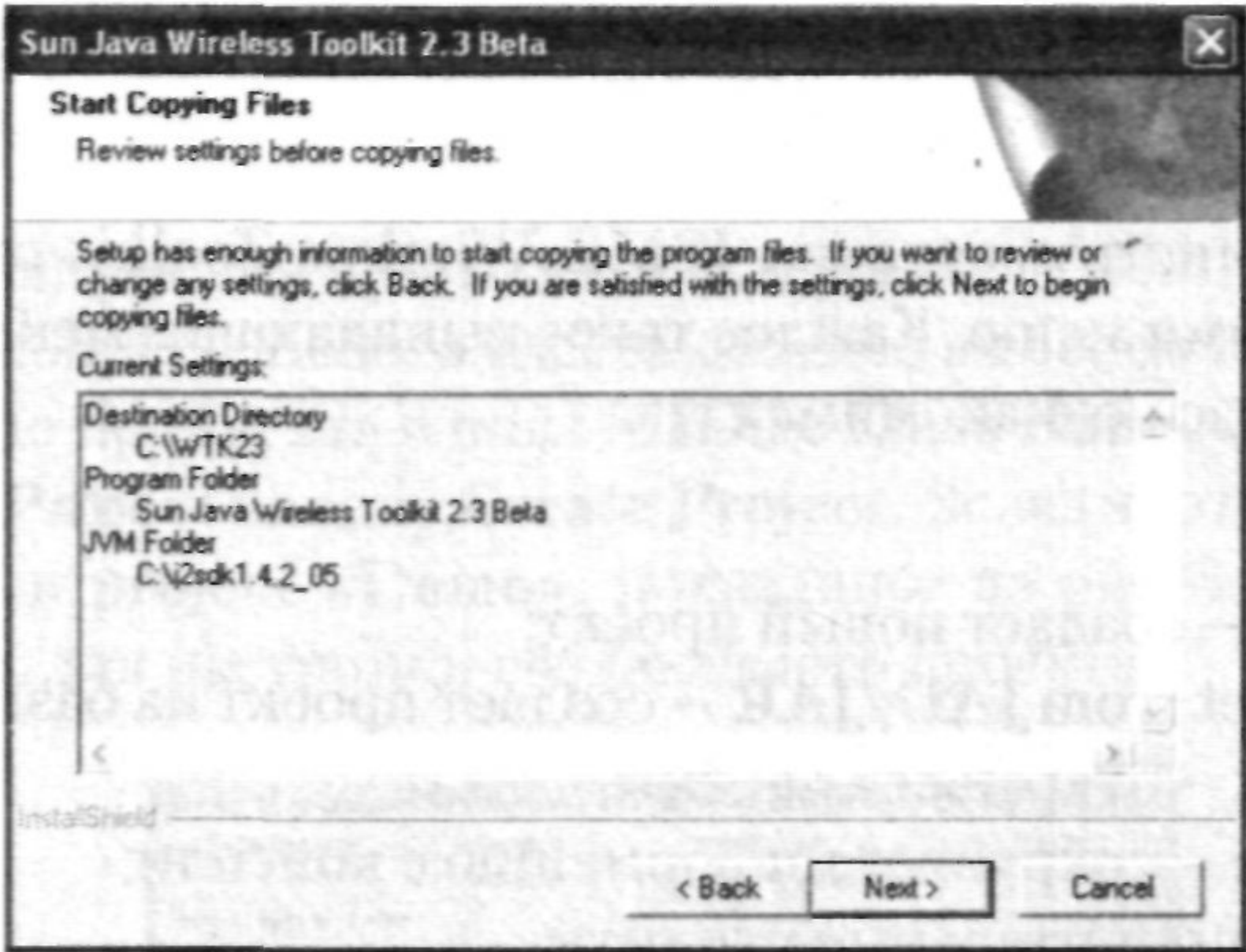


Рис. 3.8. Диалоговое окно J2ME Wireless Toolkit - Start Copying Files

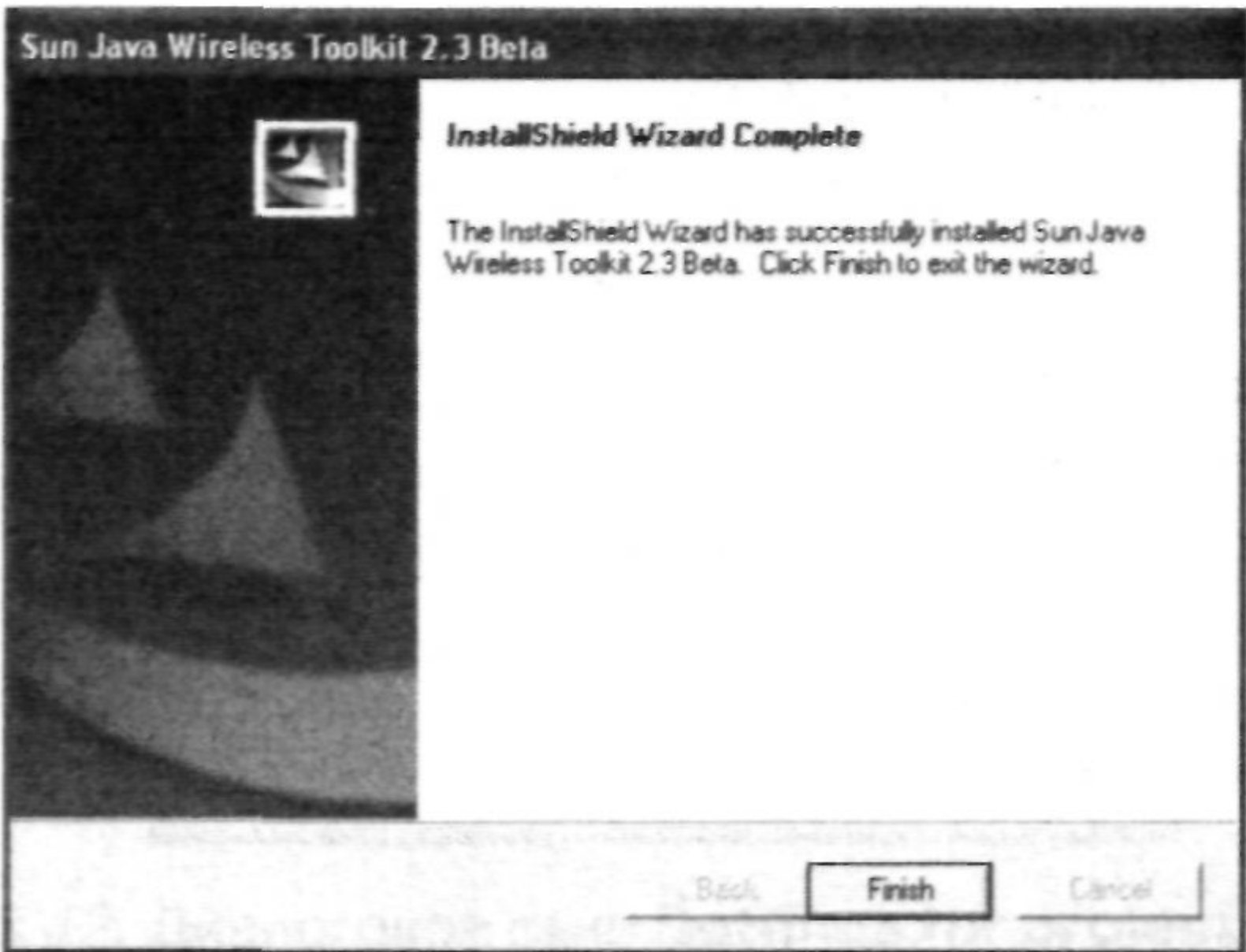


Рис. 3.9. Диалоговое окно J2ME Wireless Toolkit - Install Shield Wizard Complete

3.2.2. Знакомимся с J2ME Wireless Toolkit

После установки J2ME Wireless Toolkit выполните на компьютере команды **ПУСК => Все программы => J2ME Wireless Toolkit KToolbar**. На экране монитора откроется рабочее окно J2ME Wireless Toolkit (рис. 3.10).



Рис. 3.10. Рабочее окно J2ME Wireless Toolkit

Линейка меню инструментария J2ME Wireless Toolkit содержит четыре команды с выпадающими меню. Каждое такое выпадающее меню имеет ряд команд с которыми мы сейчас познакомимся.

Меню File

- **New Project** - создает новый проект;
- **Create Project from JAD/JAR** - создает проект на базе JAD- и JAR-файла;
- **Open Project** - открывает проект;
- **Save Console** - сохраняет информацию с консоли;
- **Utilities** - дополнительные утилиты;
- **Exit** - ВЫХОД.

Меню Edit

- **Preferences** - свойства проекта;
- **Clear Console** - очистить консоль.

Меню Project

- **Build** - компиляция проекта;
- **Clean** - закрывает открытый проект;
- **Run** - запуск откомпилированной программы;
- **Package** - упаковка проекта;
- **Debug** - отладка;
- **Settings** - установки проекта.

Help

- **Documentations** - справочная документация в формате HTML;
- **News and Updates** - обновление инструментария через сеть Интернет;
- **About** - информация о создателе программы.

3.2.3. Создание проекта в J2ME Wireless Toolkit

Для создания нового проекта нажмите на панели инструментов кнопку **New Project**, либо выберите в меню команду **File => New Project**. В появившемся окне **New Project**, показанном на рис. 3.11, в поле **Project Name** дайте имя всему проекту, например **Demo**, и в поле **MIDlet Class Name** задайте название основному классу мидлета приложения.

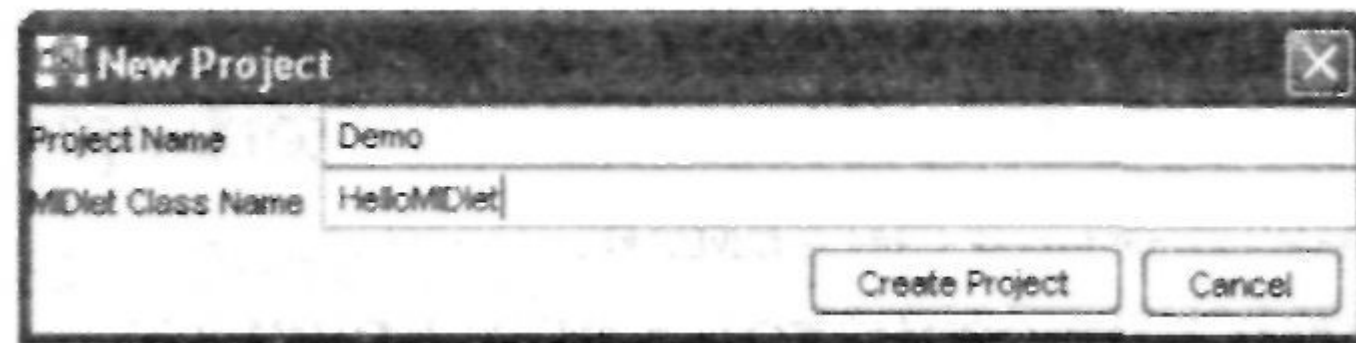


Рис. 3.11. Окно New Project

Чтобы было понятно, о чем идет речь, забегаю вперед, поясню. Мидлет в Java 2 ME - это вся программа в целом, а с основного класса мидлета начинается работа всего приложения. Поэтому название, данное в поле **MIDlet Class Name**, должно впоследствии (об этом позже) соответствовать названию основного класса мидлета программы. В *главе 5* вы найдете более подробное объяснение модели работы и построения программ в Java 2 ME.

В качестве названия основного класса мидлета выберем название **HelloMIDlet**, которое необходимо прописать в поле **MIDlet Class Name**. Затем нажмите в диалоговом окне **New Project** кнопку **Create Project**. Вслед за этим появится диалоговое окно **Settings for project «Demo»**, показанное на рис. 3.12, где нужно выполнить ряд установок для настройки создаваемого проекта.

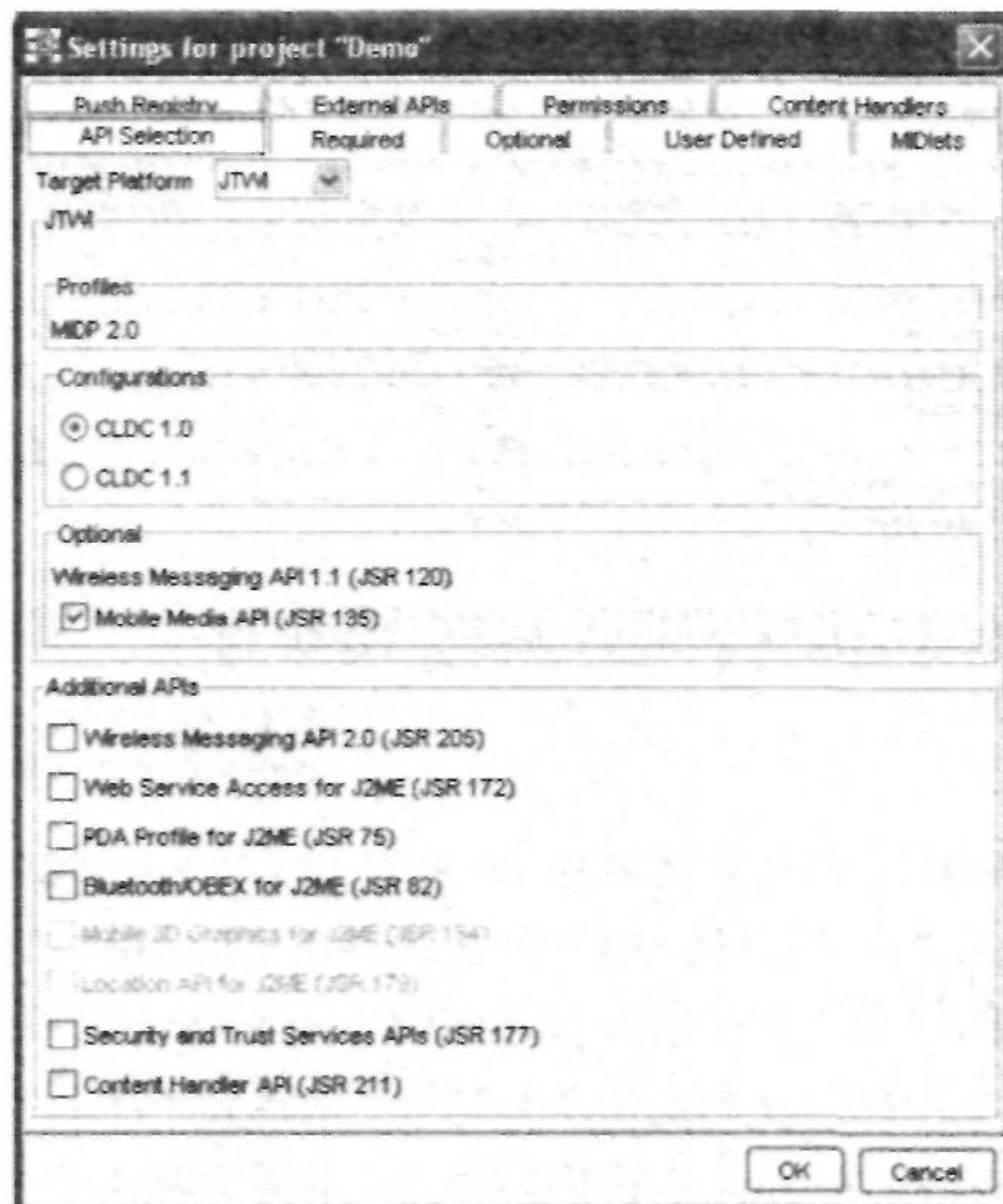


Рис. 3.12. Диалоговое окно Settings for project «Demo»

Диалоговое окно **Settings for project «Demo»** разделено на семь вкладок, в каждой из которых можно задавать различные установки для создаваемого проекта.

Первая вкладка **API Selection**, изображенная на рис. 3.12, позволяет задавать конфигурацию и профиль создаваемого проекта. И здесь необходимо быть аккуратным: если вы планируете разрабатывать приложение под профиль CLDC 1.1/MIDP 2.0, то в списке **Target Platform** (выбор платформы) нужно выбрать соответствующую конфигурацию, а в поле **Profiles** - указать необходимый профиль.

Вторая вкладка **Required** (Атрибуты), изображенная на рис. 3.13, содержит метайнформацию создаваемой программы, отраженной в виде семи различных значений, которые можно задавать. Присутствуют следующие поля:

- MIDlet - Jar - Size - размер всего создаваемого приложения, эта величина изменяется автоматически системой;
- MIDlet - Jar - URL - местонахождение проекта;
- MIDlet - Name - имя проекта;
- MIDlet - Vendor - создатель программы, и здесь вы можете прописать свое имя и фамилию (кириллица отображается некорректно);
- MIDlet - Version - версия программы;
- MicroEdition - Configuration - выбранная конфигурация;
- MicroEdition - Profile - выбранный профиль.

Впоследствии по ходу работы над проектом имеется возможность изменения всех этих атрибутов с помощью кнопки **Settings** на панели инструментов рабочего окна J2ME Wireless Toolkit. Подробно о файлах JAD и JAR будет рассказано далее в этой главе. Все оставшиеся вкладки имеют различные дополнительные свойства, и заданные на этих вкладках значения почти всегда можно оставлять по умолчанию. Нажав на кнопку ОК, в окне **Settings for project «Demo»** будет создан новый проект с названием Demo.

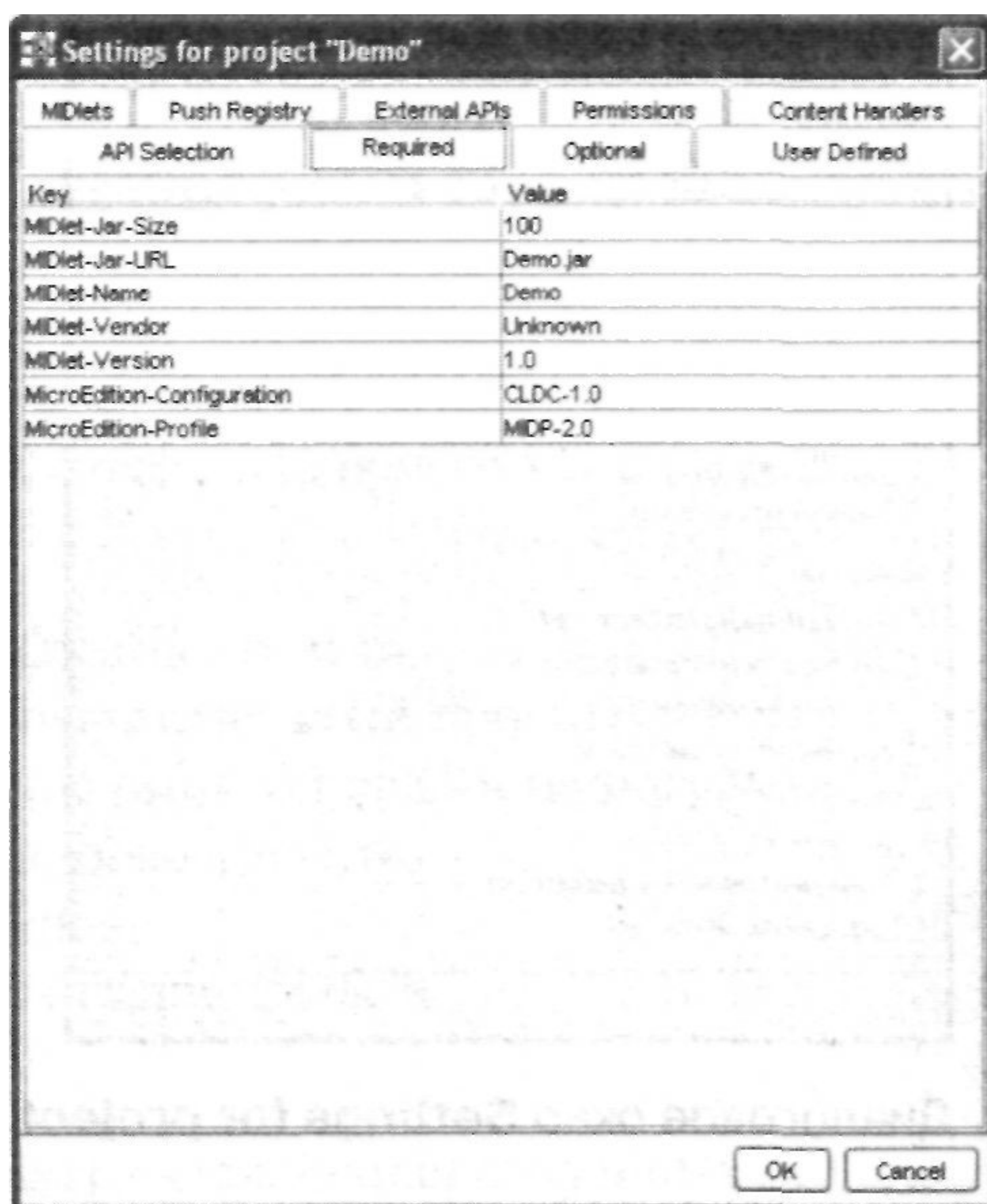


Рис. 3.13. Вкладка Required

Инструментарий J2ME Wireless Toolkit, как мы уже выяснили, к сожалению, не имеет своего текстового редактора. Для создания исходного кода приложения вам необходимо написать код программы в любом удобном текстовом редакторе и поместить этот код в рабочий каталог инструментария J2ME Wireless Toolkit. *Только из этого каталога возможно корректное создание программы, и на этом мы сейчас заострим свое внимание!*

В тот момент, когда вы нажмете на кнопку ОК, в окне **Settings for project «Demo»** будет создан новый проект, в нашем случае это проект Demo. В консоли главного окна J2ME Wireless Toolkit появятся следующие строки:

```
Creating project "Demo"  
Place Java source files in "C:\WTK23\apps\Demo\src"  
Place application resource files in "C:\WTK23\apps\Demo\res"  
Place application library files in "C:\WTK23\apps\Demo\lib"
```

Из этих строк следует, что файлы исходного кода (а это файлы с расширением *.java) необходимо поместить в директорию C:\WTK23\apps\Demo\src. Файлы ресурсов, иконки, изображения и т. д. помещаются в директорию C:\WTK21\apps\Demo\res; и файлы библиотек, если таковые используются, - в C:\WTK21\apps\Demo\lib.

Получается, что, создав исходный код приложения в текстовом редакторе или с помощью других средств, вам необходимо разместить исходные файлы проекта в директории: C:\WTK23\apps\Demo\src. Только из этой папки возможно безошибочное создание программы (компиляция, сборка и упаковка).

Кроме трех перечисленных папок, созданных в рабочем каталоге проекта Demo, будут автоматически сгенерированы еще четыре следующие папки:

- **bin** - содержит файлы jar, jad и файл манифеста. Более подробно об этих файлах рассказывается в *разделе 3.2.5*, где будет рассматриваться упаковка программы
- **classes** - хранит проверенные откомпилированные классы;
- **tmpclasses** - хранит непроверенные откомпилированные классы;
- **tmplib** - временная папка для хранения фалов.

3.2.4. КОМПИЛЯЦИЯ И ЗАПУСК ПРОГРАММЫ в J2ME Wireless Toolkit

В качестве демонстрационного примера используется простейший исходный код, который был создан специально по этому случаю. В пятой главе мы уделим этому проекту больше времени, а сейчас изучаем только работу с J2ME Wireless Toolkit. Найдите на компакт-диске в папке **Code\Demo\src** исходный файл HelloMIDlet.java. Скопируйте этот файл на ваш компьютер в директорию C:\WTK23\apps\Demo\src.

Далее в окне инструментария J2ME Wireless Toolkit на панели инструментов нажмите кнопку **Build**, для того чтобы выполнить компиляцию и компоновку всего проекта Demo. В рабочем окне J2ME Wireless Toolkit в реальном режиме появятся две строчки текста:

```
Building "Demo"  
Build complete
```

В этих строках говорится, что проект успешно откомпилирован и готов к работе. Здесь я, конечно, исключаю возможность появления ошибок при компиляции, но в программировании приложений без этого не обойтись, программист где-нибудь да забудет поставить точку с запятой.

После компиляции и компоновки проекта вы можете запустить проект Demo и посмотреть, как он работает на базе нескольких эмуляторов, поставляемых в составе J2ME Wireless Toolkit. Для этого в рабочем окне J2ME Wireless Toolkit нажмите кнопку **Run**. На экране монитора появится эмулятор DefaultColorPhone, установленный в J2ME Wireless Toolkit по умолчанию (рис. 3.14).



Рис. 3.14. Эмулятор телефона DefaultColorPhone

Первоначально на экран эмулятора будет выведено имя запускаемой программы, нажав на клавишу эмулятора **Select** или **Launch**, вы попадете в рабочий цикл программы (рис. 3.14). С помощью клавиш эмулятора можно редактировать и набирать текст, а если нажать на клавишу **Exit**, то вы выйдете из программы. После закрытия эмулятора в консоли рабочего окна J2ME Wireless Toolkit появится следующий текст:

```
Project "Demo" loaded  
Settings updated
```

```
Project settings saved
Project settings saved
Building "Demo"
Build complete
Running with storage root DefaultColorPhone
Running with locale: Russian_Russia.1251
Execution completed.
923695 bytecodes executed
28 thread switches
893 classes in the system (including system classes)
4735 dynamic objects allocated (135848 bytes)
2 garbage collections (109396 bytes collected)
```

Это строки информационного характера, которые знакомят вас с произошедшими процессами в момент работы приложения. Для того чтобы протестировать созданную программу на других телефонных эмуляторах, необходимо в поле **Device** инструментария J2ME Wireless Toolkit выбрать из списка другой эмулятор. В составе J2ME Wireless Toolkit имеются следующие эмуляторы телефонов:

- **DefaultColorPhone** - простой телефон с цветным дисплеем;
- **DefaultGrayPhone** - телефон с монохромным дисплеем;
- **MediaControlSkin** - простейший эмулятор телефона, контролирующий воспроизведение звуков;
- **QwertyDevice** - портативное устройство с клавиатурой.

3.2.5. Упаковка программ

При создании проекта и последующей компиляции исходного кода проекта Демо в папке **\bin** рабочего каталога J2ME Wireless Toolkit у вас образуются два файла. Файл JAD и файл манифеста, или Demo.jad и MANIFEST.MF. В данный момент эти два файла представляют готовую программу, которая может работать пока только на компьютере. Чтобы перенести программу на телефон, ее необходимо обязательно упаковать, или создать JAR-файл. Об этом чуть позже, а пока разберемся, что представляют собой файлы Demo.jad и MANIFEST.MF.

Файл манифеста

Файл манифеста MANIFEST.MF описывает возможные атрибуты создаваемого приложения. Откройте файл манифеста программы HelloMIDlet с помощью любого текстового редактора, например блокнота, и вы увидите следующие строки:

```
MIDlet-1: Demo, Demo.png, HelloMIDlet
MIDlet-Name: Demo
MIDlet-Vendor: Gornakov Stanislav - www.gornakov.ru
MIDlet-Version: 0.1
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
```


Эти строки описывают атрибуты приложения. При создании проекта с J2ME Wireless Toolkit упоминалось о диалоговом окне **Settings for project «Demo»**, изображенном на рис. 3.13. Это окно разделено на семь вкладок, в каждой из которых указываются различные атрибуты создаваемого приложения. На основе заданных атрибутов в диалоговом окне **Settings for project «Demo»** и происходит генерация файла манифеста и JAD-файла.

Файл JAD

Как уже упоминалось, в рабочем каталоге проекта «Demo» будет находиться еще один файл Demo.jad. Файл JAD в мобильных приложениях также называют дескриптором приложения (Java Application Descriptor). Этот файл используется телефоном во время работы программы для получения информации об имеющихся классах, изображениях, пиктограммах и звуковых файлах всей программы. На основе полученной информации происходит управление внутренними ресурсами приложения.

Если вы переместитесь в рабочий каталог проекта «Demo» и найдете сгенерированный там файл Demo.jad, то увидите иконку в виде телефона с левой стороны от названия файла. Сделав двойной щелчок левой кнопкой мыши на файле Demo.jad, вы запустите эмулятор телефона вне зависимости от того, открыта ли в данный момент одна из сред программирования или нет. Это еще раз указывает на то, что JAD-файл используется для управления работой программы. Но есть и еще одна интересная особенность JAD-файла. Откройте файл Demo.jad с помощью любого текстового редактора, и вы увидите следующие строки:

```
MIDlet-1: Demo, Demo.png, HelloMIDlet
MIDlet-Jar-Size: 30
MIDlet-Jar-URL: Demo.jar
MIDlet-Name: Demo
MIDlet-Vendor: Gornakov Stanislav - www.gornakov.ru
MIDlet-Version: 0.1
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
```

Файл JAD тоже содержит описание атрибутов приложения, и они во многом дублируются в файле манифеста MANIFEST.MF. В программе Java 2 ME JAD-файл - это дескриптор приложения, который используется для управления работой программы. Сервис телефона перед запуском программы обращается именно к JAD-файлу, определяя тем самым возможность работы всей программы на этом телефоне.

Если один из параметров будет недопустим для этой модели телефона, то приложение не будет запущено. Например, телефон не поддерживает ту или иную версию конфигурации и профиля, или размер JAR-файла больше допустимой памяти в телефоне, выделенной для Java-программ, и т. д.

Файл JAR

В языке Java существует возможность архивации файлов приложения в один файл с расширением *.jar. *Файл JAR*- это архив, содержащий сопутствующие классы, графические изображения, звуковые файлы и другие ресурсы программы.

Файл JAR основан на обыкновенном zip-формате, используемом повсеместно. При написании программ на Java под различные компьютерные операционные системы программист волен сам выбирать, будет ли он распространять свое приложение в заархивированном виде или в оригинальном.

Ситуация с распространением-переносом программ под мобильные телефоны радикально противоположная. Телефоны ограничены в своих ресурсах, и в связи с этим все мобильные программы обязаны распространяться в заархивированном виде, то есть в JAR-файле. Написав программу для телефона, необходимо ее упаковать в JAR-архив. Делается это достаточно просто, и сейчас мы рассмотрим данный процесс.

Выберете в меню инструментария J2ME Wireless Toolkit команду **Project => Package => Create Package**. В этот момент на несколько секунд в консоли J2ME Wireless Toolkit появится небольшое диалоговое окно, показывающее процесс упаковки программы в JAR-файл. После чего в каталоге проекта Demo в папке \bin появится файл Demo.jar. Перейдите для интереса в каталог проекта Demo и найдите сформированный архив Demo.jar. Затем воспользуйтесь любым архиватором, поддерживающим zip-формат, например WinRar или WinZip, и откройте файл Demo.jar. Вы увидите файл HelloMIDlet.class и папку META-INF, открыв которую вы обнаружите файл манифеста MANIFEST.MF.

В момент упаковки приложения в JAR-файл происходят копирование всех имеющихся откомпилированных и проверенных классов, графических изображений, звуковых файлов (если таковые имеются) и размещение их в JAR-файл. То есть происходит архивация всей программы в единый файл. В рассматриваемом примере HelloMIDlet существует всего один класс, но если программа имеет большое количество классов, то все они помещаются в JAR-файл. Также происходит копирование файла манифеста MANIFEST.MF в папку META-INF.

После упаковки проекта в рабочем каталоге приложения будут находиться два файла с расширением JAR и JAD, и в таком виде можно перенести программу в мобильный телефон посредством Интернета или компьютера, связанного с мобильным телефоном любым из способов. Все приложения и игры, написанные на Java 2 ME, распространяются именно таким образом. Файл с расширением JAR содержит упакованную программу, а файл с расширением JAD описывает содержимое JAR-файла.

При загрузке программ в телефон необходимо указывать путь к JAD-файлу, то есть дескриптору приложения, на основании атрибутов JAD-файла происходит работа Java-программы. Единственное исключение - это телефоны марки Siemens. При загрузке программ в телефон этой марки нужно указывать путь к JAR-файлу, о чем подробно будет рассказано в следующей главе, в которой будет рассматриваться программное обеспечение, поставляемое производителями мобильных телефонов для эмуляции работы телефонов различных моделей.

3.3. Инструментарий NetBeans IDE

Компания Sun Microsystems имеет в своем активе множество различных интегрированных сред программирования, направленных на работу с языком Java. Одним из таких инструментариев является NetBeans IDE. Этот инструмент предназначен для работы как с платформами Java EE, Java SE, так и с мобильной платформой

Java 2 ME. В этом случае дополнительно к NetBeans IDE необходимо установить Mobility Pack, который встраивает необходимые элементы для создания мобильных приложений в NetBeans IDE.

В отличие от предыдущего рассмотренного инструментария, NetBeans IDE имеет полный набор функциональных возможностей для работы с мобильными программами. В состав инструментария NetBeans IDE входят текстовый редактор с отличной интеллектуальной системой, компилятор, линковщик, отладчик, упаковщик программ Java 2 ME, большая справочная система и еще множество других различных утилит. Это отличный бесплатный визуальный инструментарий с огромным потенциалом!

По сравнению с инструментариями Sun ONE Studio 4 Mobile Edition и Sun Java Studio Mobility 6 2004Q3 инструментарий NetBeans IDE значительно проще в использовании и, как мне кажется, на порядок мощнее. Например, тот факт, что, сформировав один проект в NetBeans IDE, вы можете прямо из окна текстового редактора этого проекта (не закрывая редактора или создавая новый проект) создать неограниченное количество установочных пакетов под любой телефон, говорит сам за себя!

3.3.1. Установка NetBeans IDE

Перейдем к установке NetBeans IDE на ваш компьютер. Весь этап инсталляции инструментария будет рассмотрен в пошаговом режиме.

1. На компакт-диске в папке \IDE находится установочный пакет инструментария NetBeans IDE с названием netbeans-5_0-windows. Кликните на названии этого файла два раза левой кнопкой мыши. В первом диалоговом окне инсталлятора программы вы увидите приветственное сообщение (рис. 3.15). Для продолжения установки NetBeans IDE нажмите кнопку **Next**.

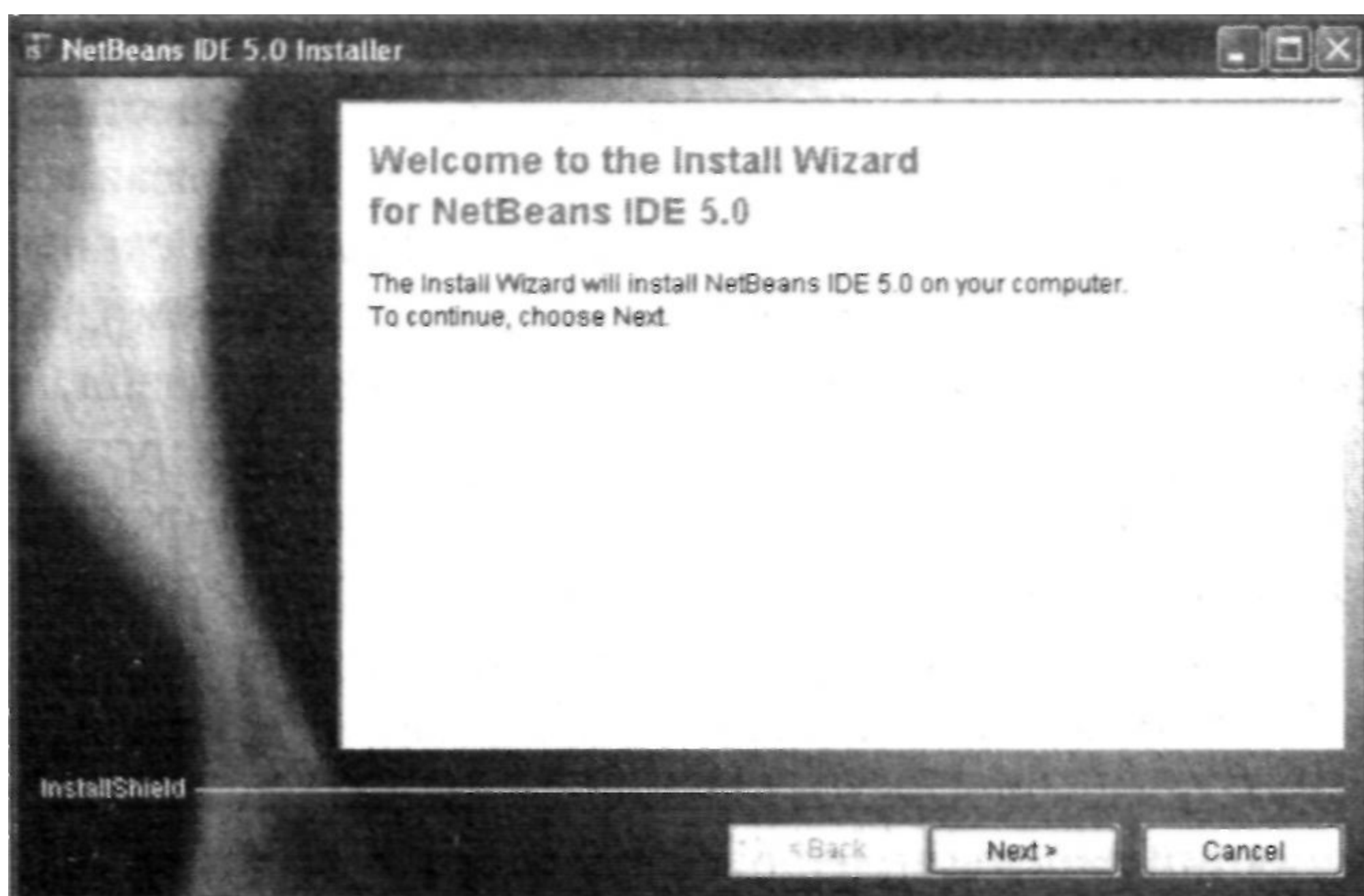


Рис. 3.15. Окно приветствия инсталлятора NetBeans IDE

2. Второе диалоговое окно содержит текст лицензионного соглашения. Выберите в этом окне флажок **I accept the terms in the license agreement** и нажмите кнопку **Next**, чтобы продолжить установку инструментария (рис. 3.16).

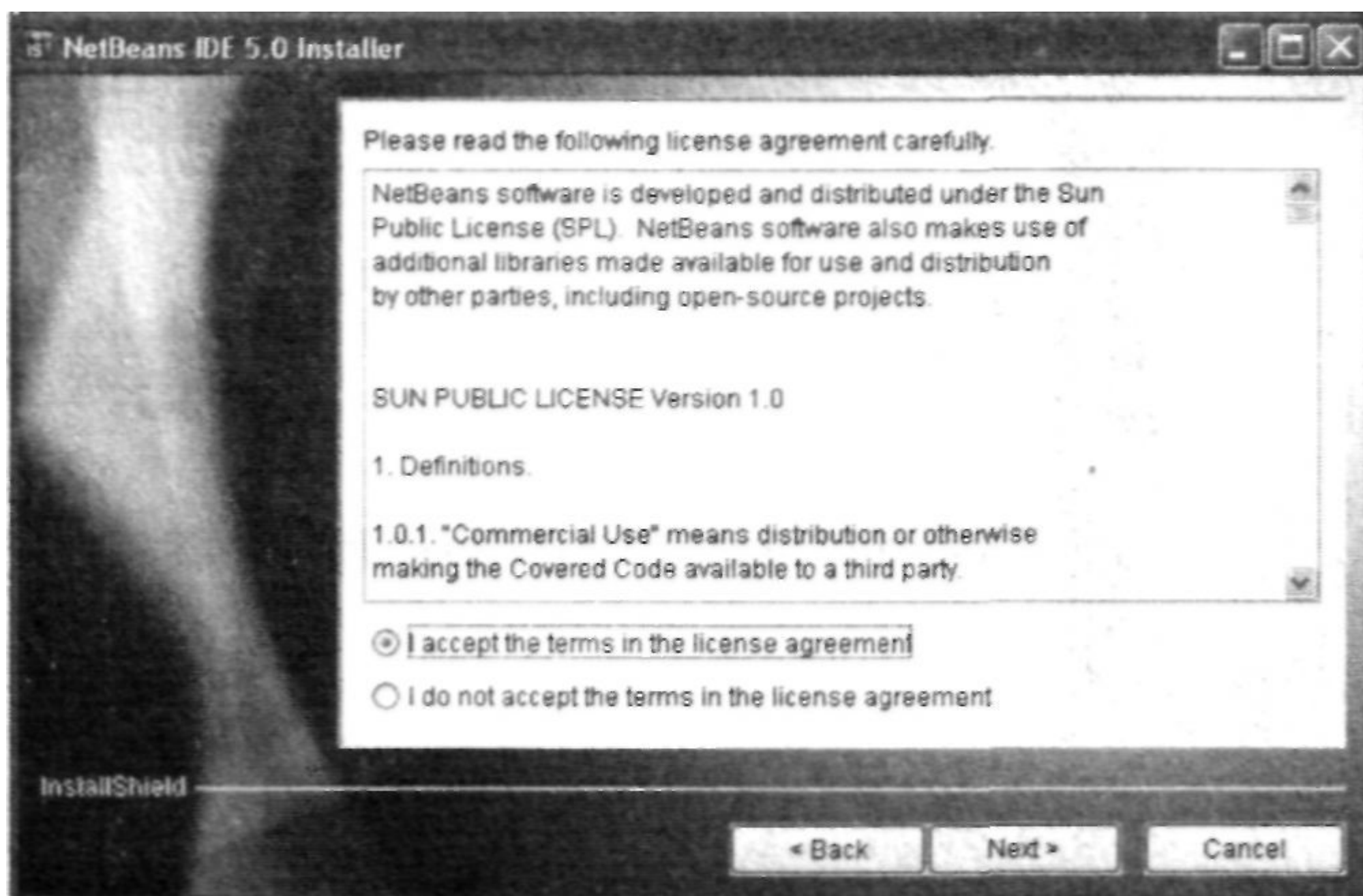


Рис. 3.16. Лицензионное соглашение

3. Следующее диалоговое окно позволяет задать каталог для установки инструментария NetBeans IDE (рис. 3.17). По умолчанию предлагается новый каталог в папке **Program Files**. В этот самый каталог лучше всего и ставить NetBeans IDE. Ограничений на установку NetBeans IDE в другую папку не имеется, единственное условие - это обязательный выбор диска «С», с других дисков (даже виртуальных) могут возникать проблемы при компиляции и сборке проектов. Но проще и лучше всего оставить директорию по умолчанию, предлагаемую инсталлятором, в этом случае проблем точно не будет.

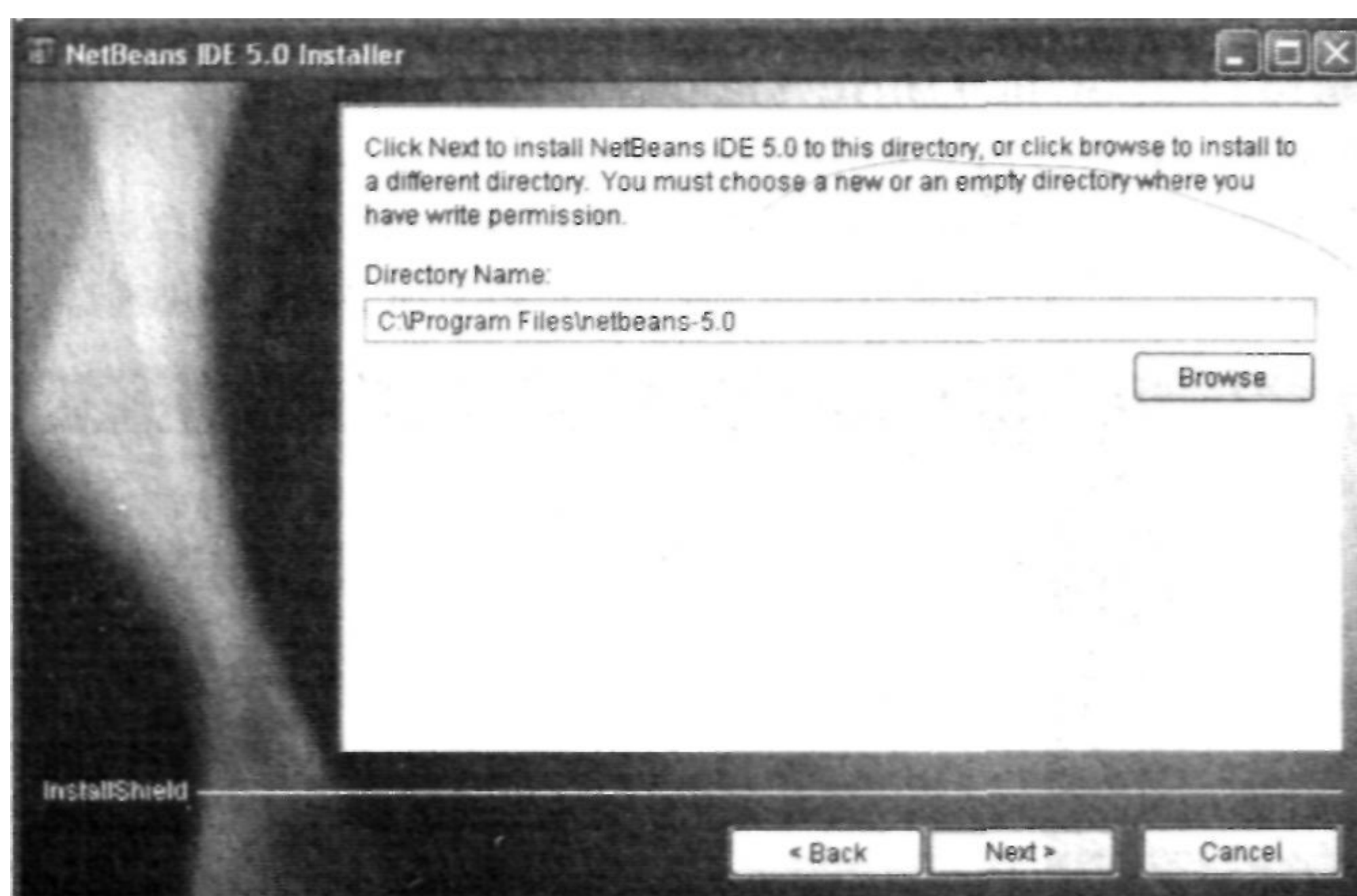


Рис. 3.17. Окно выбора директории установки

4. В следующем диалоговом окне установщик NetBeans IDE автоматически определяет папку на вашем компьютере с установленным Java SDK. Как мы договорились ранее, этот комплект разработчика у нас установлен в корневом каталоге диска «С». Поэтому на рис. 3.18 установщик определил путь

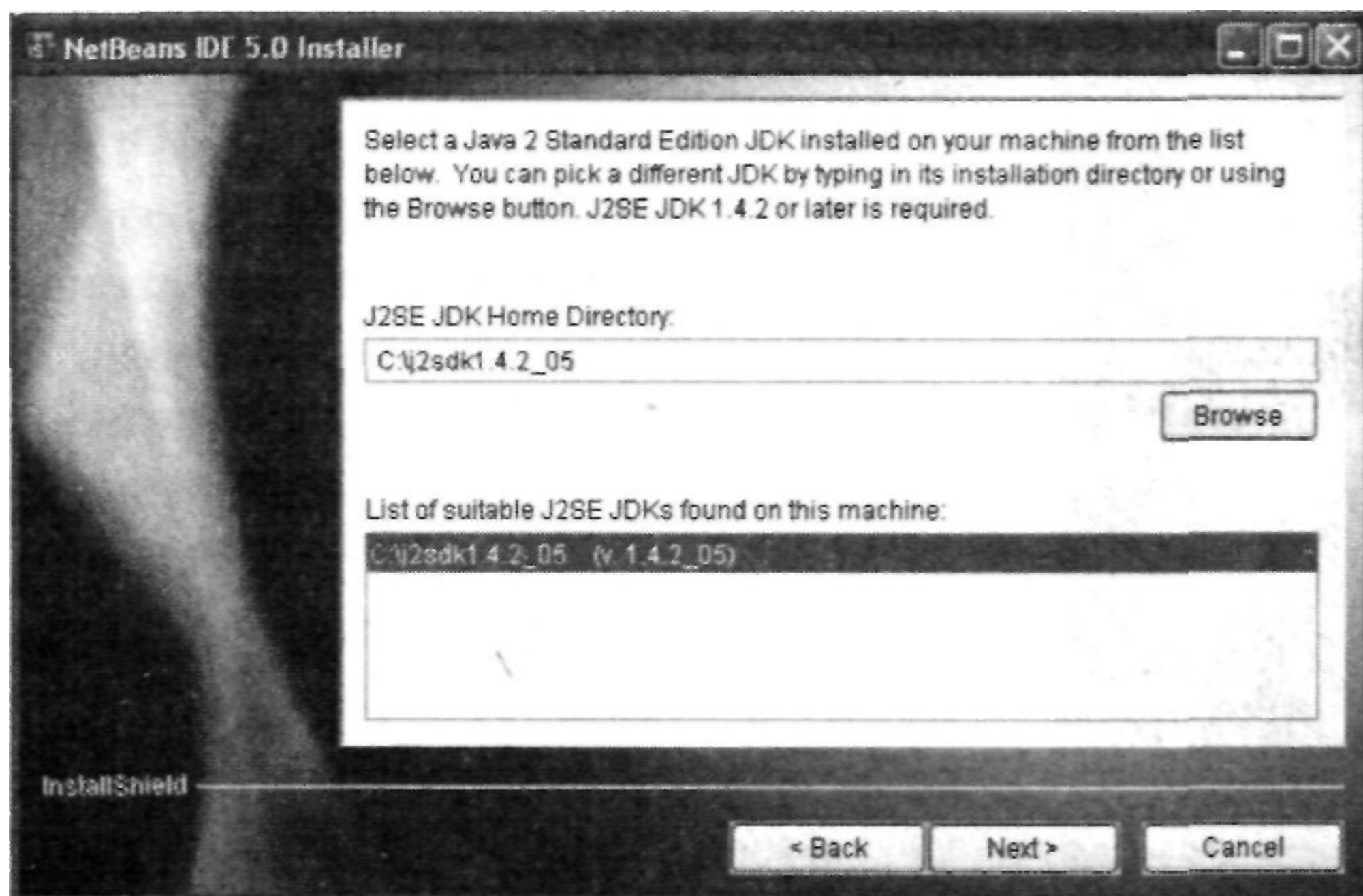


Рис. 3.18. Выбор директории с установленным Java SDK

к Java SDK как путь `c:\j2sdk1.4.2_05`. У вас, так же как и на этом рисунке, должен совпасть путь к Java SDK. Для продолжения нажимаем кнопку **Next**.

5. Затем откроется еще одно диалоговое окно, в котором инсталлятор сообщит о требуемом размере свободного пространства на диске и укажет папку, куда именно будет происходить установка Net Beans IDE (рис. 3.19). Нажав в этом окне кнопку **Next**, вы запустите процесс установки инструментария на ваш компьютер (рис. 3.20). Инсталляция программы идет примерно 3-5 минут, в зависимости от мощности вашей системы. После установки инструментария появится последнее диалоговое окно (рис. 3.21). В этом окне будет сообщаться обо всех произведенных этапах установки. Для окончания процесса инсталляции нажмите кнопку **Finish**. На этом процесс установки Net Beans IDE окончен, и можно плавно переходить к инсталляции Mobilit у Pack. Этот мобильный дополнительный пакет позволит нам создавать в инструментарии NetBeans IDE мобильные приложения.

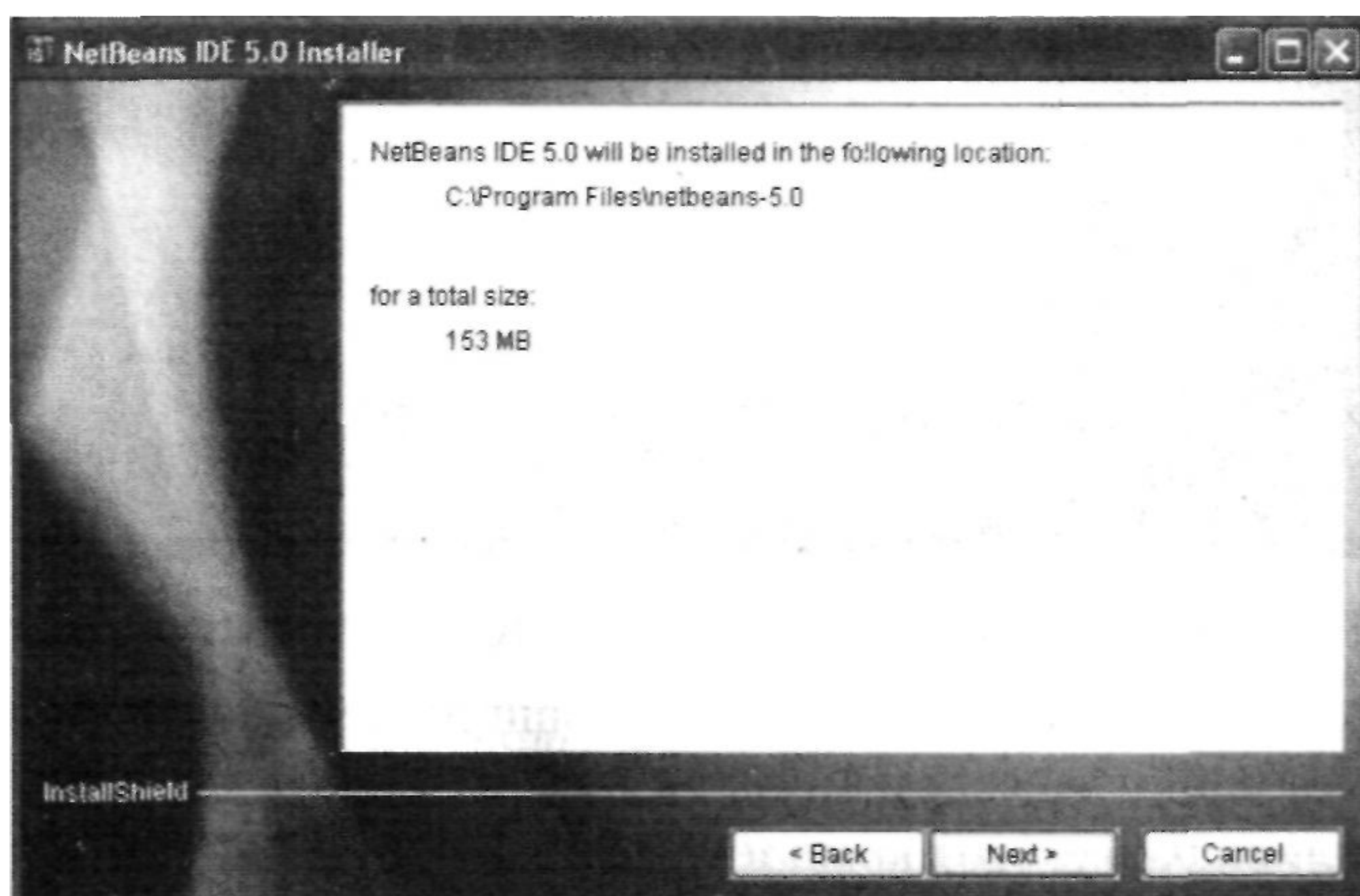


Рис. 3.19. Информационное сообщение

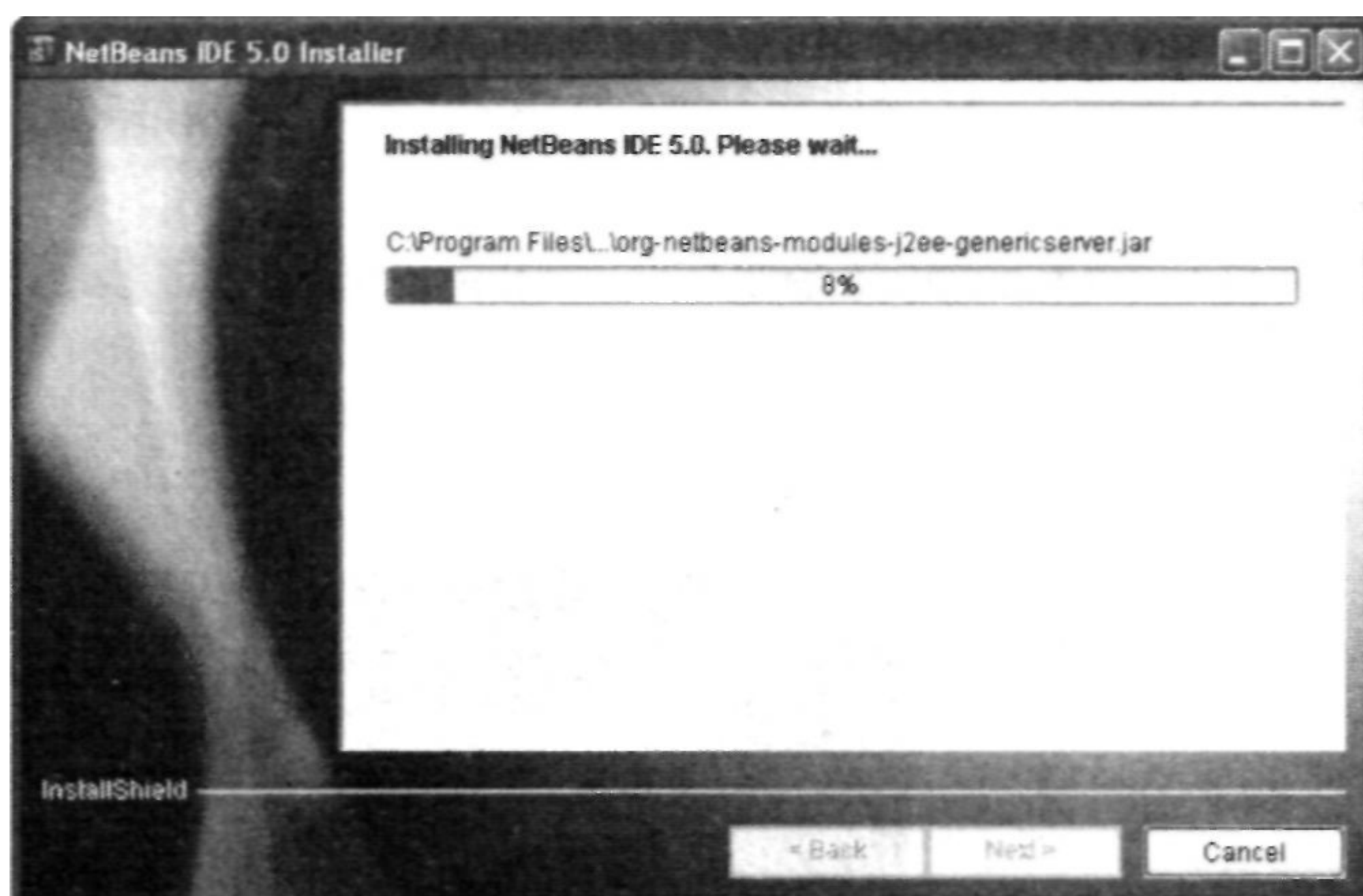


Рис. 3.20. Процесс установки NetBeans IDE на ваш компьютер

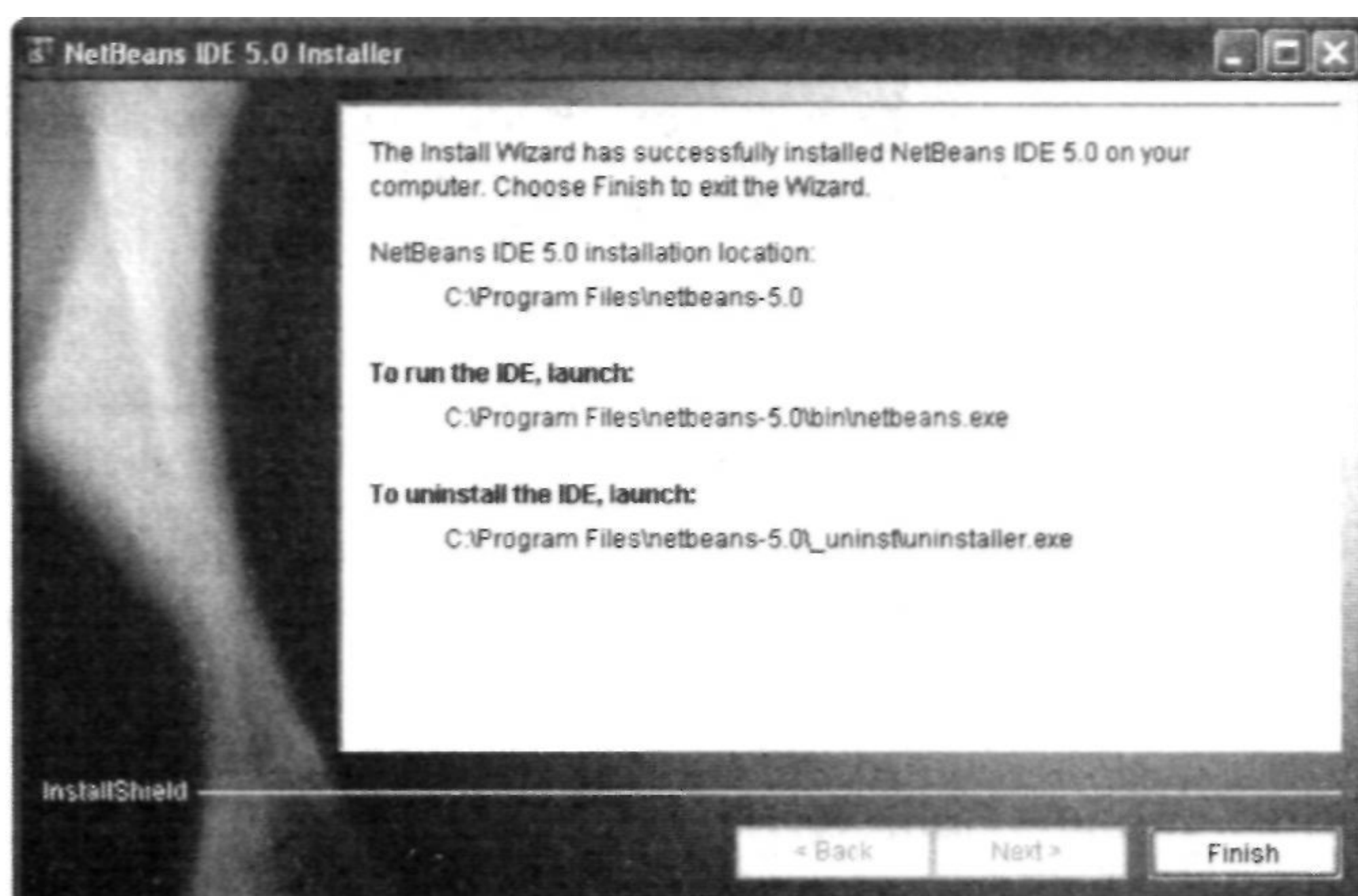


Рис. 3.21. Финальное диалоговое окно

3.3.2. Установка *Mobility Pack*

На компакт-диске к книге в папке \IDE также находится Mobility Pack для NetBeans IDE. Это файл с названием netbeans_mobility-5_0-win. Установка Mobility Pack у вас не должна вызвать проблем, поскольку она абсолютно идентична как по набору диалоговых окон, так и по самому ходу установки только что рассмотренной инсталляции NetBeans IDE. Прделайте весь процесс инсталляции Mobility Pack самостоятельно, затем запустите NetBeans IDE.

3.3.3. Создание проекта

Открыть инструментарий можно командами **Пуск => Все программы => NetBeans 5.0 => NetBeans IDE**, либо воспользоваться ярлыком инструментария,

находящимся на рабочем столе, который появляется сразу после установки NetBeans IDE. При первом открытии NetBeans IDE появляется рабочее окно, изображенное на рис. 3.22.

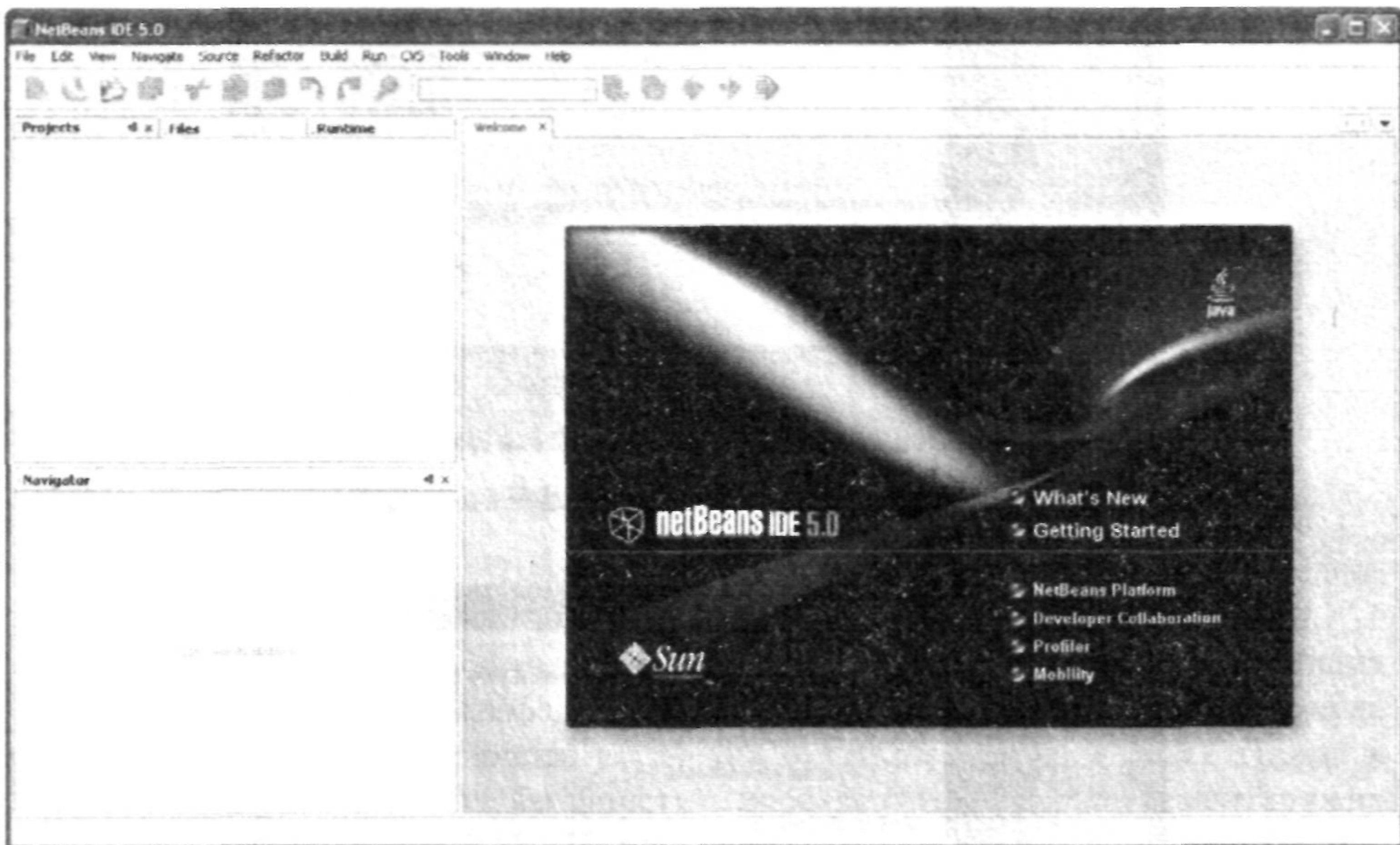


Рис. 3.22. Рабочее окно NetBeans ID

В рабочей области окна инструментария с левой стороны каскадом располагаются два окна для отображения всех задействованных элементов открытого проекта, а самая большая часть рабочего окна отведена под текстовый редактор. При первом старте в пространстве текстового редактора вы увидите так называемую стартовую страницу приветствия NetBeans IDE.

Чтобы создать новый проект или проект на базе шаблона NetBeans IDE, в открытом окне инструментария выполните команды **File => New Project**. Откроется диалоговое окно **New Project** (рис. 3.23). В этом окне в списке **Categories** находятся несколько папок, одна из папок с названием **Mobile** дает вам возможность создать проект на базе различных шаблонов. Выделив папку **Mobile** в списке **Categories**, во втором списке **Project** вы увидите все доступные виды проектов. В поле **Description** будет находиться описание текущего выбранного вида проекта. Всего имеется пять следующих проектов:

- **Mobile Application** - выбор шаблона создаст простое мобильное приложение с одним основным классом мидлета. Этот простейший проект подобен исходному коду, рассмотренному нами при создании первого проекта в Wireless Toolkit Project;
- **Mobile Class Library** - стартовый шаблон для создания библиотеки;
- **Mobile Project from Existing MIDP Sources** - этот шаблон позволяет создать проект на базе имеющегося у вас набора готовых классов;

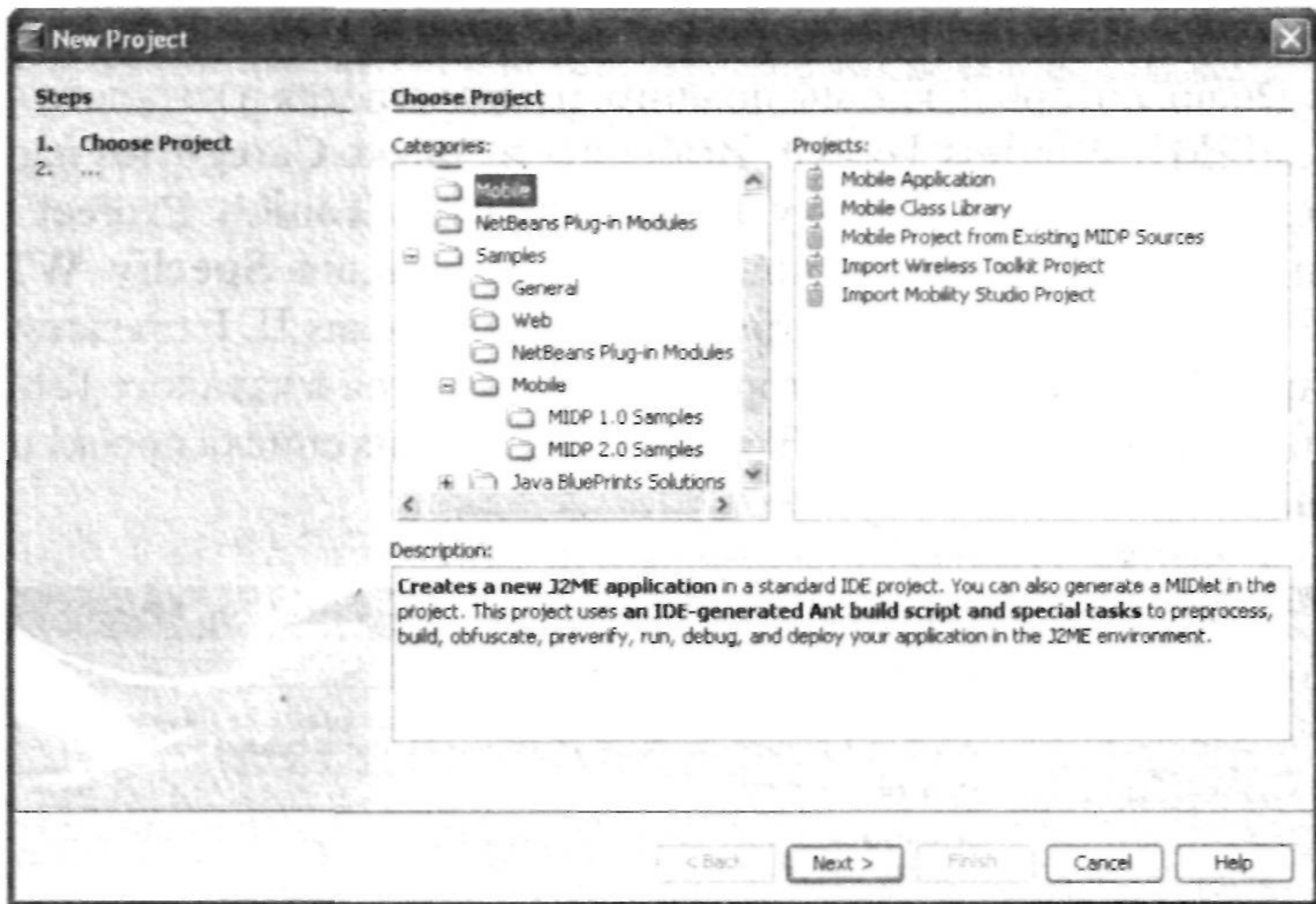


Рис. 3.23. Создание нового проекта

- **Import Wireless Toolkit Project** - импортирует любой проект из инструментария J2ME Wireless Toolkit Project;
- **Import Mobility Studio Project** - импортирует любой проект из предыдущих версий инструментария Mobility Studio.

Дополнительно в списке Categories находится еще ряд папок для работы уже с платформами Java EE и Java SE, но одна папка с названием **Samples** содержит вложенную папку **Mobile** (рис. 3.24). В этой папке располагаются различные демонстрационные программы, на базе которых вы также можете создать собственный проект.

Далее в пошаговом режиме мы сформируем новый проект.

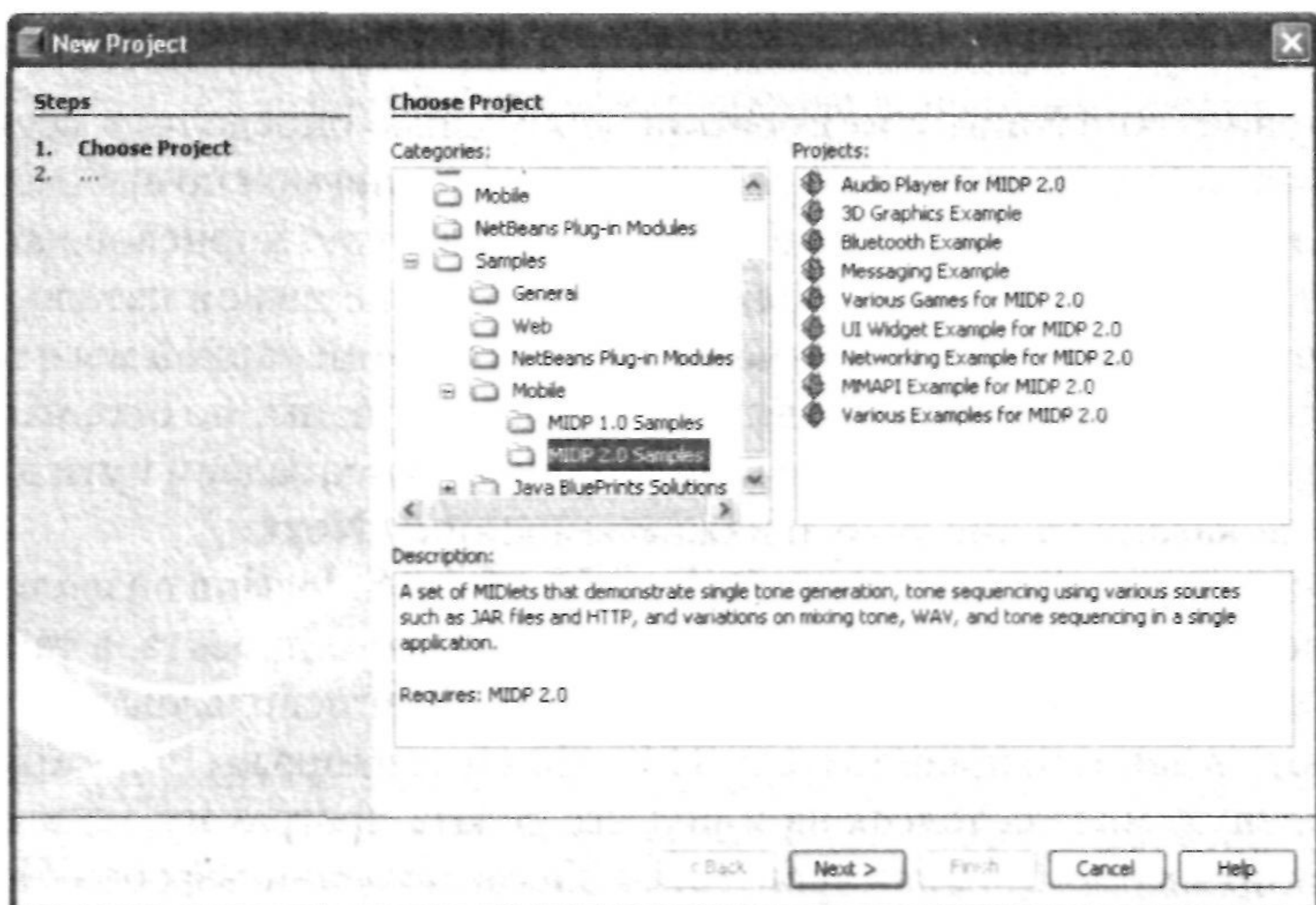


Рис. 3.24. Папка Samples с вложенной папкой Mobile

1. Для примера создадим проект на базе импорта нашего предыдущего проекта Demo, который, как мы помним, располагается в каталоге инструментария J2ME Wireless Toolkit. Выберите в списке **Categories** папку **Mobile**, а в списке **Project** - строку **Import Wireless Toolkit Project** и нажмите кнопку **Next**. Откроется новое диалоговое окно **Specify WTK Project** (рис. 3.25). В этом окне инструментарий NetBeans IDE представит вам все обнаруженные проекты, которые располагаются в каталоге J2ME Wireless Toolkit (это папка `c:\WTK23\apps`). Выберите из списка проект с названием **Demo** и нажмите кнопку **Next**.

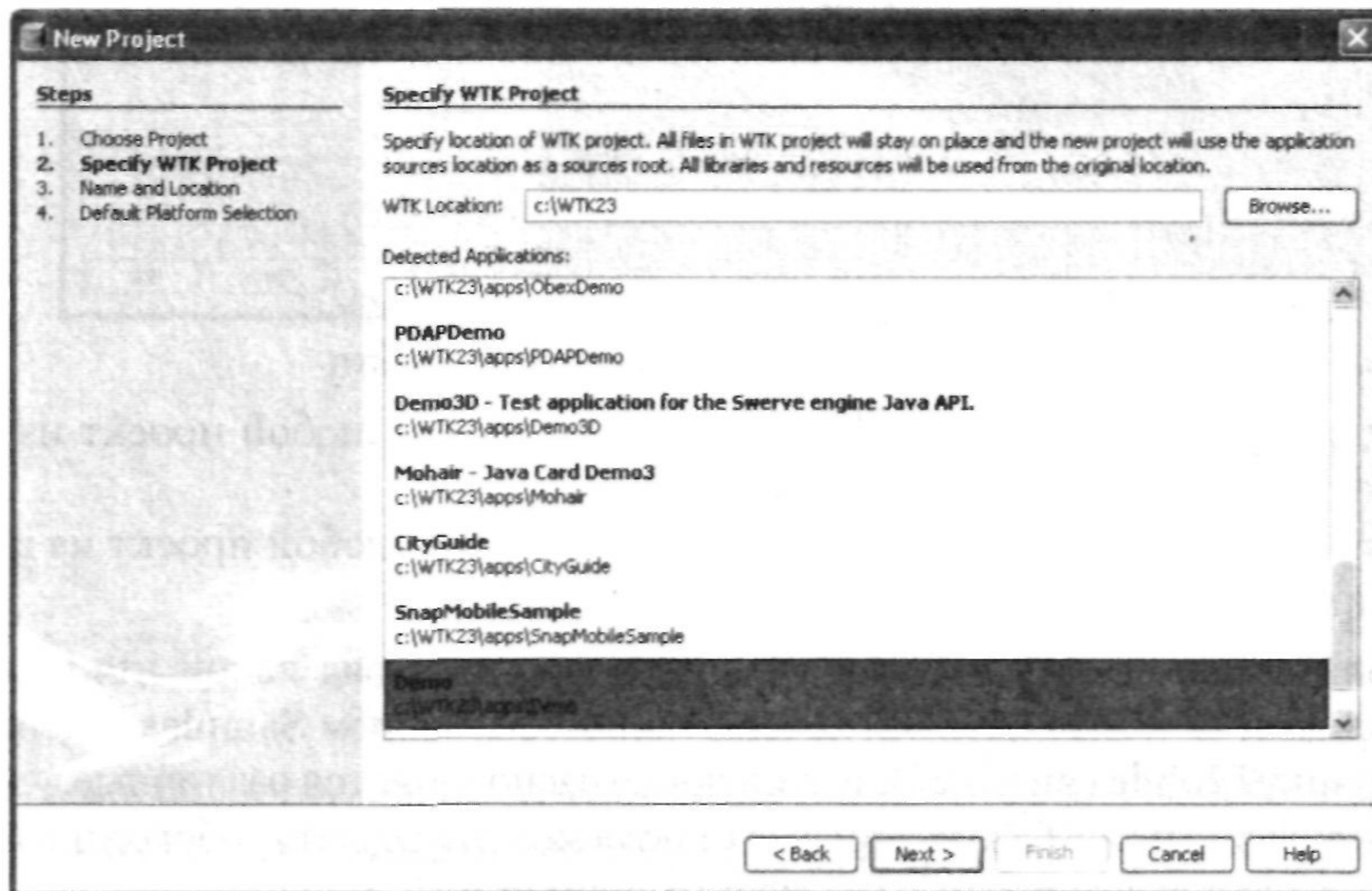
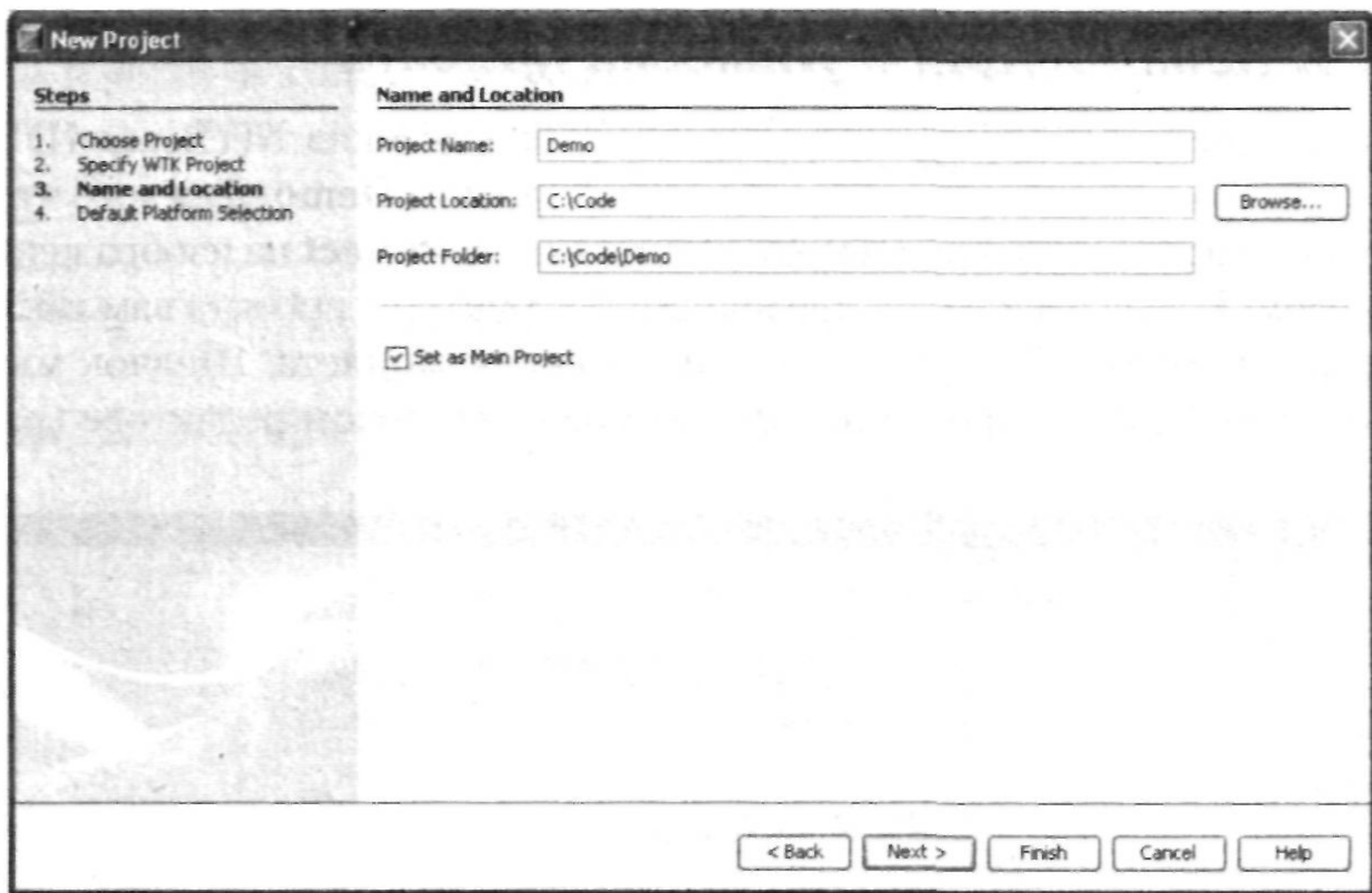
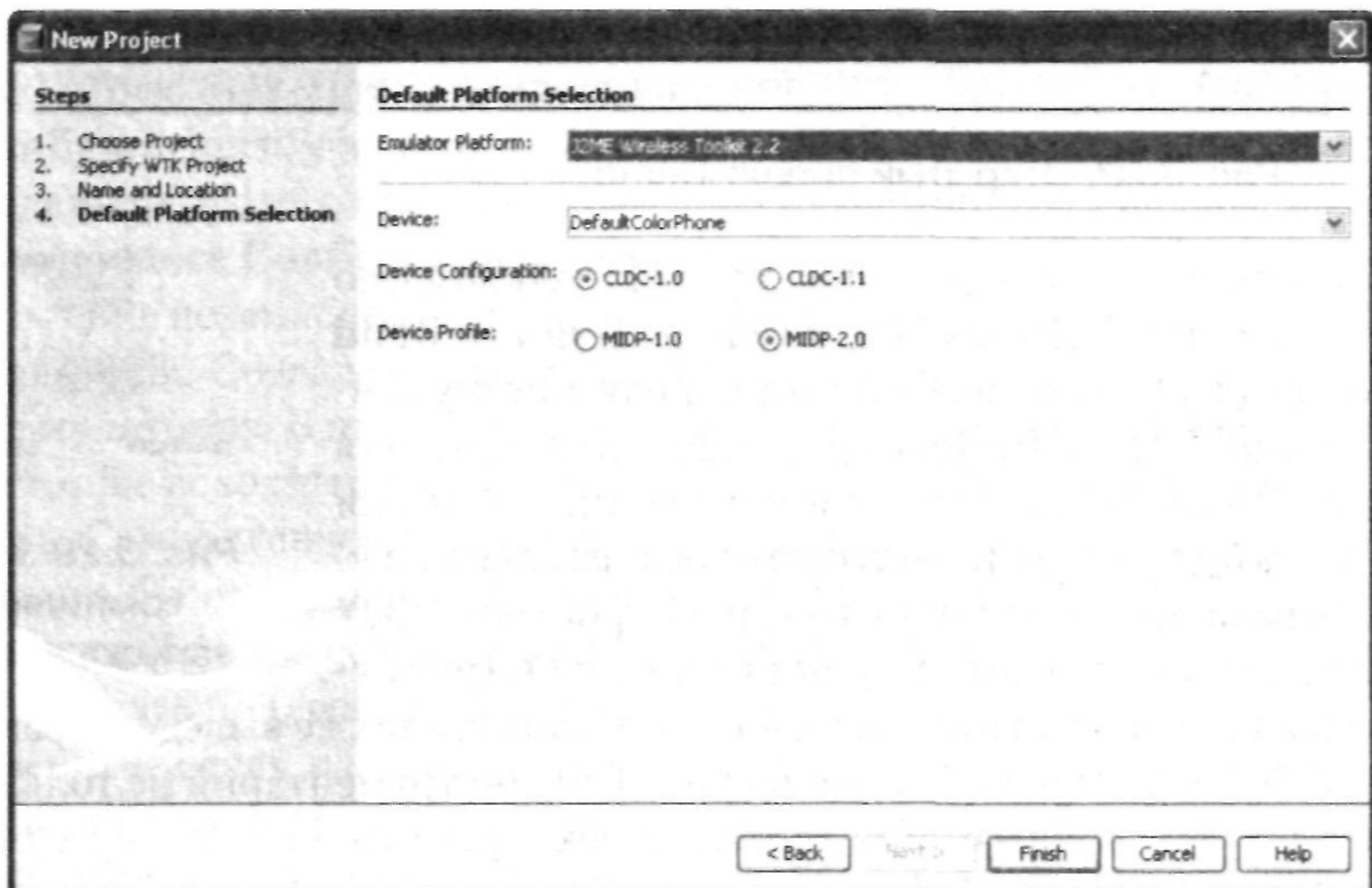


Рис. 3.25. Выбор проекта для импорта

2. В новом окне **Name and Location** необходимо определить место для дислокации будущего проекта. Место для хранения можно выбрать любое, **но только на диске "C"**, а еще лучше, если это будет корневой каталог, тогда проблем точно не будет. Поэтому выбираем корневой каталог диска «C» и создаем там папку с названием Code. Вот с этой папкой мы и будем работать на протяжении всей книги, и именно эту папку по окончании работы над книгой я положу вам на компакт-диск. Итак, задаем пути для проекта, как показано на рис. 3.26, и нажимаем кнопку **Next**.
3. Последнее диалоговое окно **Default Platform Selection** позволяет выбрать конфигурацию и профиль для импортируемого проекта, а также указать мобильный эмулятор телефона, который будет использоваться по умолчанию для данного проекта (рис. 3.27). По умолчанию вы можете задать определенную модель телефона и под нее делать программу, но впоследствии этот проект можно адаптировать под другие модели телефонов. На этом этапе задается модель телефона, которая будет использоваться по умолчанию



и являться основной платформой для разработки приложения. Выбор же эмулятора производится в списке **Device** диалогового окна **Default Platform Selection**. Сейчас мы имеем ограниченное количество эмуляторов, предоставленных NetBeans IDE, но уже в следующей главе мы интегрируем в инструментарий эмуляторы от компаний Motorola, Nokia, Sony Ericsson и Benq-Siemens. Изберите в списке **Device** эмулятор с названием **DefaultColorPhone** и нажмите кнопку **Finish**.



3.3.4. Компиляция и упаковка проекта

После импорта проекта в правой части рабочего окна NetBeans IDE в окне **Project** появятся импортированные элементы проекта **Demo**. Для того чтобы раскрыть проект, щелкните левой кнопкой мыши в окне **Project** на изображении квадрата с плюсом. После раскрытия древовидной структуры проекта вам также будет доступен файл `HelloMIDlet.java` с исходным кодом мидлета. Щелчок мышью по названию этого файла откроет его содержимое в текстовом редакторе (рис. 3.28).

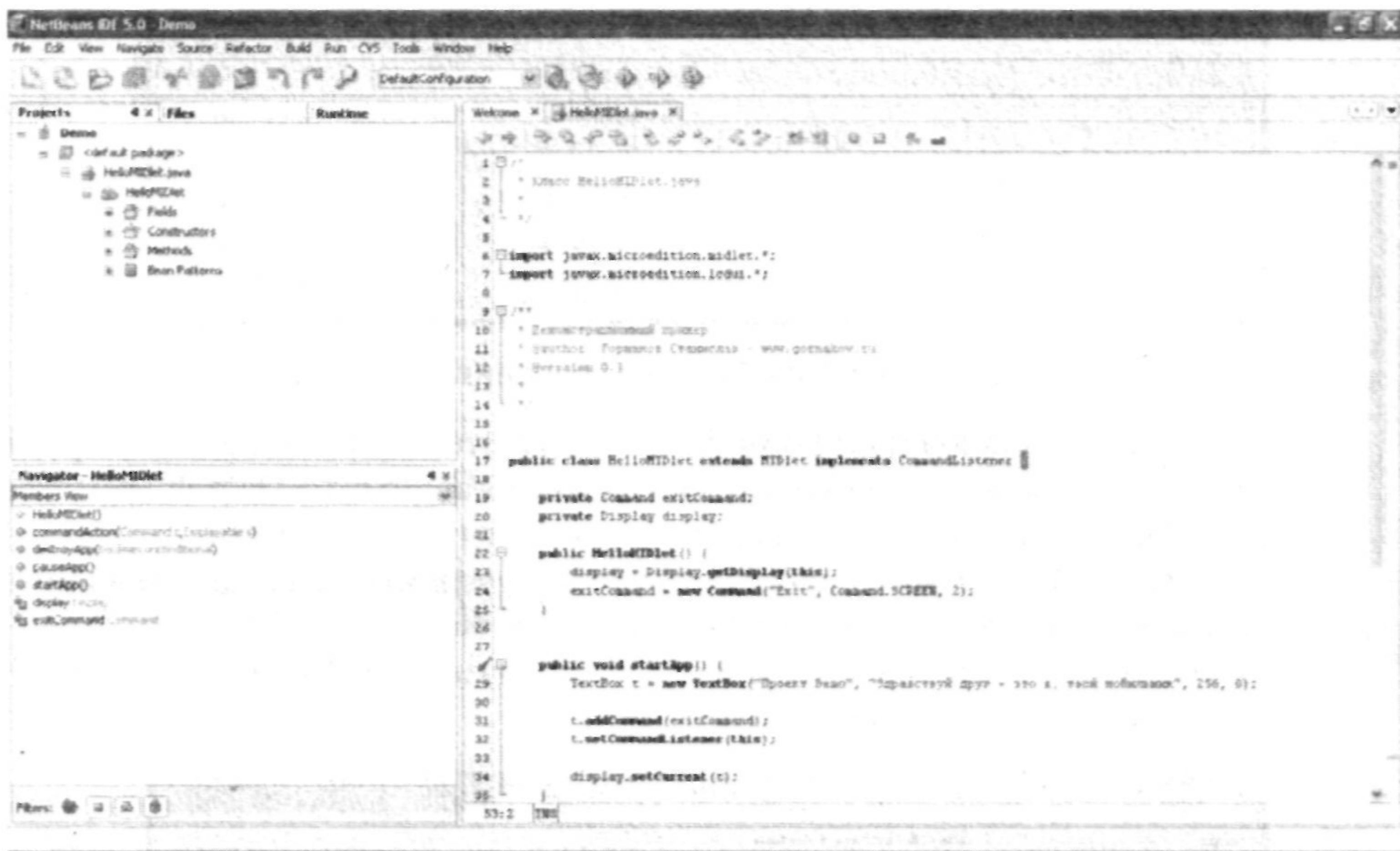


Рис. 3.28. Открытый проект Demo

Чтобы откомпилировать весь проект, необходимо выполнить в меню NetBeans IDE команды **Build => Build Main Project** (F11) либо выбрать на панели инструментов значок **Build Main Project** (рис. 3.29). Эти действия приведут NetBeans IDE в состояние компиляции и сборки проекта, о чем будет свидетельствовать появившаяся в нижней части экрана инструментария панель **Output** (рис. 3.30). По окончании процесса компиляции в панели **Output** будет приведена статистика произведенных операций, а также выведены различные сообщения. В частности, как видно из рис. 3.30, инструментарий не только скомпилировал проект, но сразу и упаковал проект, расположив файл JAR в папке **C:\Code\Demo\dist\Demo.jad**.



Рис. 3.29. Кнопки компиляции и запуска эмулятора



Рис. 3.30. Панель Output

3.3.5. Добавление в проект новых эмуляторов

Формируя новый проект на этапе выбора эмулятора (диалоговое окно **Default Platform Selection**), у нас была возможность избрать только один эмулятор, потому что мы импортировали проект из J2ME Wireless Toolkit. Если бы мы создавали проект с нуля, выбрав, например, шаблон **Mobile Application**, то после окна **Default Platform Selection** появилось еще одно дополнительное окно с названием **More Configurations Selection**. Посмотрите на рис. 3.31, где представлена ситуация с созданием пустого проекта на базе шаблона **Mobile Application** и новое диалоговое окно **More Configurations Selection**.

В этом окне перечислены все интегрированные в NetBeans IDE на данный момент эмуляторы. Создавая проект, вы можете избрать из этого списка сразу несколько эмуляторов. В этом случае проект будет иметь несколько конфигураций и один и тот же исходный код, можно будет адаптировать для различных моделей телефонов. Самое главное, что после компиляции и упаковки проекта для каждого эмулятора будет создана отдельная папка в каталоге, который вы отвели для всего проекта. То есть, создавая один раз проект, вы в реальном времени можете его адаптировать под любые интегрированные в IDE эмуляторы.

В том случае если на этапе создания проекта вы не указали один из эмуляторов или, как в нашем случае, импортировали проект из J2ME Wireless Toolkit, то в текущий проект можно явно добавить один и более эмуляторов. Для добавления

нового эмулятора в текущий проект выполните команды меню Edit => Processor Blocks => Add Configurations to Project. Либо щелкните правой кнопкой мыши в окне текстового редактора. В появившемся контекстном меню выберите команды Processor Blocks => Add Configurations to Project. В ответ на эти действия откроются сразу два диалоговых окна, одно поверх другого (рис. 3.32).

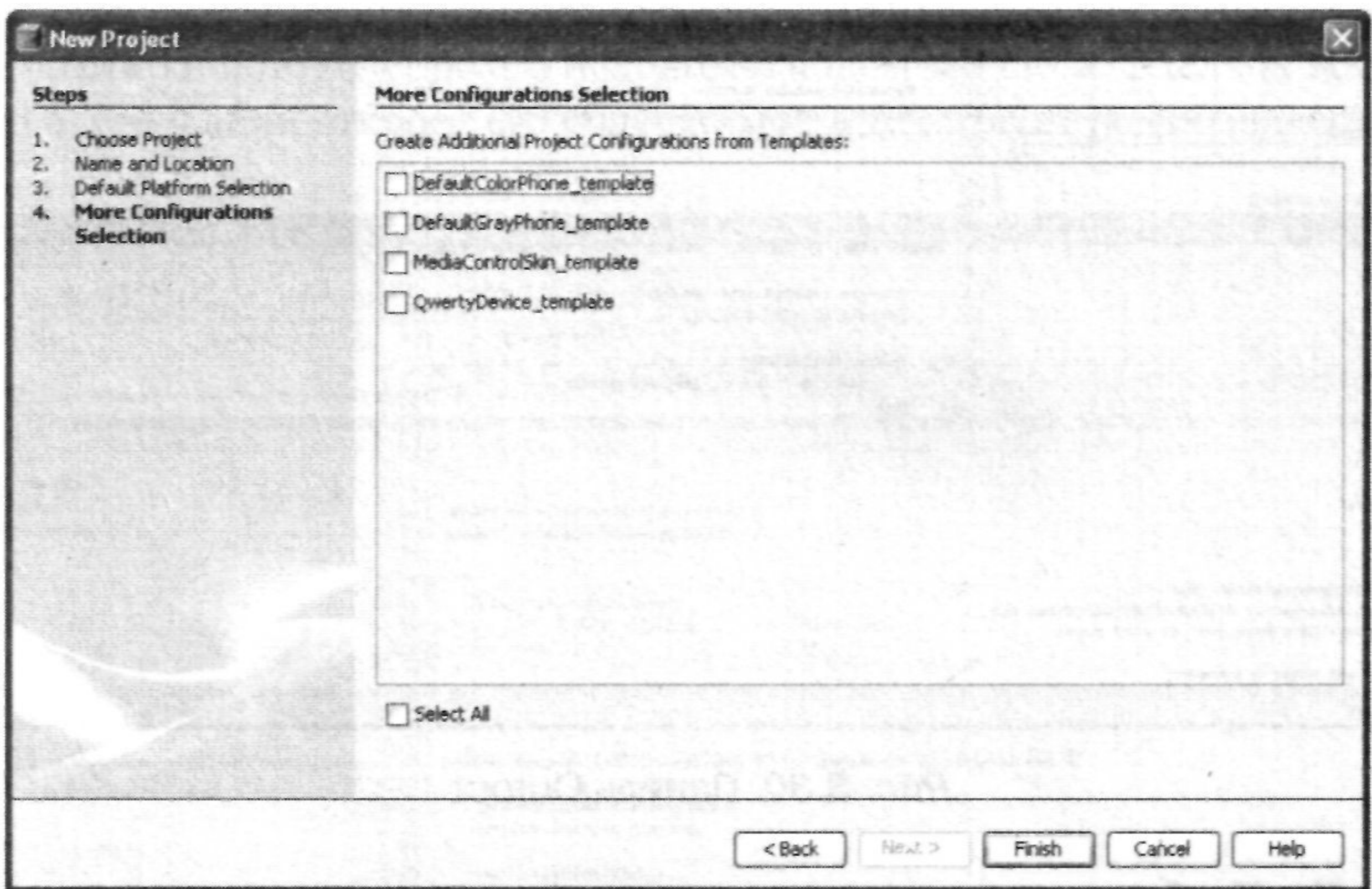


Рис. 3.31. Окно More Configurations Selection

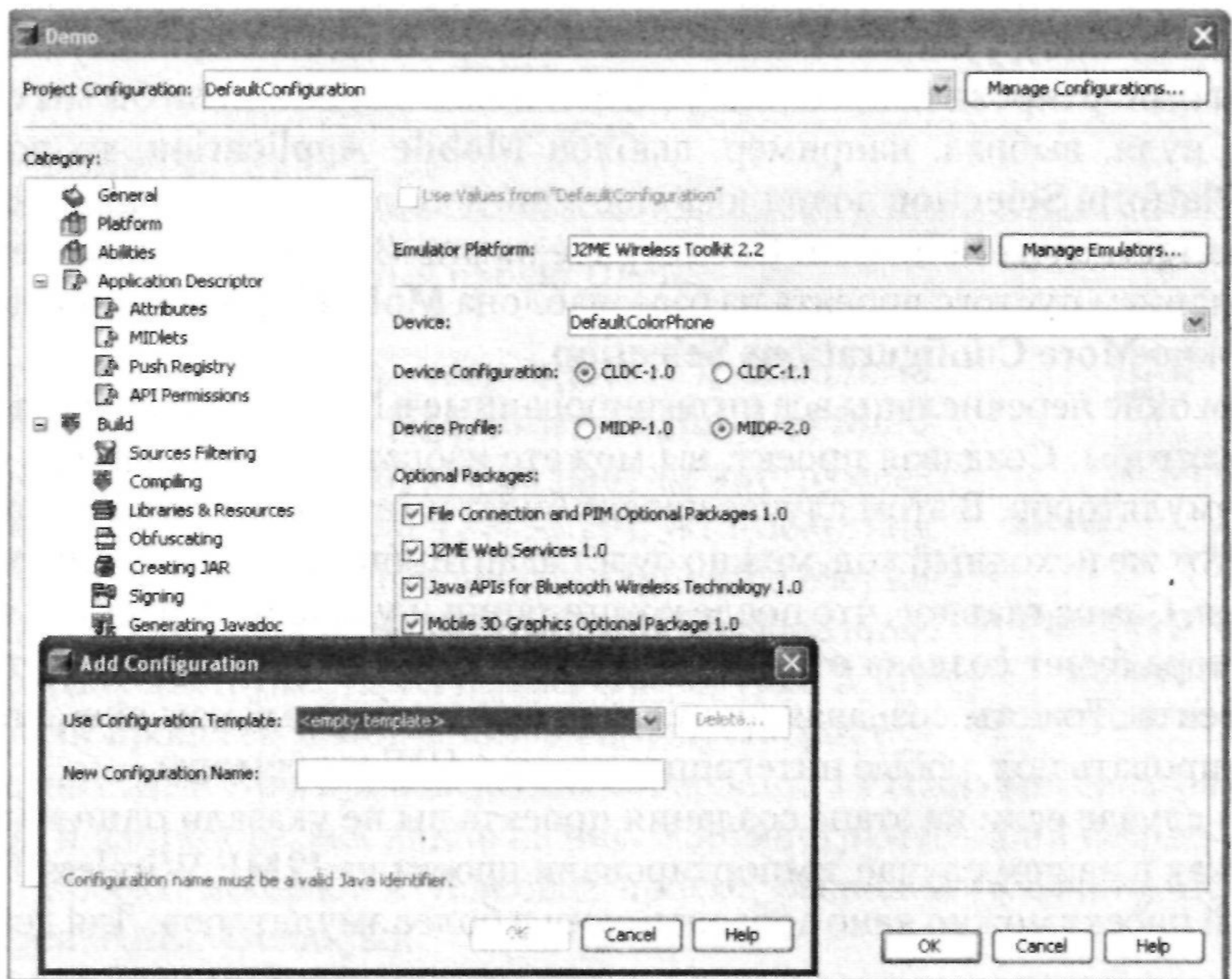


Рис. 3.32. Изменяем эмулятор

В маленьком окне **Add Configurations** возможно задать новую конфигурацию для проекта, например назвав такую конфигурацию как Nokia N93. Затем, нажав кнопку ОК, вы попадете в большое окно, где с помощью менеджера эмуляторов можете выбрать необходимый для этой конфигурации эмулятор. Все перечисленное работает только в том случае, если в инструментарий NetBeans IDE были ранее интегрированы дополнительные SDK от производителей телефонов. Если нет, то в маленьком окне **Add Configurations** достаточно нажать **Cancel**, а в большом окне с названием текущего проекта в списке **Device** - просто сменить эмулятор. То есть в стандартной поставке NetBeans IDE имеется возможность только смены эмуляторов из состава инструментария, а вот после установки SDK от различных производителей телефонов вы уже будете иметь возможность использовать и другие эмуляторы. Поэтому давайте переходить к следующей главе, где мы обязательно рассмотрим и способы интеграции SDK в NetBeans IDE.

Глава 4. Телефонные эмуляторы

Рассмотренные в предыдущих главах инструментарии имеют в своем составе эмуляторы телефонов для тестирования создаваемой программы на компьютере. Эти телефонные эмуляторы не имеют прототипов в реальной жизни и содержат некий собирательный образ различных марок телефонов с минимальным набором технических характеристик. К сожалению, на данных эмуляторах не всегда можно достоверно протестировать создаваемую программу. С другой стороны, иметь три-четыре десятка телефонов различных марок могут позволить себе компании, серьезно занимающиеся созданием игр для рынка мобильных развлечений. Можно, конечно, приобрести и отдельные экземпляры телефонов, характеризующие в общих чертах определенную серию телефонов разных производителей.

Иметь некоторое количество реальных телефонов - это хорошо, но, работая в течение дня над программным кодом, вы вносите ряд изменений в программу и желаете ее тут же протестировать. Нельзя по десять раз на день упаковывать приложение, переносить его на телефон и устанавливать только для того, чтобы посмотреть, как будет выглядеть, например, новый шрифт в игре. Безусловно, такие тесты необходимы, потому что разница, как выглядит графический контекст на экране монитора и на экране телефона, очень большая. Вот только периодичность, с которой необходимо выполнять эти действия, должна быть в пределах разумного. Например, при создании уровня игры, главного персонажа, врагов, артефактов это делать необходимо, и совсем не обязательно загружать игру в телефон, чтобы посмотреть, как вы обработали столкновения корабля с кромкой экрана и т. д. Для этих целей проще использовать эмуляторы телефонов.

Большинство производителей телефонов имеют средства, которые позволяют эмулировать работу реальных телефонов на компьютере. Такие средства носят название Software Developer Kit, или SDK (Программный пакет разработчика). Способ распространения SDK у всех производителей примерно одинаков - это абсолютно бесплатное программное обеспечение, которое можно скачать с сайта компании - производителя телефона. Максимум, что от вас может потребоваться, - это регистрация на сайте, и не более того. Все рассмотренные в книге SDK вы сможете найти на компакт-диске, идущем в комплекте с книгой.

В состав SDK, как правило, входит разное количество телефонных эмуляторов с возможностью подключения новых эмуляторов, средства для тестирования, мониторинга и профилирования программ, средства для работы со звуком, Bluetooth, GPRS, камерой. В целом это готовый и работоспособный комплект программного обеспечения, предоставляющий программисту полный спектр возможностей в разработке и тестировании программы на компьютере без использования

реальных мобильных устройств. В этой главе мы рассмотрим SDK от компаний Motorola, Nokia, BenQ-Siemens, Sony Ericsson и Samsung.

4.1. Motorola

Американская компания Motorola выпускает огромное количество разнообразных мобильных устройств с платформой Java 2 ME, а соответственно, и поддержка программистов организована на высоком уровне. На сайте компании в Интернете по адресу <http://developer.motorola.com> вы всегда сможете найти множество различной документации и «свежее» программное обеспечение (рис. 4.1).

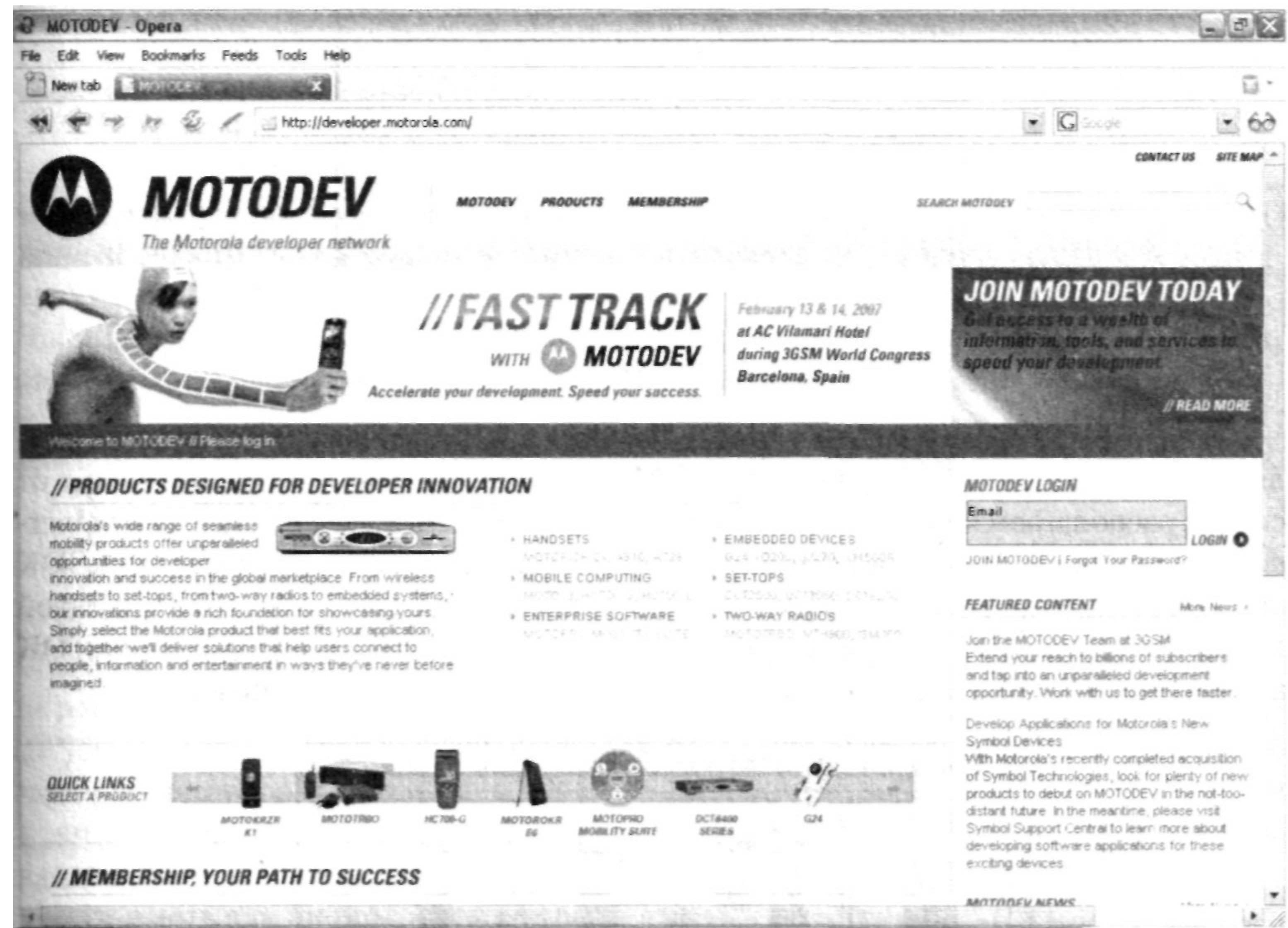


Рис. 4.1. Сайт компании Motorola

Пакет разработчика от компании Motorola распространяется в виде одной цельной программы. На компакт-диске эта программа находится в папках **\SDK\Motorola**. Двойной клик мыши на инсталляционном пакете запустит процесс установки SDK на ваш компьютер. Инсталляция проста, поэтому рассматривать ее отдельно не имеет смысла. Единственное условие - это установленная у вас на компьютере программа Quicktime. Найти последнюю версию этой программы вы сможете в Интернете по адресу <http://quicktime.apple.com>. Без Quicktime программное обеспечение от Motorola работает нормально, но почему-то требует Quicktime с настойчивой периодичностью. После установки SDK запустить

программу можно через команды **Пуск => Все программы => Motorola J2ME(TM) SDK v6.1.1 for Motorola OS Products => Motorola Launchpad**. После запуска программы откроется рабочее окно, представленное на рис. 4.2.

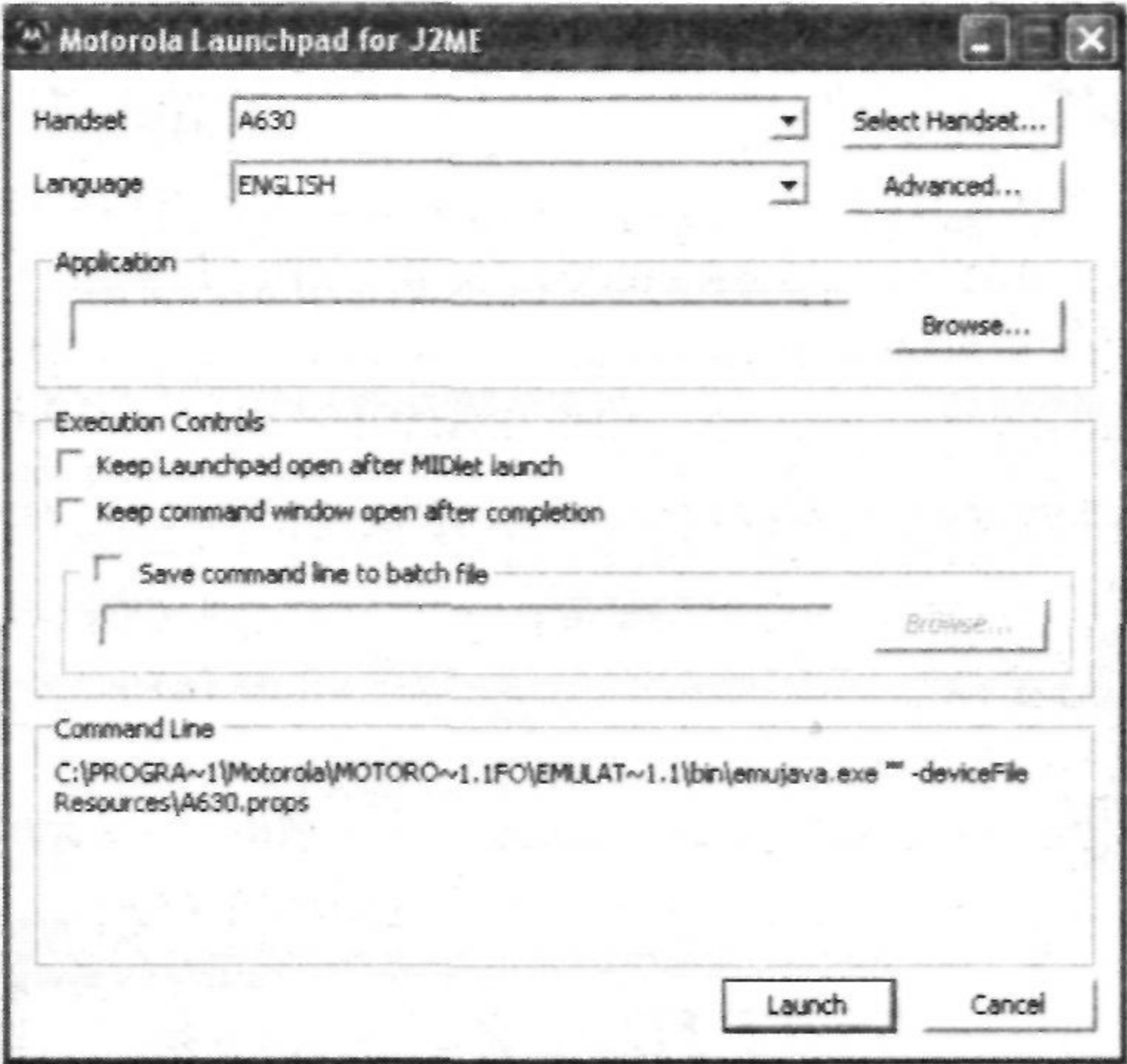


Рис. 4.2. Программа Motorola Lanchpad for J2ME

В рабочем окне программы в списке **Handset** представлены все имеющиеся эмуляторы телефонов с поддержкой Java 2 ME. Дополнительно с правой стороны от списка располагается кнопка **Select Handset**. Если нажать на эту кнопку, то откроется окно **Simulator Thumbnail Window**, представляющее все интегрированные в SDK эмуляторы в виде красочно оформленных изображений (рис. 4.3). Щелчок левой кнопкой мыши на любом изображении эмулятора и нажатие в этом окне кнопки ОК автоматически подставит выбранную модель телефона в список **Handset**.



Рис. 4.3. Окно Simulator Thumbnail Window

Рис. 4.4. Запуск программы на эмуляторе

Для запуска Java-приложения на эмуляторе выберите из списка **Handset** необходимую модель телефона, а затем в поле **Application** укажите путь к тестируемому приложению (к JAD-файлу). Сам путь можно прописать вручную, а можно воспользоваться кнопкой **Browse**. Далее нажмите на кнопку **Launch**. На экране монитора появится выбранный телефонный эмулятор, в котором будет произведен запуск выбранной Java-программы. В предыдущей главе мы создали простое приложение **Demo**, вот его давайте и запустим на одном из эмуляторов (рис. 4.4).

Программа Motorola Launchpad for J2ME после запуска эмулятора автоматически закрывается. Для того чтобы эта программа не закрывалась каждый раз при запуске нового эмулятора, нужно просто поставить галочку в поле **Keep Launchpad open after MIDlet Launch**. В конце этой главы вы узнаете, как можно интегрировать телефонные эмуляторы компании Motorola в инструментарий NetBeans IDE.



4.2. Nokia

Финская компания Nokia - одна из ведущих компаний в мире по производству телефонов. Это одна из старейших компаний по производству мобильных устройств. Дизайнерские решения компании в оформлении и стиле телефонов порой удивляют, чего стоит только телефон Nokia 7280 или совсем новенький телефон N-серии Nokia N91 с четырьмя гигабайтами памяти и двухмегапиксельной камерой под управлением Symbian OS 9. Качество телефонов этой компании как в дизайне, так и в аппаратном обеспечении одно из высочайших в мире и не зря заслуживает высокой оценки. Модельный ряд телефонов в своем обозначении может иметь как цифры, так и буквы.

Если в обозначении используются три цифры, то это телефоны стандарта NMT, четыре цифры - стандарт GSM, а если номер заканчивается числом 20 - это телефон стандарта AMPS. Во всех обозначениях первая цифра указывает на то, к какому из классов принадлежит устройство. Так, имеются следующие варианты обозначений:

- 1xxx и 2xxx - это самые простые телефоны класса Basic (Простые), имеющие минимальный набор функций и работающие на основе прошивки;
- 3xxx - это так называемая молодежная серия телефонов, которая может выделяться дизайном и работать как на прошивке, так и на базе Symbian OS;
- 5xxx - с цифры пять начинаются модели, приспособленные к экстремальным ситуациям в стильном противоударном корпусе. Работают на основе прошивки;

- 6xxx - это классические бизнес-модели (и не только), рассчитанные на широкий круг людей, которые в основном работают под управлением операционной системы Symbian, однако и имеют модели на основе прошивки;
- 7xxx - в английском языке есть слово Fashion (Мода), именно на это указывает первая цифра семь. Эти трубки могут работать на прошивке и на операционной Системе;
- 8xxx - дорогие имиджевые телефоны, работающие на основе прошивки;
- 9xxx - это коммуникаторы, как правило, с полноценной QWERTY-клавиатурой, работающие под управлением Symbian OS и рассчитанные скорее на корпоративных потребителей;
- буквы E и N - буквенные обозначения в моделях мобильных устройств появились недавно. Буква E обозначает принадлежность устройства к бизнес-классу, а буква N обозначает мультимедийную направленность устройства.

Данная классификация различает телефоны по классам и рассчитана, скажем так, на обычного покупателя.. Для разработчиков программного обеспечения все телефоны делятся на серии, причем бытующее мнение о том, что первая цифра в обозначениях телефонов указывает на серию, на самом деле не соответствует действительности. Так, к примеру, смартфон Nokia 3230 принадлежит к серии 60, а телефон 6170 - к серии 40. Имеется несколько серий.

- Серия 40 - телефоны, работающие на базе прошивки.
- Серия 60 - смартфоны и коммуникаторы, работающие на базе операционной системы Symbian OS.
- Серия 80 - коммуникаторы, работающие на базе операционной системы Symbian OS. Эта серия более не поддерживается, и все последующие устройства будут относиться к шестидесятой серии.
- Серия 90 - коммуникаторы, работающие на базе операционной системы Symbian OS. Эта серия более не поддерживается, и все последующие устройства будут относиться к шестидесятой серии.

Дополнительно все мобильные устройства от Nokia можно подразделить по разрешениям экранов. Присутствуют модели со следующими размерами экранов:

- 128 x 128 пикселей;
- 128 x 160 пикселей;
- 176 x 208 пикселей;
- 208 x 208 пикселей;
- 240 x 320 пикселей;
- 320 x 240 пикселей;
- 352 x 412 пикселей.

4.2.1. Сайт компании Nokia

Сайт компании, находящийся в Интернете по адресу <http://www.iorum.nokia.com>, предназначен специально для программистов (рис. 4.5). Это огромный ресурс, созданный специально для разработчиков и содержащий всевозможные SDK,

документацию, примеры исходного кода, статьи по разным тематикам. Для того чтобы стать полноправным членом клуба Nokia, необходимо обязательно зарегистрироваться на сайте этой компании. Более того, применение SDK от Nokia требует обязательной регистрации, иначе этот комплект разработчика проработает у вас на компьютере всего 14 дней.

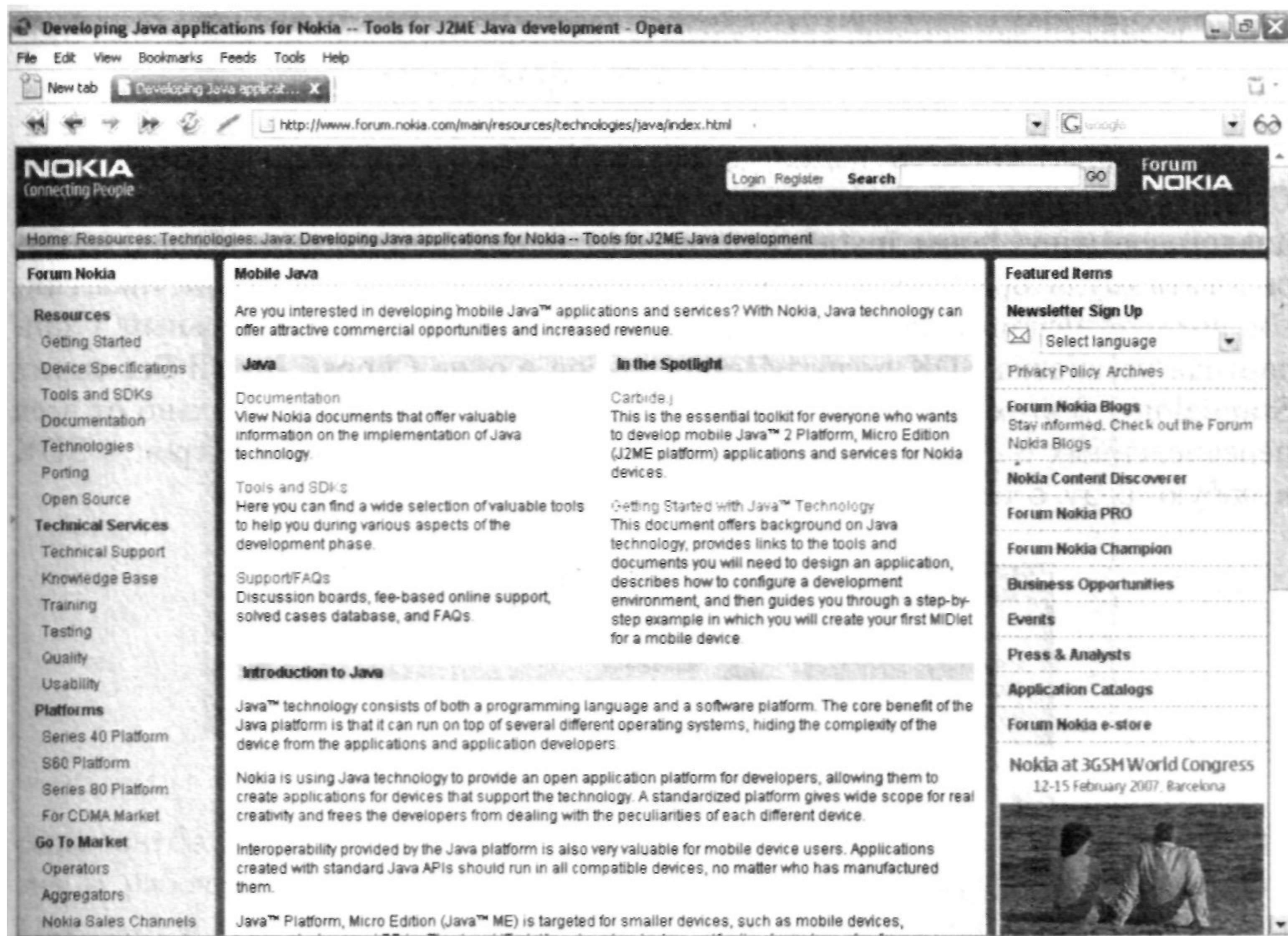


Рис. 4.5. Сайт компании Nokia

Для регистрации на сайте в верхней части интернет-страницы найдите ссылку **Registration**. Перейдя по этой ссылке, вы попадете на страницу регистрации. Заполняя форму, обязательно укажите рабочий адрес вашей электронной почты. По окончании регистрации на указанный адрес электронной почты придет ссылка, по которой вы должны подтвердить свое желание зарегистрироваться на сайте Nokia. После этого вы будете являться официально зарегистрированным пользователем.

4.2.2. Carbide.j

Компания Nokia имеет огромное количество SDK для различных моделей телефонов, на каком-то этапе даже практиковался выпуск SDK под один конкретный телефон. Впоследствии пакеты разработчика стали компоноваться в одну программу. За это время было выпущено множество всевозможных SDK с разными названиями. На данный момент последнее и полное SDK носит название

Carbide.j. Этот пакет разработчика включает в себя эмуляторы телефонов, которые способны представить любой телефон компании.

Примечание. Комплект Carbide также имеется и для языка программирования C++ с похожим названием Carbide.c. Этот пакет можно найти на сайте компании. Если вы интересуетесь программированием смартфонов и коммуникаторов на C++, то, возможно, вам поможет книга «Symbian OS. Программирование мобильных телефонов на C++ и Java 2 ME» издательства «ДМК-Пресс».

На компакт-диске в папке **\SDK\Nokia** находится установочный файл Carbide.j. Установка SDK стандартна, но на начальном этапе установки откроется диалоговое окно **Choose Install Set** (рис. 4.6). В этом окне вы можете сразу интегрировать эмуляторы SDK в один из инструментариев. Как видно из рисунка, список поддерживаемых сред программирования достаточно внушительный. Стандартная установка SDK подразумевает выбор в окне **Choose Install Set** опции **Standalone**. В этом случае вы сможете использовать Carbide.j отдельно от всех перечисленных инструментариев, а при желании в дальнейшем встроить SDK в любую среду, о чем мы обязательно поговорим в конце этой главы.



Рис. 4.6. Окно Choose Install Set

Кроме самого SDK, на ваш компьютер без спроса будет установлен пакет PC Suite, который используется для связи мобильного устройства Nokia с компьютером, а также еще несколько дополнительных программ. Запуск установленного SDK можно осуществить через команды **Пуск => Все программы => Carbide.j**. По выполнении этих команд откроется рабочее окно Carbide.j, представленное на рис. 4.7.

Рабочее окно Carbide.j имеет две вкладки: **Run** (на эту вкладку вы попадаете изначально) и **OTA Simulation**. Для запуска Java-программы необходимо в рабочем окне Carbide.j и вкладке **Run** (рис. 4.7), в поле **Application** прописать путь к JAR-файлу запускаемой программы. Затем в списке **Select Device** выбрать



Рис. 4.7. Рабочее окно Carbide.j

один или более эмуляторов посредством флажка и нажать на кнопку **Emulate**. Далее в рабочем окне Carbide.j перейдите на вторую вкладку **OTA Simulation** (рис. 4.8) и нажмите там кнопку **Install, run and remove from URL** или **Run previously installed application**. После выполнения этой команды произойдет запуск выбранного ранее эмулятора (рис. 4.9). В качестве тестируемой программы был выбран проект Demo, созданный нами в предыдущей главе.

После установки Carbide.j на компьютер вам необходимо в течение 14 дней зарегистрировать SDK через Интернет. Для этого достаточно подключиться к Интернету, открыть Carbide.j и выполнить команды **Help => Registration**. Затем следуйте инструкциям по регистрации продукта. Единственное условие при регистрации - это то, что вы должны уже быть зарегистрированным пользователем сайта www.forum.nokia.com.

4.3. BenQ-Siemens

В качестве программного пакета для работы с Java 2 ME в BenQ-Siemens используется программа Mobility Toolkit for Java Development (МТК). Эта программа является своего рода программной оболочкой, в которую необходимо интегрировать эмуляторы отдельных марок телефонов этой компании. На компакт-диске

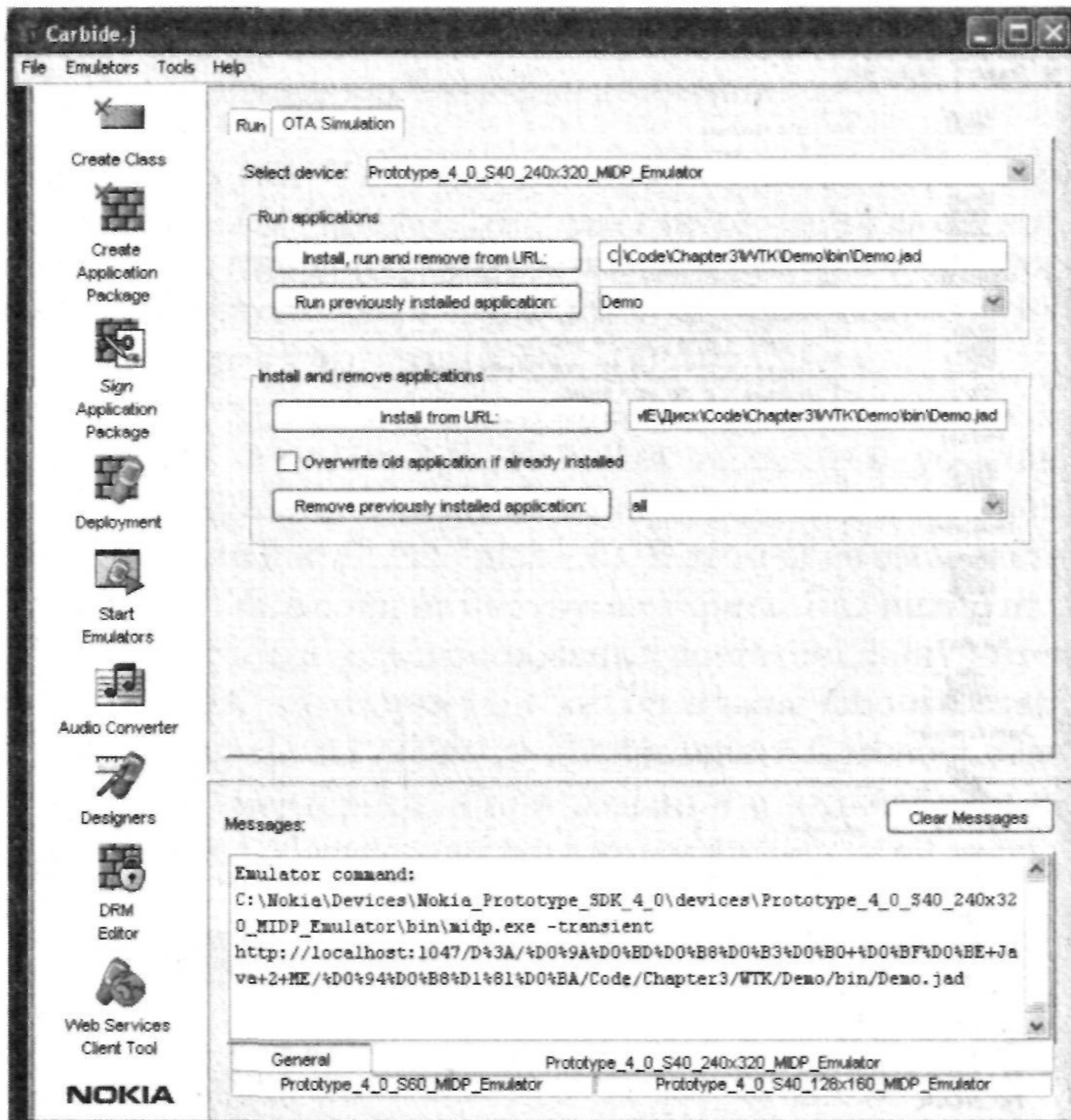


Рис. 4.8. Вкладка OTA Simulation



Рис. 4.9. Запуск программы на эмуляторе

в папке \BenQ-Siemens находится файл smtk_3_2 - это и есть Mobility Toolkit for Java Development. Все остальные файлы в этой папке представляют эмулятор той или иной модели телефона. К сожалению, компакт-диск заполнен под завязку, поэтому некоторые эмуляторы BenQ-Siemens просто не поместились.

После установки МТК и всех или части эмуляторов у вас на рабочем столе появится ярлык МТК Control Center. Запустив утилиту МТК, вы увидите небольшое рабочее окно программы, изображенное на рис. 4.10.

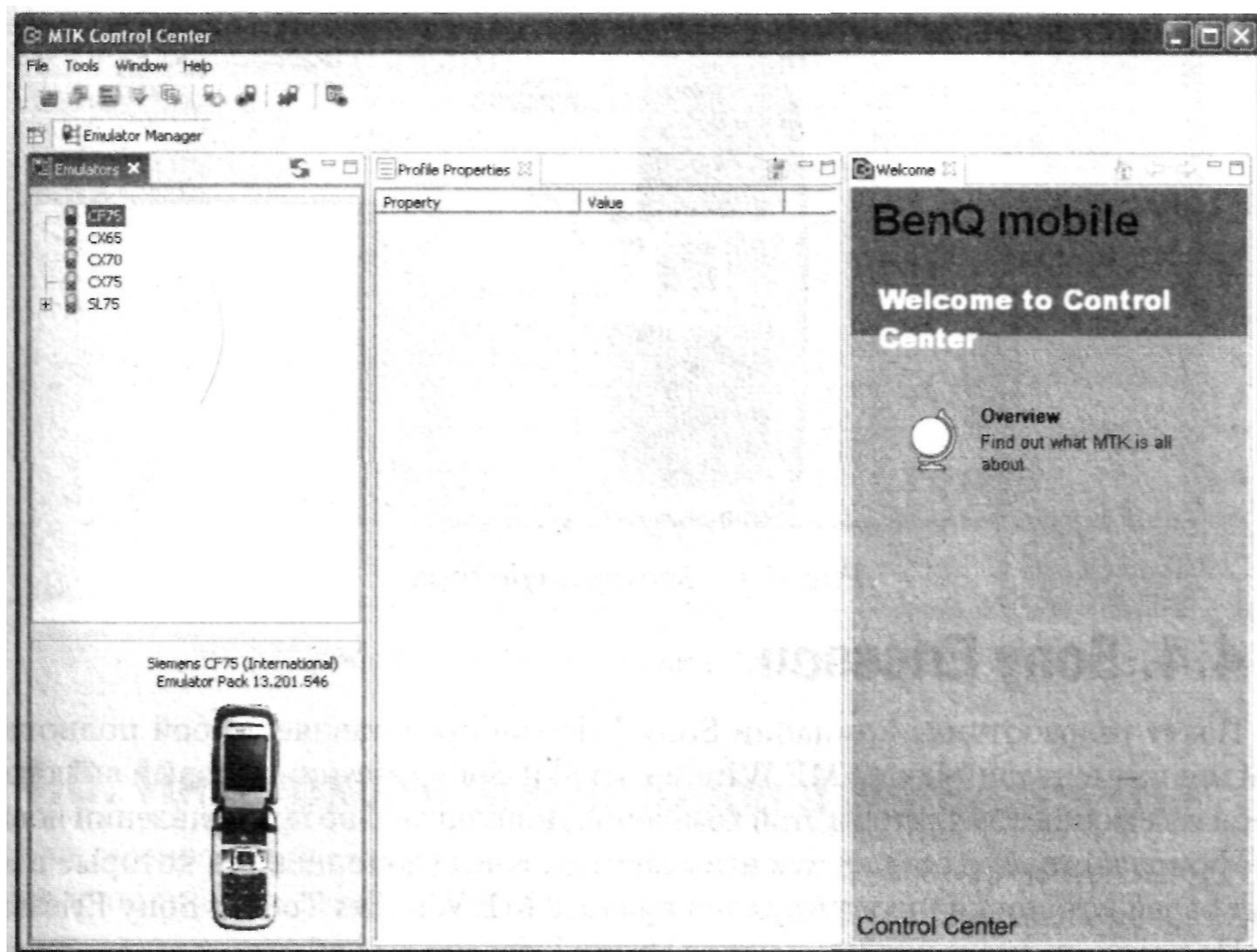


Рис. 4.10. Рабочее окно MTK Control Center

Примечание. При первом старте МТК рабочее окно программы выглядит несколько иначе, чем показано на рис. 4.10. Чтобы развернуть рабочее окно программы на весь экран, выполните команды **Windows => Show View => Emulator**.

Для того чтобы запустить Java-программу с помощью МТК, щелкните два раза левой кнопкой мыши на названии эмулятора на вкладке **Emulator**, которая располагается с левой стороны главного окна МТК (рис. 4.10). После того как на экране появится избранный эмулятор, щелкните прямо на нем правой кнопкой мыши и в появившемся контекстном меню выберите команды **Phone Commands => Start Application** (рис. 4.11). Откроется диалоговое окно, в котором необходимо указать путь до JAR-файла. Сама программа должна обязательно располагаться на диске «С». Заккрыть эмулятор можно в том же контекстном меню, выполнив команду **Exit (Alt+F4)**.

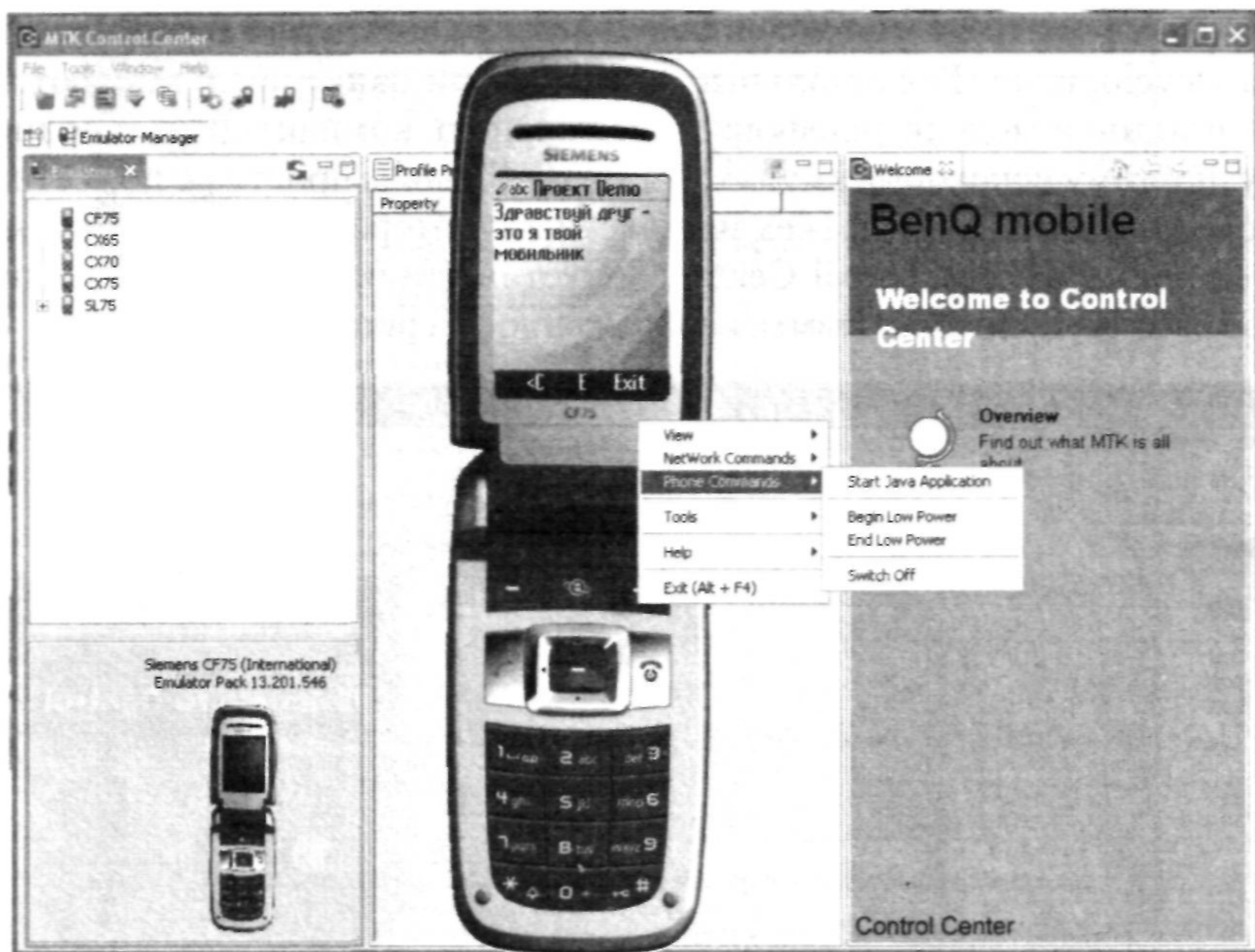


Рис. 4.11. Запуск эмулятора

4.4. Sony Ericsson

Пакет разработчика компании Sony Ericsson представляет собой полноценный инструментальный Java 2 ME Wireless Toolkit Sony Ericsson, который включает в себя телефонные эмуляторы этой компании. Дополнительно при появлении новых телефонов компания создает так называемые адоны (дополнения), которые в момент своей установки интегрируются в Java 2 ME Wireless Toolkit Sony Ericsson.

На компакт-диске к книге в папке **\Sony Ericsson** вы найдете два дополнения с различными эмуляторами и Java 2 ME Wireless Toolkit Sony Ericsson. Установка пакета Sony Ericsson J2ME SDK происходит стандартным образом, но в качестве каталога необходимо обязательно использовать директорию, заданную Sony Ericsson J2ME SDK по умолчанию.

Как всегда, на сайте компании в Интернете по адресу <http://developer.sonyericsson.com> можно найти дополнительную информацию и обновить свои программные средства (рис. 4.12).

4.5. Samsung

В папке **\Samsung** на компакт-диске найдите файл **SJSDKv3.0** - это программа установки. Инсталлируйте программу Samsung JSDK на свой компьютер, а затем откройте установленную программу. Откроется основное окно **Samsung JSDK**, изображенное на рис. 4.13. Для запуска Java-программы выполните команды **File => Import MIDlet**.

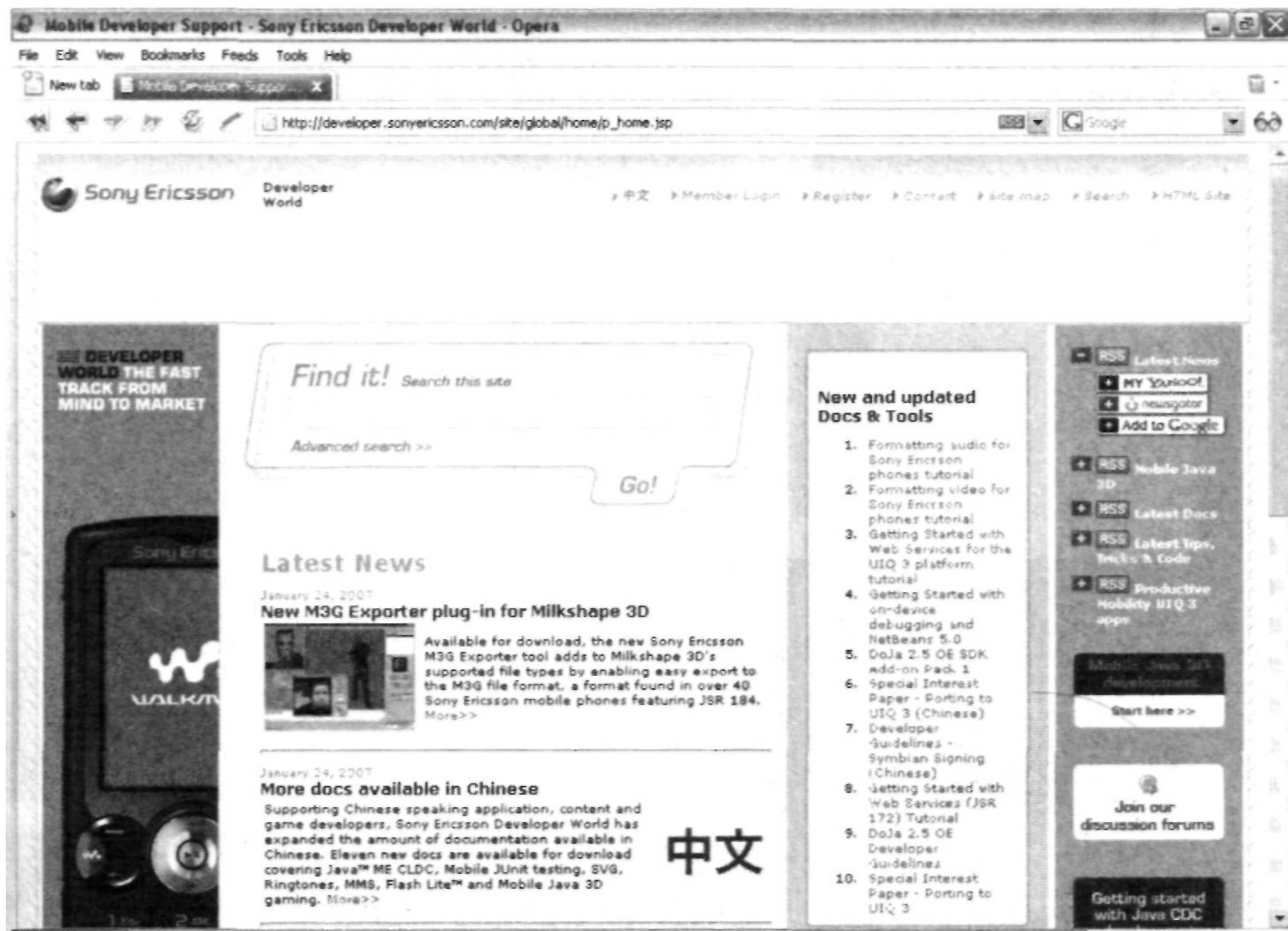


Рис. 4.12. Сайт компании Sony Ericsson

4.6. Интеграция эмуляторов в Net Beans IDE

Для тестирования создаваемой программы в инструментарии Net Beans IDE можно использовать встроенные в программу эмуляторы. Но куда лучше применять эмуляторы непосредственно производителей телефонов, поэтому компания Sun Microsystems предусмотрела возможность встраивания в Net Beans IDE сторонних эмуляторов. Давайте сейчас на конкретном примере в пошаговом режиме рассмотрим механизм интеграции SDK в инструментарий NetBeans IDE.

1. Итак, откройте NetBeans IDE с любым проектом. Затем выполните команды меню **Edit => Processor Blocks => Add Configurations to Project**. Либо щелкните правой кнопкой мыши в окне текстового редактора. В появившемся контекстном меню выберите команды **Processor Blocks => Add Configurations to Project**. В ответ на эти действия откроются сразу два диалоговых окна, одно поверх другого. В маленьком окне **Add Configuration** в поле **New Configuration Name** задайте имя, которое будет характеризовать добавляемые эмуляторы (рис. 4.14). В качестве примера я выбрал название Sony_Ericsson, и именно эмуляторы этой компании мы будем интегрировать в NetBeans IDE, но механизм для всех одинаков. Имя конфигурации, задаваемое в поле **New Configuration Name**, не должно содержать пробелов и кириллицы. Затем нажмите кнопку **OK**.

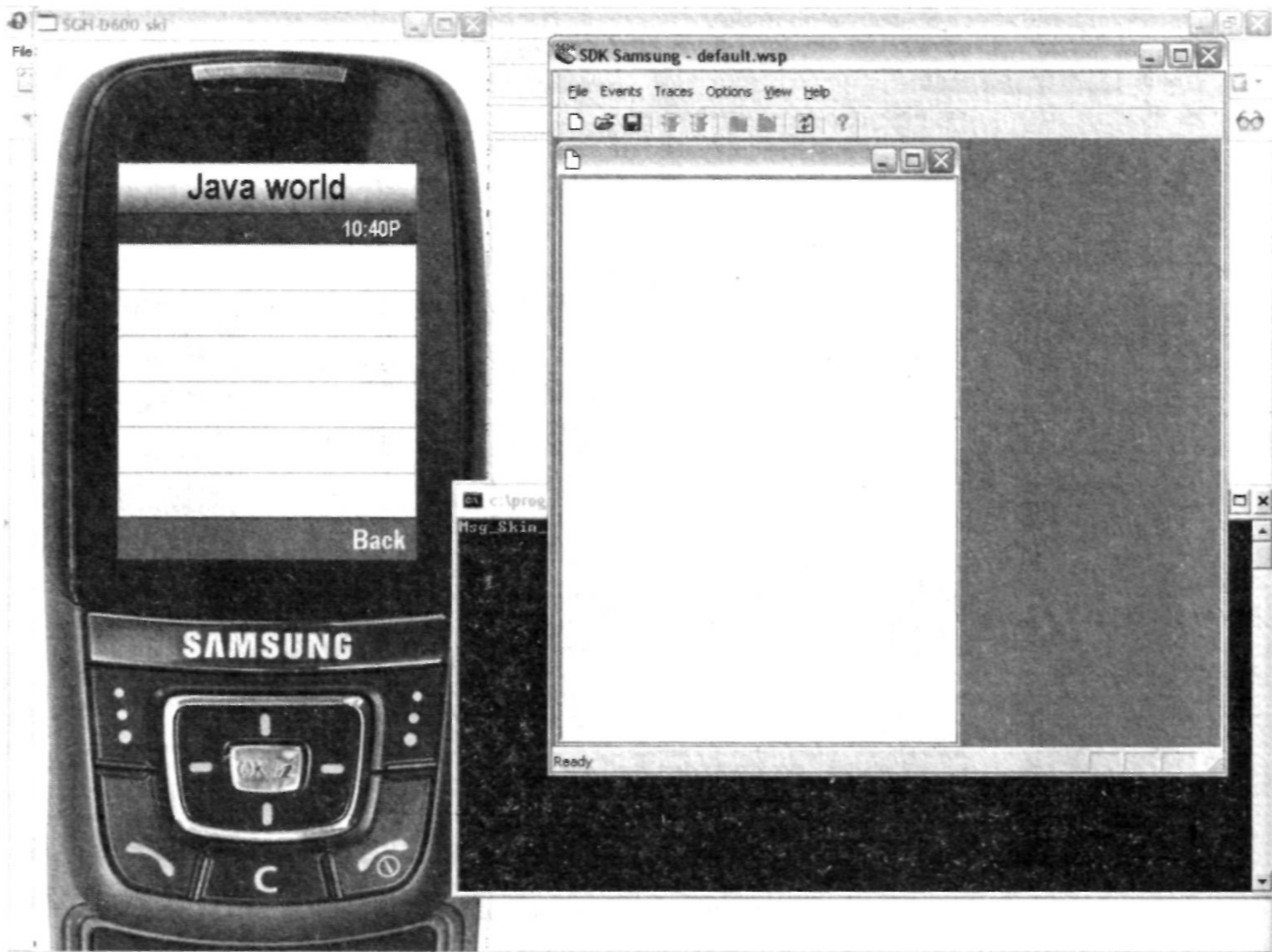


Рис. 4.13. Программа Samsung JSDK

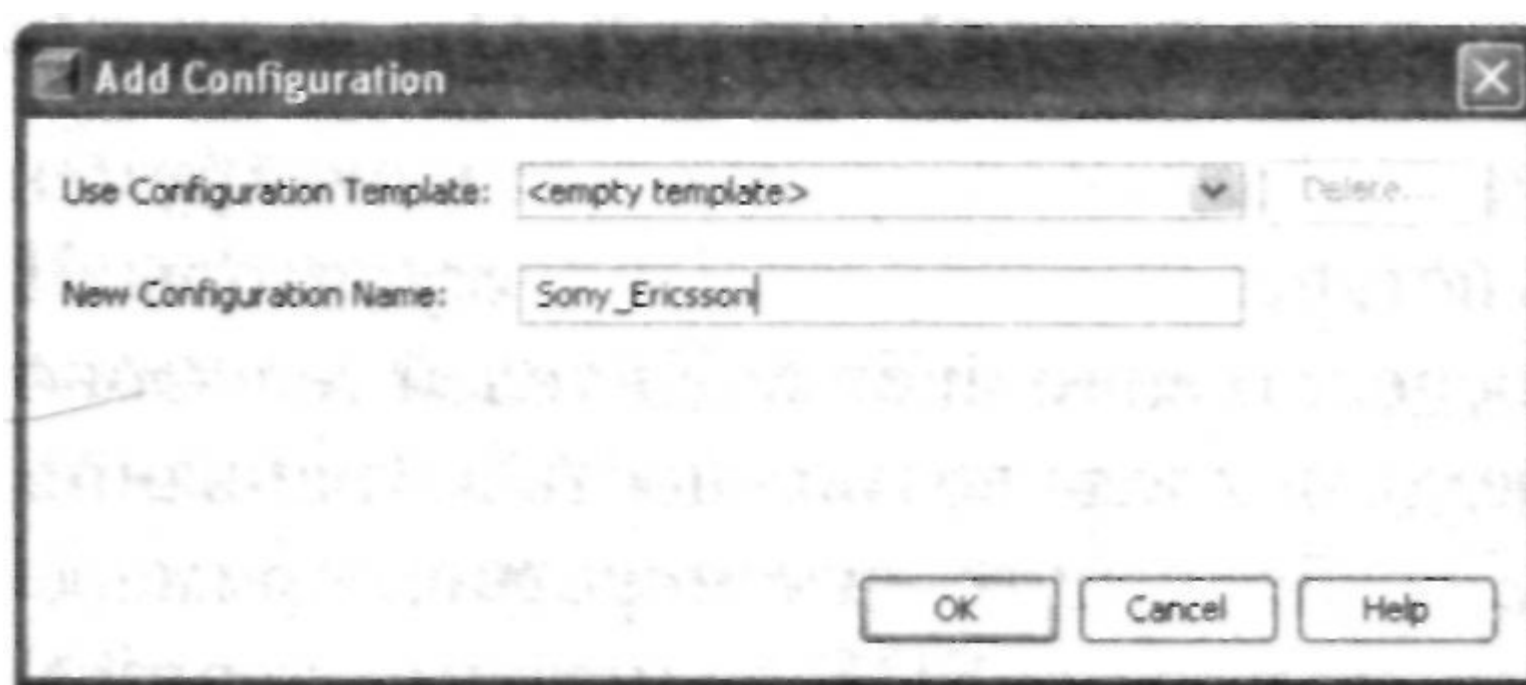


Рис. 4.14. Окно Add Configuration

2. Маленькое окно закроется, и вам станут доступны элементы управления в большом окне с названием текущего проекта (рис. 4.15). Снимите в этом окне флажок **Use value from «Default Configuration»** и нажмите кнопку **Manage Emulators**. Кстати, вы можете и вовсе не использовать окно Add Configuration для задания конфигурации. Достаточно не задавать имя конфигурации в этом окне и нажать кнопку **Cancel**. В этом случае на всех последующих шагах имя конфигурации станет по умолчанию, а название эмуляторов будет взято NetBeans IDE от названий встраиваемых эмуляторов.
3. Откроется новое окно **Java Platform Manager**, представленное на рис. 4.16. В центре этого окна располагается набор вкладок. Вкладка с названием **Device**

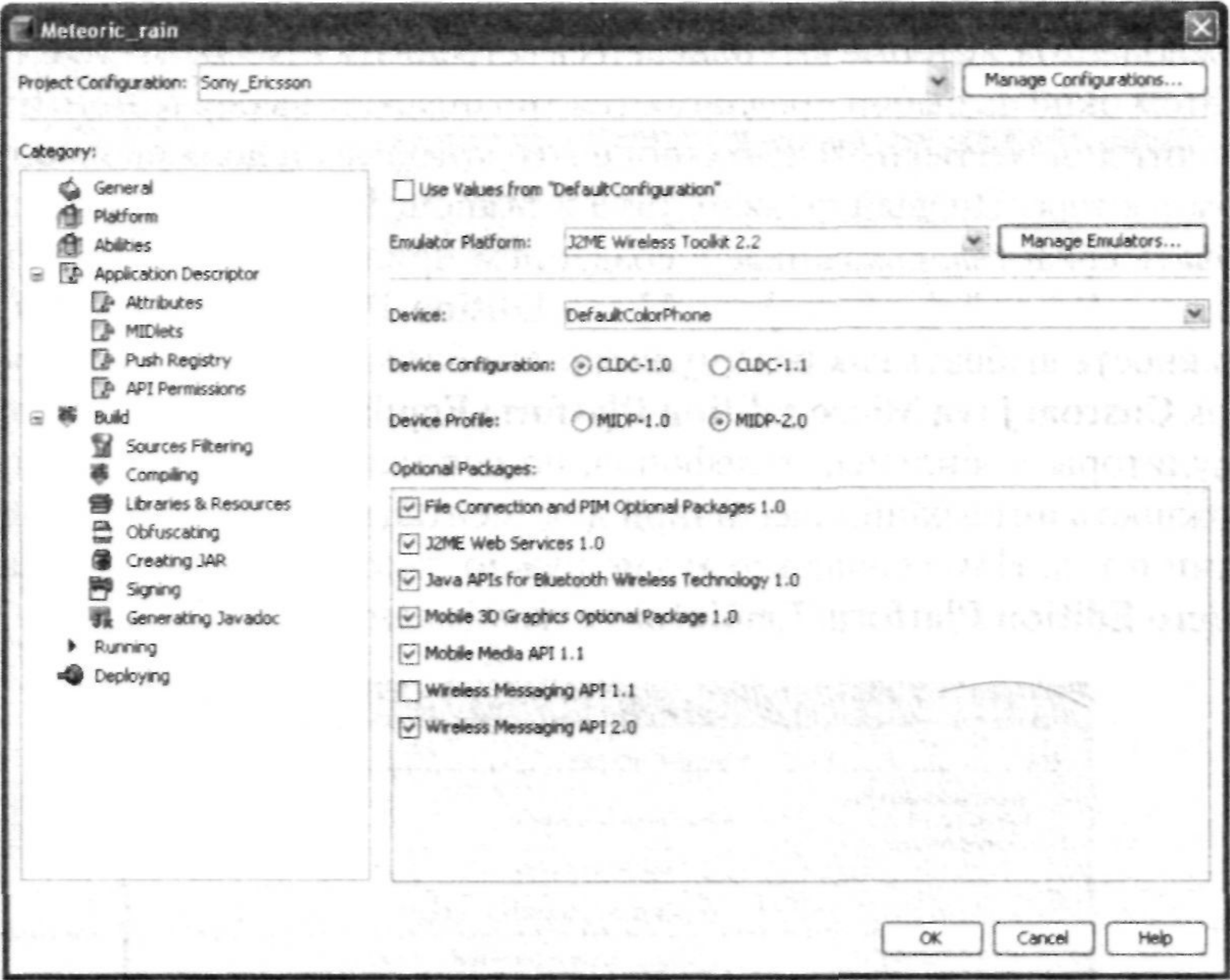


Рис. 4.15. Окно конфигурации

указывает на то, какие эмуляторы на данный момент интегрированы NetBeans IDE. Чтобы добавить дополнительные эмуляторы, нажмите в окне **Java Platform Manager** кнопку **Add platform**. Откроется следующее окно **Add Java Platform**.



Рис. 4.16. Окно Java Platform Manager

4. Новое окно **Add Java Platform - Select platform types** позволяет задать тип компонентов, которые вы собираетесь встраивать в NetBeans IDE (рис. 4.17). В этом окне на выбор предлагаются три флажка, выбор одного из них обозначит для NetBeans IDE компоненты, которые он должен искать у вас на компьютере. Первый флажок **Java 2 Standard Edition** позволяет интегрировать средства, связанные с созданием приложений для компьютерных систем. Вторым флажок **Java Micro Edition Platform Emulators** дает возможность выбрать как раз эмуляторы мобильных телефонов. Третий флажок **Custom Java Micro Edition Platform Emulators** также позволяет найти эмуляторы мобильных телефонов, но дополнительно предоставляет возможность интеграции различной документации, исходных кодов с примерами и т. д. Нам сейчас все это не нужно, поэтому избираем флажок **Java Micro Edition Platform Emulators** и нажимаем кнопку **Next**.

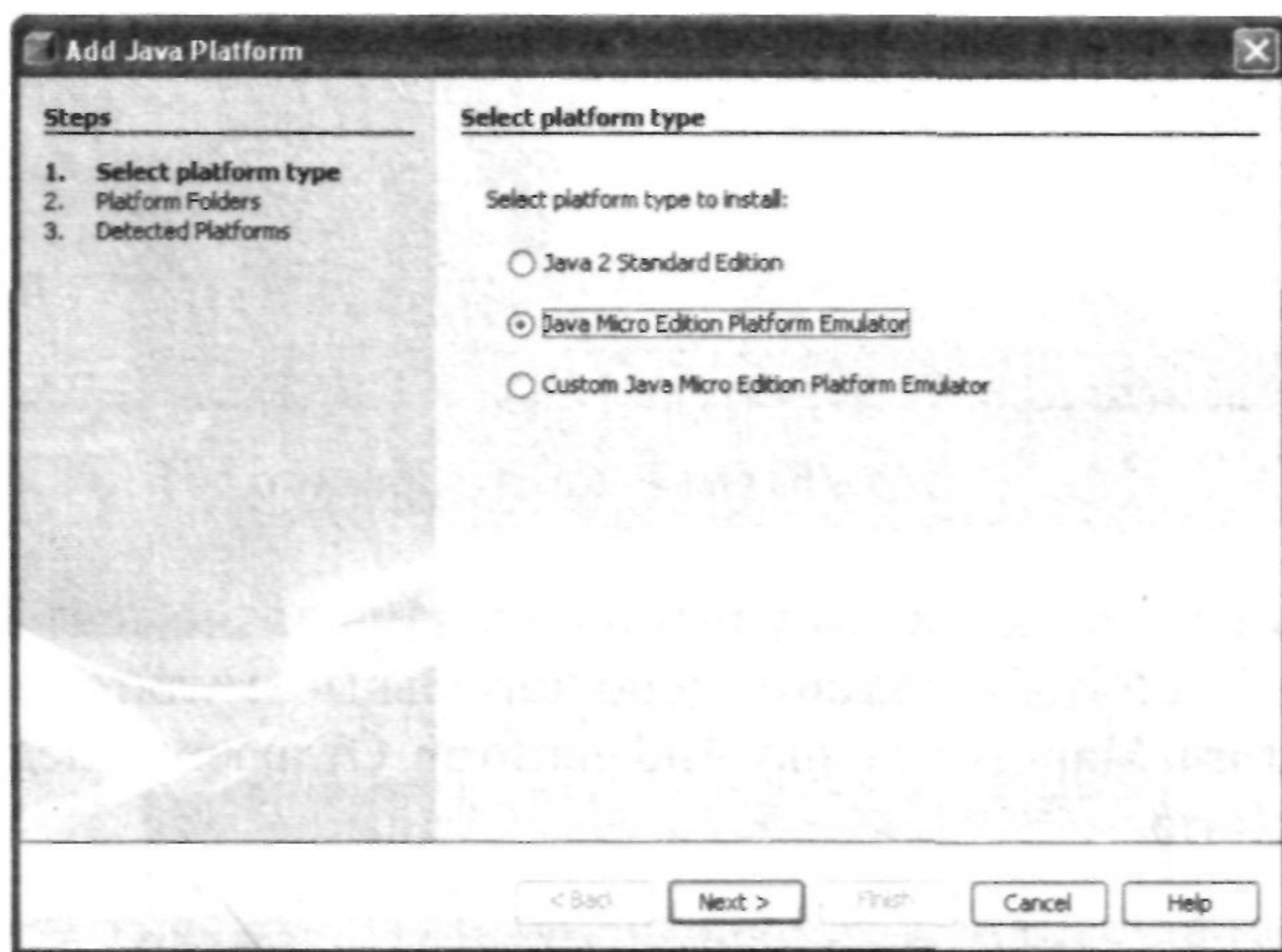


Рис. 4.17. Окно Add Java Platform - Select platform types

5. В этот момент инструмент NetBeans IDE в автоматическом режиме найдет все установленные на компьютере эмуляторы и представит их в новом окне **Add Java Platform - Platform Folders** (рис. 4.18). Поскольку мы сейчас интегрируем в инструмент исключительно эмуляторы компании Sony Ericsson, то в поле **Selected Platform to Detected** избираем эмуляторы этой компании с помощью выбора флажка напротив одноименного названия. Для продолжения нажимаем кнопку **Next**.
6. В следующем окне **Add Java Platform - Detected Platform** (рис. 4.19) инструмент обозначит все эмуляторы компании Sony Ericsson, найденные им. В частности, в текстовом поле **Devices** будет присутствовать модельное перечисление телефонов, встраиваемых в NetBeans IDE. Нажмите в этом окне кнопку **Finish**.



7. После всех этих операций вы вернетесь в окно **Java Platform Manager**. В списке **Platforms** этого окна вы сможете обнаружить добавленные в инструментарий эмуляторы. Нажмите в этом окне кнопку **Close** для возврата к окну конфигурации проекта (рис. 4.20), с которого начиналась интеграция эмуляторов в инструментарий.

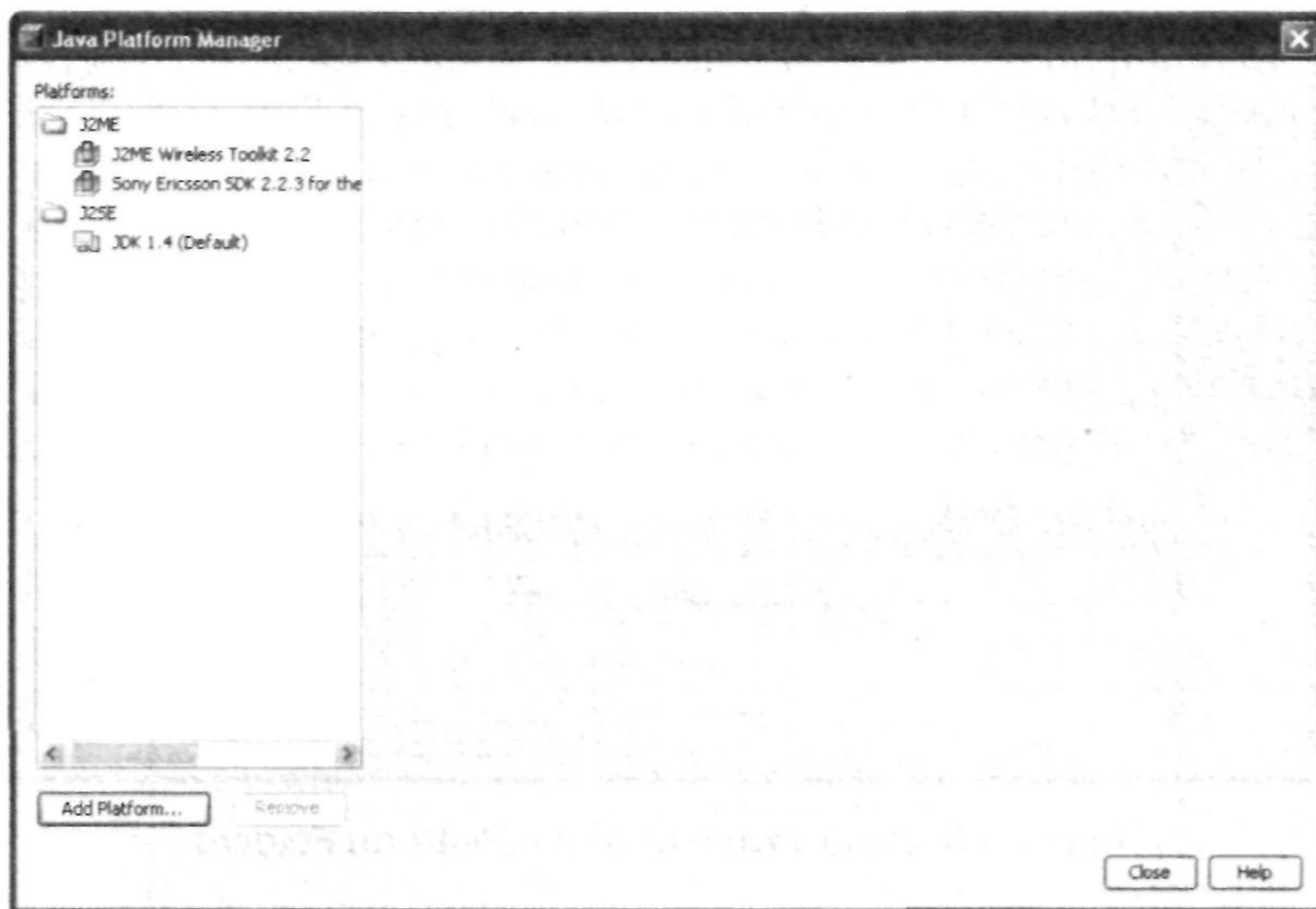


Рис. 4.20. Окно конфигурации проекта

8. В окне конфигурации в списке **Project Configuration** выберите конфигурацию **Sony_Ericsson**, а в списке **Devices** можно подобрать необходимый эмулятор (рис. 4.21). Нажав после этого в этом окне кнопку **ОК**, вы закроете данное окно и вернетесь в рабочее окно инструментария. На этом интеграция эмуляторов в Net Beans IDE завершена.

Возвратившись в рабочее окно инструментария, на панели инструментов в списке **Devices** появится только что добавленная конфигурация (рис. 4.22). В отдельно взятый момент будет использоваться эмулятор этой компании, которой был выбран в окне конфигурации проекта (рис. 4.21) в списке **Devices**. Для смены эмулятора откройте это окно (**Edit => Processor Blocks => Add Configurations to Project**) и выберите в списке **Devices** другой эмулятор.

Еще одно существенное изменение заключается в том, что в рабочем каталоге проекта появилась дополнительная папка с названием добавленной конфигурации. В итоге теперь у вас для каждой отдельной конфигурации будет присутствовать отдельный проект и соответственно отдельный установочный пакет! На вкладке **Files** рабочего окна инструментария это хорошо видно (рис. 4.23).

В главах 3 и 4 вы изучили установку, настройку, работу с интегрированными средствами программирования приложений и инструментальными средствами производителей телефонов. В следующих главах мы перейдем непосредственно к работе с кодом, где будет представлено для изучения большое количество интерфейсов, классов, методов из состава платформы Java 2 ME.

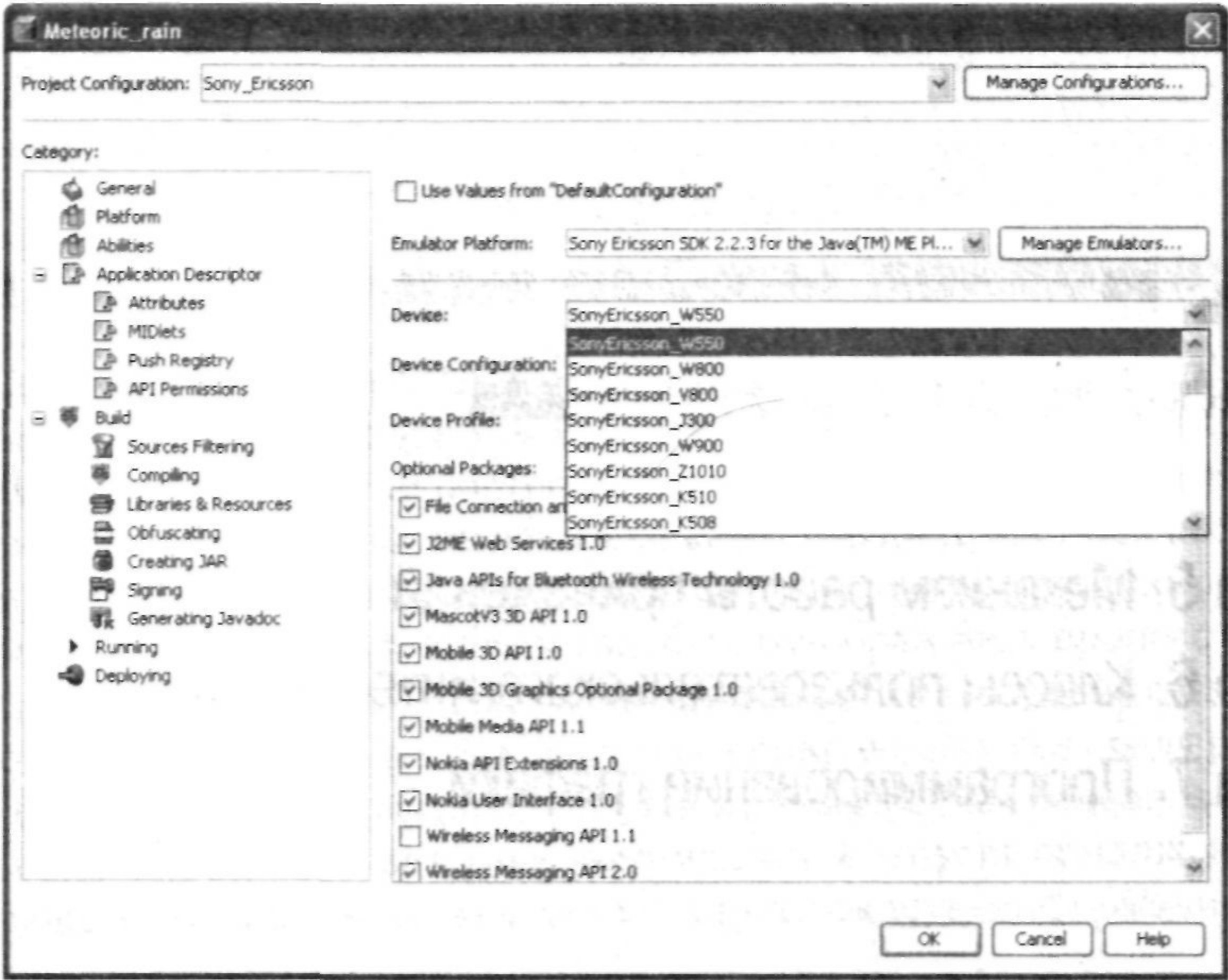


Рис. 4.21. Окно конфигурации



Рис. 4.22. Список Devices



Рис. 4.23. Вкладка Files



Часть II

Разработка программ

Глава 5. Механизм работы приложений

Глава 6. Классы пользовательского интерфейса

Глава 7. Программирование графики

<http://palata-x.narod.ru>



<http://palata-x.narod.ru>

Глава 5. Механизм работы приложений

С этой главы непосредственно начинается описание работы с кодом. Специфика приложений Java 2 ME несколько своеобразная, но совсем не сложная. Достаточно разобраться в общей модели построения программ, и все сразу станет понятно.

В предыдущих двух главах вашему вниманию были представлены две среды программирования мобильных приложений и большое количество разнообразных телефонных эмуляторов. Полностью был разобран весь процесс установки этих средств, а также режимы создания, компиляции кода и просмотр получившегося приложения на эмуляторе. Выберите себе понравившуюся среду разработки или работайте с теми средствами, к которым привыкли. Мы больше не будем отвлекаться на процесс написания кода, компиляции и запуска приложения на эмуляторе. Предшествующие две главы дали исчерпывающую информацию по этому поводу.

5.1. Мидлет

Приложение, написанное для мобильного телефона, может состоять из различного количества классов. Одни классы отвечают за загрузку ресурсов, другие классы - за обработку данных, третьи выполняют еще какие-то дополнительные функции. Как программист вы вправе выбирать любую удобную для вас модель построения программы. В итоге все созданные вами классы, объединенные в одно целое, будут составлять одну программу или приложение. Все приложения, сформированные для работы в среде Java мобильных телефонов, носят название мидлет.

Мидлет - это программа, написанная для мобильного телефона с использованием платформы Java 2 ME. Определять количество классов программы - привилегия программиста, но среди всех классов одной программы существует один основной класс, с которого начинается работа всей программы. Это основной класс мидлета, сердце приложения, он наследуется от класса `javax.microedition.midlet.MIDlet`. В этом классе описывается код, отвечающий за управление процессом создания интерфейса пользователя, объявление набора данных, необходимых для работы всего приложения, создаются объекты имеющихся классов, и, что самое главное, он является отправной точкой в работе приложения. Такой класс в Java 2 ME носит название *основной класс мидлета*.

Рабочие функции, выполняемые этим классом, практически идентичны методу `main()`. Помните запись, с которой начинался рабочий процесс приложений, написанных на Java 2 SE:

```
public static void main(String[] args)
```


На мобильных устройствах аналогичные действия возложены на подкласс класса MIDlet, производящий управление рабочим процессом всего приложения. В дополнение к основному классу может создаваться ряд классов, необходимых для реализации поставленной перед вами задачи. Также имеется возможность собирать несколько мидлетов в один архив. Подобная комплектация программ, или мидлетов, помещенных в один JAR-файл, носит название MIDlet suite (набор мидлетов).

В предыдущих главах в качестве примера приводился проект Demo, состоящий из одного класса с названием HelloMIDlet, который в этом приложении также играет роль основного класса мидлета. Посмотрите на листинг 5.1, где приведен исходный код класса HelloMIDlet проекта Demo. На компакт-диске листинг находится в папке `\Code\Chapter\Listing5_1\src`.

```
/**
Листинг 5.1
Класс HelloMIDlet.Java
*/

// импорт пакетов
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
// создается класс HelloMIDlet
public class HelloMIDlet extends MIDlet implements
CommandListener
{
// команда выхода из программы
private Command exitCommand;
// дисплей телефона
private Display mydisplay;
// конструктор класса HelloMIDlet
public HelloMIDlet()
{
mydisplay = Display.getDisplay(this);
// выход из программы
exitCommand = new Command("Выход", Command.SCREEN, 2);
}
// входная точка приложения
public void startApp()
{
// объект класса TextBox
TextBox t = new TextBox("HelloMIDlet", "Текст", 256, 0);
// добавляет команду выхода
t.addCommand(exitCommand);
// устанавливает обработчик событий для команды выхода
t.setCommandListener(this);
}
```

```
// отражает текущий экран телефона
mydisplay.setCurrent(t) ;

// пауза в работе приложения
public void pauseApp() {}
// выход из программы
public void destroyApp(boolean unconditional) {}
// обработчик событий для команд приложения
public void commandAction(Command c, Displayable s)
{
    // обработка команды выход
    if (c == exitCommand)
    {
        // освобождает ресурсы и выгружает из памяти мидлет
        destroyApp(false) ;
        notifyDestroyed() ;
    }
}
}
```

Первое, что бросается в глаза при детальном рассмотрении кода мидлета, - это три метода: `startApp()`, `pauseApp()` и `destroyApp()`. Программисты, которые знакомы с апплетами, найдут явное сходство в структуре мидлета и апплета, даже названия в некоторой степени схожи. В апплете также имеются подобные методы: `start()`, `stop()` и `destroy()`. Можно сказать, что основной класс мидлета - это некий симбиоз апплета и метода `main()`, играющего основную и управляющую роль в приложении для мобильных устройств. Создавая свои классы и реализуя их код, вы должны возложить основные рабочие функции на основной класс мидлета. На рис. 5.1 показан принцип работы мобильных приложений.

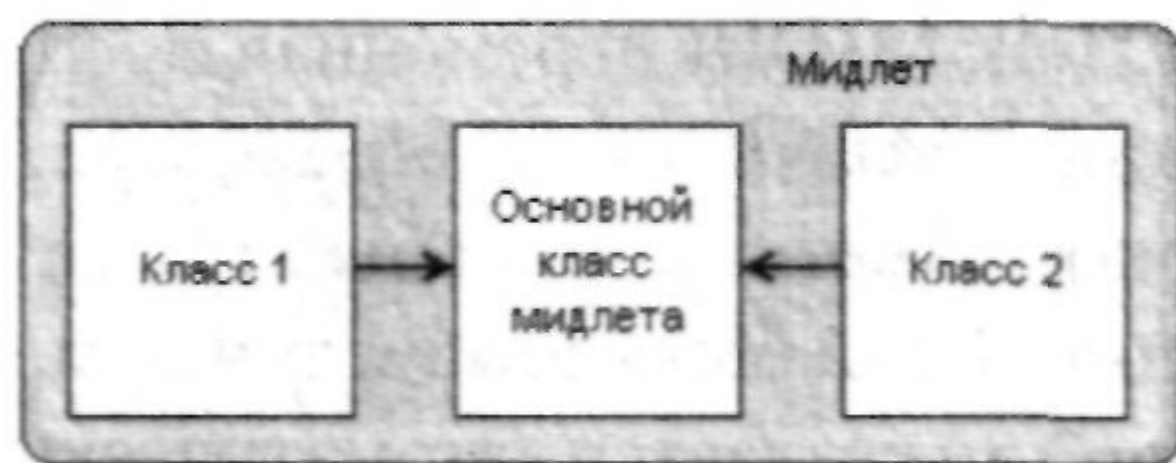


Рис. 5.1. Принцип работы приложения

Сейчас мы вкратце разберем код примера `HelloMIDlet` проекта `Demo`, а потом перейдем к более детальному анализу схемы работы приложений в `Java 2 ME`. Первые две строки кода из листинга 5.1 - это *библиотеки импорта*:

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

В этих строках происходит импорт двух пакетов платформы `Java 2 ME`. Первый пакет `javax.microedition.midlet.*` содержит класс `MIDlet`, с помощью которого

происходит связь с MIDP. Каждый код, создающий основной класс мидлета, обязан импортировать этот пакет. Второй пакет содержит классы, позволяющие создавать пользовательский интерфейс приложения. В следующей главе будут подробно рассмотрены все классы пользовательского интерфейса. Импорт двух пакетов дает возможность задействовать все имеющиеся в этих пакетах интерфейсы, классы, методы и константы, необходимые для работы всего создаваемого приложения. Следующая строка кода:

```
public class HelloMIDlet extends MIDlet implements  
CommandListener
```

создает основной класс мидлета, наследуемый от суперкласса MIDlet, реализующий при этом методы `startApp()`, `pauseApp()` и `destroyApp()`. Также задействуется интерфейс `CommandListener`, необходимый для обработки событий, происходящих в приложении при помощи метода `commandAction()`. Далее идут две строки, создающие объекты классов `Command` и `Display`:

```
private Command exitCommand;  
private Display mydisplay;
```

С помощью переменной `exitCommand` осуществляется выход из приложения посредством *метода* `commandAction()`. Переменная `mydisplay` будет представлять дисплей телефона при помощи абстрактного класса `Display`, играющего роль диспетчера телефонных экранов. В конструкторе класса `HelloMIDlet` инициализируются объекты `exitCommand` и `mydisplay`.

```
public HelloMIDlet()  
{  
mydisplay = Display.getDisplay(this);  
exitCommand = new Command("Выход", Command.EXIT, 2);  
}
```

В конструкторе класса `HelloMIDlet` переменная `mydisplay` получает ссылку на объект `Display` при помощи *метода* `getDisplay()`. Давайте разберем эти действия подробно. Вы, наверное, заметили, что объект `Display` явно не создается при помощи ключевого слова `new`. Если быть совсем точным, вы просто не можете этого сделать. Объект класса `Display` создается автоматически с помощью сервисов телефона для каждого мидлета. Класс `Display` играет роль диспетчера телефонных экранов, на которые проецируется в итоге все то, что вы будете видеть на дисплее телефона. В телефоне существует всего один основной экран, и поэтому в отдельный промежуток времени может быть отражен только один экран, за который отвечает класс `Display`. С помощью метода `getDisplay()` основной класс мидлета получает ссылку на класс `Display` и содержит ее в переменной `mydisplay`. Впоследствии, для того чтобы отразить содержимое текущего дисплея, необходимо воспользоваться *методом* `setCurrent()`, например следующим образом:

```
mydisplay.setCurrent();
```


Вторая и последняя строка кода в конструкторе класса `HelloMIDlet()` создает экземпляр класса `Command` и инициализирует созданный ранее объект `exit Command`. Класс `Command` создает набор информации, или набор команд, которые можно отобразить на экране телефона для обработки событий, полученных от пользователя. На основе определенного набора команд с помощью интерфейса `CommandListener` происходит обработка фактических действий пользователя. Посмотрите на рис. 5.2, где на дисплее показана команда **Выход**, расположенная под экранной клавишей телефона.

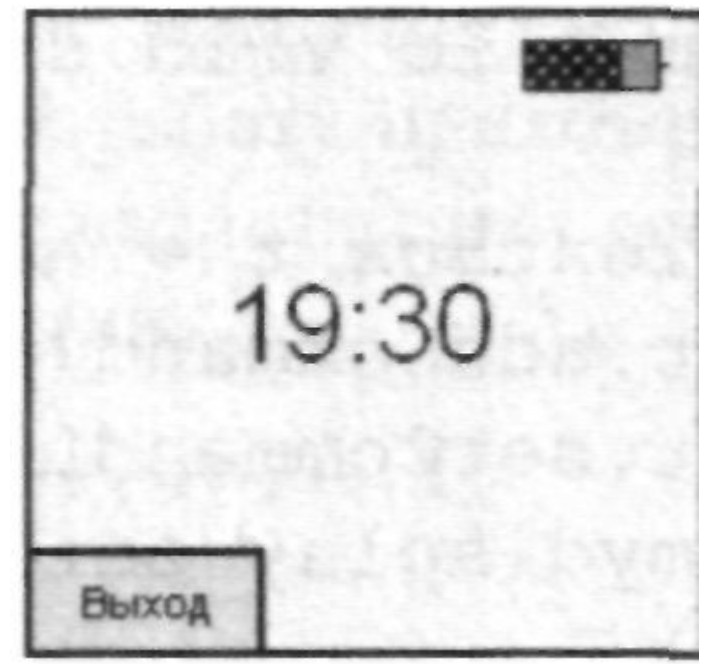


Рис. 5.2. Команда выхода из программы

Такая схема отображения различных команд реализована на всех без исключения мобильных телефонах. Единственное, что может изменяться, - это способ отображения этих команд. У некоторых производителей мобильных телефонов команды располагаются под экранными клавишами, а у других ряд команд формируется в виде меню.

Класс `Command` имеет два конструктора с тремя и четырьмя параметрами. В примере был использован конструктор из трех параметров, посмотрим, как выглядит прототип конструктора класса `Command`:

```
public Command(String label,int commandType,int priority)
```

Параметры конструктора класса `Command`:

- `label` - это переменная типа `String`, содержащая метку в виде простого текста. В примере использовалась команда **Выход**. Эта строка текста может находиться под кнопками дисплея либо в виде элемента меню. Обычно команда - это короткое слово. Если необходимо использовать длинное слово, то нужно вызвать конструктор класса `Command` с четырьмя параметрами;
- `commandType` - тип команды, соответствующий выбранным действиям. Имеются команды `BACK`, `CANCEL`, `EXIT`, `HELP`, `ITEM`, `SCREEN`, `STOP`:
 - `BACK` - возврат к предыдущему экрану;
 - `CANCEL` - отмена произведенных действий;
 - `EXIT` - выход из приложения;
 - `HELP` - вызов справки;
 - `ITEM` - обычно используется для работы с классом `Choice` и производит выбор элемента из группы элементов;
 - `SCREEN` - представление нового экрана;
 - `STOP` - остановка выполняемых действий;
- `priority` - это приоритет, назначенный для данной команды относительно других команд, имеющихся на дисплее. Приоритет задается целым числом, где более низкое число указывает на более высокое значение.

За конструктором класса `HelloMIDlet()` следует ключевой метод основного класса мидлета `startApp()`.

```
public void startApp()
{
    TextBox t = new TextBox("HelloMIDlet", "Текст", 256, 0);
    t.addCommand(exitCommand);
    t.setCommandListener(this);
    mydisplay.setCurrent(t);
}
```

Метод `startApp()` вводит приложение в активное состояние и является своего рода входной точкой для всей программы. В рассматриваемом примере для вывода текста на экран используется класс `TextBox`. Чтобы разобраться, как это происходит, необходимо рассмотреть конструктор класса *TextBox*:

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

Параметры конструктора класса `TextBox`:

- `title` - это титл или основное название, размещенное в верхней части дисплея;
- `text` - текст, выводимый непосредственно на экран. Текст, выводимый на экран, можно редактировать и даже не выводить вовсе, выставив значение этого параметра в значение `null`;
- `maxSize` - максимальное количество символов выводимого на экран текста;
- `constraints` - с помощью этого параметра можно производить специальные ограничения, но пока рекомендую выставить параметр в 0.

То есть, создав объект класса `TextBox`, вы задаете область экрана, в которой существует свое название, и количество выводимых символов. В следующих двух строках кода:

```
t.addCommand(exitCommand);
t.setCommandListener(this);
```

сначала добавляется команда **Выход** к текущему экрану телефона, представленному классом `TextBox`, а затем с помощью метода `setCommandListener()` устанавливается обработка событий для этой команды **Выход**, используя метод интерфейса `CommandListener`. Самая последняя строка кода:

```
mydisplay.setCurrent(t);
```

отражает текущую информацию на экране. Этот метод подобен кнопке **Обновить** в Internet Explorer. Вызывая каждый раз метод `setCurrent()`, вы будете обновлять информацию на дисплее телефона. Следующий метод в примере:

```
public void pauseApp()
```

Тело этого метода пока отсутствует, но с его помощью можно поместить мидлет в состояние паузы и освободить часть используемых ресурсов приложения. Далее

идет метод `destroyApp()`, освобождающий захваченные мидлетом ресурсы и удаляющий сам мидлет из памяти, то есть закрывающий работу приложения. И самый последний метод основного класса мидлета `commandAction()` интерфейса `CommandListener` обрабатывает назначенные события.

```
public void commandAction(Command c, Displayable s)
{
    if (c == exitCommand)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
```

По сути, этот метод является обработчиком событий приложения для всех имеющихся команд. В нашем примере происходит обработка команды **Выход**. После чего происходит освобождение всех захваченных ресурсов и мидлет выгружается из памяти с помощью методов `destroyApp()` и `notifyDestroyed()`. Рассмотрев вкратце общую модель работы мидлета, давайте разберем более подробно процесс работы приложений, а также жизненный цикл всей программы.

5.1.1. Модель работы мидлета

После успешной компиляции примера `HelloMIDlet` и запуска на любом понравившемся вам эмуляторе на дисплее эмулятора в верхнем левом углу появится надпись **HelloMIDlet**. В нашем случае мидлет всего один, но возможна ситуация, когда будет несколько мидлетов, то есть `MIDlet suite` (набор мидлетов), тогда сервис телефона автоматически создаст меню, выстроив все имеющиеся мидлеты друг под другом, предоставляя возможность в выборе программы. Но это не значит, что после того, как вы войдете в ваше приложение, сервис телефона и далее будет так же создавать за вас меню. Нет, действия по структуризации приложения, созданию списков и меню лежат на плечах программиста. На рис. 5.3 изображено меню телефона с набором различных программ.

Для того чтобы запустить программу на эмуляторе, необходимо нажать кнопку **Select** (выбор), после чего вы попадаете непосредственно в рабочий цикл приложения. После нажатия клавиши **Select**, то есть подачи команды активизации приложения, в мидлете происходит поиск метода `startApp()`, который является начальной и входной точкой всех программ. И уже в методе `startApp()` идет обработка соответствующего программного кода и как следствие выполнение работы приложения. Конечно, код, находящийся до метода `startApp()`: объекты, конструктор и т. д., -



Рис. 5.3. Меню

также инициализируется, но активизация рабочего процесса мидлета происходит с вызовом метода `startApp()`.

В рассматриваемом примере мы воспользовались классом `TextBox`, который создает область для вывода текста. В связи с этим на экране появится текстовая строка, изначально прописанная в программном коде в параметре `text` конструктора класса `TextBox`. Этот текст на дисплее можно отредактировать как угодно, воспользовавшись клавишами эмулятора телефона. В нижнем левом углу экрана вы увидите надпись **Выход**. Нажав на клавишу телефона под этой надписью, вы автоматически выйдете из программы. При нажатии подэкранной клавиши **Выход** происходит запуск метода `commandAction()`, реакция которого на команду **Выход** равносильна обработке событий для переменной `exitCommand`. Обработка события в данном случае подразумевает вызов двух методов - `destroyApp()` и `notifyDestroyed()`, благодаря которым происходят обнуление ссылок, если таковые имеются, и выгрузка мидлета и всего приложения из памяти телефона, возвращая вас в меню телефона, откуда производился запуск рассматриваемой программы. В итоге весь жизненный цикл работы приложения сводится к периоду активизации программы до выхода из нее. На рис. 5.4 показана общая модель работы мидлета.

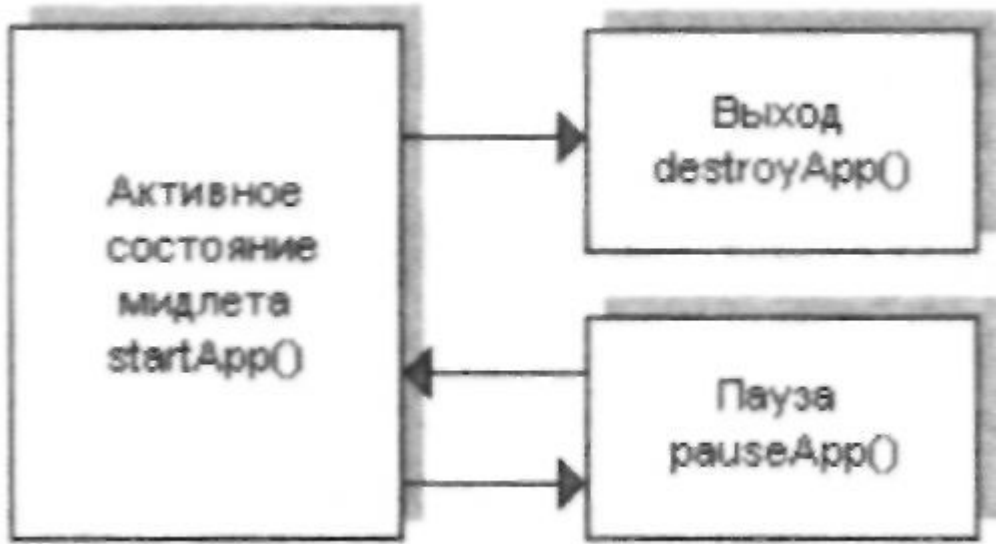


Рис. 5.4. Модель работы мидлета

Основной класс мидлета - это своего рода мотор всего приложения, тогда как функциональную часть лучше разбить на необходимое количество классов. Конечно, возможна интеграция всего программного кода приложения в код основного класса мидлета, но, во-первых, это непрофессионально, а во-вторых, это просто неудобно. Хорошо, если вы пишете совсем маленькое приложение подобно нашему примеру, но когда речь идет о более серьезном программном продукте, надо разработать четкую структуру классов, продумать общую модель взаимодействия и в конечном итоге написать код приложения.

При рассмотрении примеров этой и следующей главы мы будем пока использовать основной класс мидлета, формируя в этом классе дополнительные объекты рассматриваемых классов платформы Java 2 ME, для того чтобы не путаться в большом количестве исходных файлов. Но, начиная с главы 7, перейдем к профессиональной модели построения приложений.

5.2. Пользовательский интерфейс

Когда мы рассматривали механизм работы примера из листинга 5.1, я думаю, вы подметили некий поэкранный принцип отображения информации на дисплее. Первый экран показывал список доступных приложений, после выбора одного из

них вы попадали на экран этого приложения. Нажав кнопку **Выход**, происходило возвращение к экрану выбора. В программах Java 2 ME такая схема поэкранного отображения информации является основной. Если вы никогда не обращали на это внимания при работе со своим телефоном, то самое время взять его и пощелкать джойстиком. В приложениях для мобильных телефонов, основанных на экранах в Java 2 ME отсутствуют окна и фреймы, в отличие от Java 2 SE.

Телефоны ограничены в системных ресурсах, и разнообразная красивая роскошь, к сожалению, не позволительна для этих маленьких устройств. Поэтому при создании конечного продукта стоит с особой тщательностью продумывать основные составляющие будущего приложения. Основываясь на поэкранном отображении информации, необходимо создавать интуитивно понятную структуру приложения, образуя при этом четкую экранную навигацию. Если пользователь заблудится в вашей программе, он просто удалит ее из памяти телефона и никогда к ней больше не вернется, а вы потеряете потенциального покупателя.

Как уже отмечалось, экран телефона представлен классом `Display`. Каждый мидлет может иметь только один объект класса `Display`, возвращаемый мидлету при помощи метода `getDisplay()`, определяя тем самым текущий дисплей телефона для мидлета.

Платформа Java 2 ME обладает пакетом `javax.microedition.lcdui`, включающим в себя классы для работы с пользовательским интерфейсом UI (user interface). Большое количество классов, входящих в этот пакет, будут подробно рассматриваться в следующей главе. Самым главным классом пользовательского интерфейса является класс `Displayable`. С основы абстрактного класса `Displayable` происходит построение основной части графического интерфейса приложения. На рис. 5.5 показана структура пользовательского интерфейса пакета `javax.microedition.lcdui`.

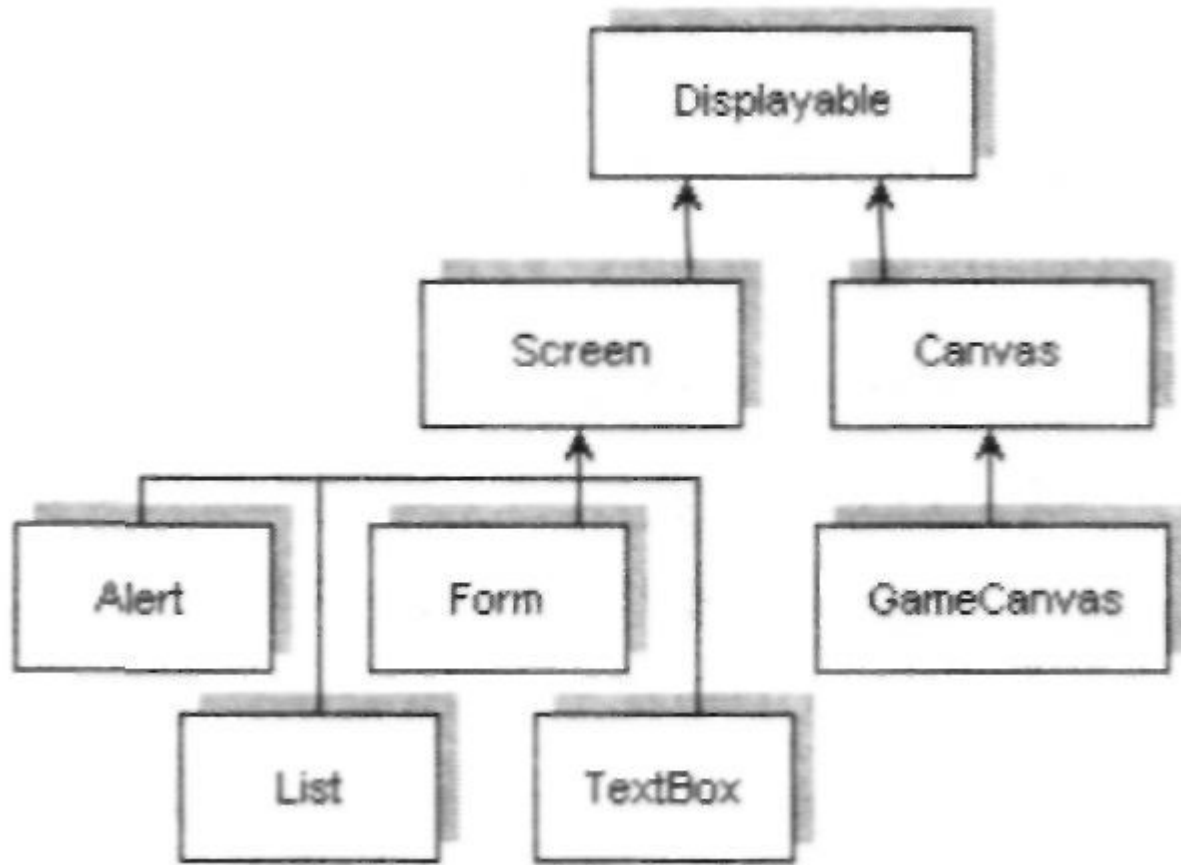


Рис. 5.5. Структура пользовательского интерфейса

От класса диспетчера `Display` зависит, какой из классов `Displayable` будет отображен на экране. В свою очередь, только один класс `Displayable` может быть одновременно показан на экране. То есть объект класса `TextBox`, грубо говоря, существует в своем экране, объект класса `List` - в своем, и оба объекта не могут существовать вместе на одном экране, определяя тем самым правило поэкранного отражения информации на дисплее телефона.

Далее в иерархии структуры пользовательского интерфейса, показанного с помощью рис. 5.5, идут два абстрактных класса: `Screen` и `Canvas`. На этой стадии происходит разделение классов пользовательского интерфейса на высокоуровневый класс, назначенный классу `Screen` и всей его дальнейшей иерархии наследования, и низкоуровневый класс `Canvas`. Оба класса создают структуру интерфейсов, разделенную на высокоуровневый и низкоуровневый пользовательский интерфейсы.

Высокоуровневый интерфейс содержит средства для работы с пользовательским интерфейсом, созданные на основе классов шаблонов, использование которых приводит к построению жестко заданного интерфейса. Например, задействованный в исходном коде HelloMIDlet проекта Demo класс TextBox не может никаким образом изменить экран телефона. Экран, представленный классом TextBox, - это текстовый контейнер, в котором можно осуществлять вывод, удаление и редакцию текста, и не более того. То есть классы высокоуровневого интерфейса - это жестко заданная модель отображения пользовательского интерфейса на экране телефона, с помощью которых программист организует навигацию, списки, меню, текстовые контейнеры, группы выбираемых элементов и т. д.

Низкоуровневый интерфейс предоставляет графические средства для рисования на экране различных графических элементов и обработку команд, посылаемых с клавиш телефона. Низкоуровневый интерфейс позволяет рисовать на экране телефона, добавляя тем самым в приложение красивые графические элементы в виде таблицы, изображений, полей, заставок и т. д. Такое разделение классов существует сугубо в теоретическом виде, и ничто вам не мешает комбинировать элементы обоих интерфейсов в одной программе, создавая красивое интерактивное приложение.

На рис. 5.5 была представлена часть пакета javax.microedition.lcdui.*, потому что классы Canvas и GameCanvas будут подробно рассматриваться соответственно в главах 7 и 8. Рассмотрим некоторые характеристики классов Alert, TextBox, Form и List, представляющие высокоуровневый интерфейс:

- Alert - предоставляет возможность в создании уведомлений или сообщений об ошибках в виде отдельных экранов;
- TextBox - массив данных, содержащий текстовую информацию с возможностью ее редакции;
- Form - экранный контейнер, в котором можно разместить различные элементы интерфейса с помощью дополнительных подклассов класса Item;
- List - список элементов, позволяющий производить выбор различных операций.

5.3. Переход с экрана на экран

Прежде чем мы приступим к изучению основ перехода в приложении с одного экрана на другой, хочу обозначить стоящие перед нами задачи в рассмотрении средств по созданию пользовательского интерфейса. Сейчас вы имеете некоторое представление о модели работы программ для мобильных телефонов. Далее вы изучите основной способ перехода от экрана к экрану внутри приложения на простом примере. Потом код несколько усложнится и будет показан механизм навигации в программах на Java 2 ME. И уже в главе 6 будут изучены все классы пользовательского интерфейса для создания действительно красивых программ. Сейчас мы идем от простого к сложному, и я хочу предложить вам новый пример кода, на основе которого будет изучена схема перехода с одного экрана на другой. Попутно мы задействуем все четыре высокоуровневых класса: TextBox, Form, List и Alert.

Прежде чем коснуться непосредственно программирования любого приложения, нужно уделить внимание теоретической части создаваемой программы. Лично я, когда разрабатываю некую программу для мобильного телефона, беру чистый листок бумаги, карандаш и рисую предполагаемый набор экранов, указывая на связь между ними с помощью стрелочек. Это, конечно, не язык UML, но достаточно просто и эффективно. На рис. 5.6 даются все четыре дисплея и связь между ними.

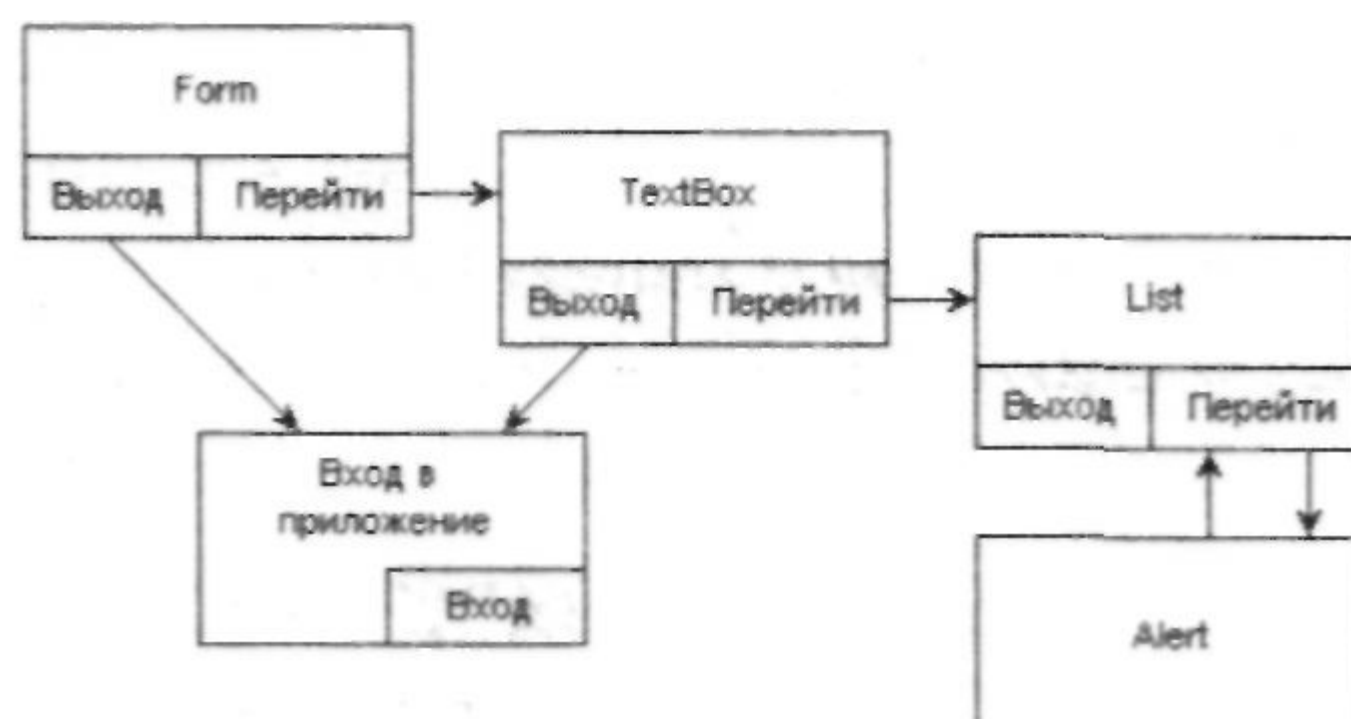


Рис. 5.6. Переход с экрана на экран

Идея этого показательного примера очень проста. Первым делом после входа в приложение вы попадаете в класс `Form`, являющийся неким контейнером для элементов пользовательского интерфейса. В этом примере классы `Form`, `List`, `TextBox` и `Alert` не задействованы в полном объеме, а только показывают информативную надпись названия класса. После попадания на экран, представленный

классом `Form`, появятся две кнопки: выхода из программы и перехода на следующий экран, представленный классом `TextBox`. После перехода с экрана, представленного классом `Form` в `TextBox`, на экране появятся две кнопки: выхода из приложения и перехода. Кнопка перехода приведет вас на экран, представленный

лассом `List`, дав аналогичную возможность выбора перехода. Выбрав переход на экран, представленный *классом* `Alert`, вы увидите на некоторое время экран с надписью **Alert** и автоматически возвратитесь в `List`. По своей специфике класс `Alert` предназначен для сообщения информации об ошибке или исключительной ситуации, этим объясняются и соответствующие действия этого класса. Приступим к написанию кода этого примера. Первым делом импортируем две библиотеки:

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

Затем создаем класс `Perexod`, наследуемый от класса `M1Dlet`.

```
public class Perexod extends MIDlet implements
CommandListener
```

Теперь необходимо создать объекты класса `Command`. Объект `exitMidlet` нужен для выхода из программы. Код, реализующий это событие, аналогичен коду из предыдущего примера, рассмотренного в листинге 5.1. И еще три объекта будут служить для перехода от экрана к экрану:

```
private Command exitMidlet;  
private Command perexodTextBox;
```

```
private Command perexodList;
private Command perexodAlert;
private Display mydisplay;
```

Названия этих объектов достаточно информативны и в объяснении не нуждаются. Далее очередь подошла к конструктору класса `Perexod`. Первым делом сохраним ссылку на `Display` в переменной `mydisplay`:

```
mydisplay = Display.getDisplay(this);
```

Следующим шагом создадим два объекта класса `Command`: один - для выхода из программы, другой - для перехода на экран, представленный классом `TextBox`.

```
exitMidlet = new Command("Выход", Command.EXIT, 1);
perexodTextBox = new Command("Перейти", Command.SCREEN, 2);
```

Создание этих объектов в конструкторе - не обязательное условие. Просто я основывался на предыдущем примере и оставил примерную структуру приложения для понимания. На самом деле все четыре объекта класса `Command` можно было инициализировать еще при их объявлении в начале класса `MainMidlet`, что более читабельно. Следующим кодом за конструктором идет метод `startApp()`, внутри которого создается объект класса `Form`. Добавим при помощи метода `addCommand()` команду **Выход** (это выход из приложения) и команду **Переход** (это переход на экран, представленный классом `TextBox`). Назначим обработчик событий классу `Form` методом `setCommandListener()` и присоединим объект `myform` класса `Form` к текущему дисплею при помощи метода `setCurrent()`:

```
public void startApp()
{
    Form myform = new Form(" Это объект класса Form " );

    myform.addCommand(exitMidlet);
    myform.addCommand(perexodTextBox);
    myform.setCommandListener(this);
    // отражает текущий экран
    mydisplay.setCurrent(myform);
}
```

Когда вы запустите программу или войдете в рабочий цикл мидлета, то автоматически инициализируются объекты классов и конструктор класса `Perexod`, а работа программы начнется с вызова метода `startApp()`. Теперь необходимо назначить соответствующие действия клавишам перехода в методе `commandAction()` интерфейса `CommandListener` для обработки пользовательских команд. Переход по кнопке **Выход** вам уже знаком из предыдущего примера, поэтому оставим все почти без изменения, за исключением информационной команды `exitMidlet`:

```
if (c == exitMidlet)
{
```

```
destroyApp(false);  
notifyDestroyed();  
}
```

А теперь вплотную займемся командой **Перейти**. Что от нас требуется? В момент запуска программы мы попадаем на экран, представленный классом `Form` посредством команды `perexodTextBox`, а требуется перейти на экран, представленный классом `TextBox`. Для создания обработчика событий команды **Перейти** нужно сформировать объект класса `TextBox`, позаботиться о следующей кнопке перехода `perexodList` для перехода на экран, представленный классом `List`, добавить команду **Выход** для выхода из программы и команду **Перейти**. Осталось добавить обработчик событий и присоединить созданный `TextBox` к текущему экрану. Смотрим, что у нас получилось:

```
if (c == perexodTextBox)  
{  
    TextBox tb = new TextBox("TextBox ", "Текст ", 256, 0);  
    perexodList = new Command("Перейти", Command.SCREEN, 2);  
    tb.addCommand(exitMidlet);  
    tb.addCommand(perexodList);  
    tb.setCommandListener(this);  
    Display.getDisplay(this).setCurrent(tb);  
}
```

Обратите внимание на последнюю строку кода в теле условного оператора `if`:

```
Display.getDisplay(this).setCurrent(tb);
```

В данном случае присоединяется созданный объект `tb` класса `TextBox` к текущему экрану. Мы говорим о смене экранов для создания четкого и информативного пользовательского интерфейса, на самом деле смены экранов в буквальном смысле не происходит. Существует только один дисплей, назначенный для класса `Display`, который отвечает за то, что будет нарисовано на экране телефона, а именно какой из объектов `Displayable`. Только один объект `Displayable` может быть отображен за один раз на экране. То есть существует всего один физический дисплей, к которому присоединяются необходимые объекты классов пользовательского интерфейса через обработку событий. Имеющийся экран просто постоянно перерисовывается, отображая тот или иной объект востребованного класса, а иначе говоря, к текущему дисплею присоединяется объект класса, создавая иллюзию смены экранов. Системные ресурсы телефонов пока малы, поэтому приходится идти на такие хитрости.

Затем в рассматриваемом примере необходимо перейти на экран, представленный классом `List`. Поскольку мы имеем аналогичные требования к экземпляру этого класса, то обработка событий и создание класса `List` будут идентичными классу `TextBox`.

```
if (c == perexodList)  
{  
    List mylist = new List("List", List.IMPLICIT);  
    perexodAlert = new Command("Перейти", Command.SCREEN, 2);  
}
```



```
mylist.addCommand(exitMidlet);
mylist.addCommand(perexodAlert);
mylist.setCommandListener(this);
Display.getDisplay(this).setCurrent(mylist);
}
```

Позаботившись о переходе на экран, представленный классом `Alert`, и о выходе из приложения, можно создать код для объекта `Alert`, который впоследствии можно присоединить к текущему экрану. Класс `Alert` несколько специфичен, это вам станет понятно, как только вы откроете окно, отвечающее за отображение объекта. Попробуйте после компиляции рассмотренного примера сделать для `Alert` команду **Выход** и посмотрите, что получится. Теперь соединим рассмотренный код в одно целое, получив готовую программу, представленную в листинге 5.2. Пример также можно найти на прилагаемом к книге компакт-диске в папке `\Code\ Chapter5\Listing5_2\src`.

```
/**
Листинг 5.2
Переход с экрана на экран
*/

// подключаем пакеты
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
// создаем класс Perexod
public class Perexod extends MIDlet implements
CommandListener
{
// команда выхода из программы
private Command exitMidlet;
// команда перехода в программе
private Command perexodTextBox;
private Command perexodList;
private Command perexodAlert;
// дисплей
private Display mydisplay;
// конструктор класса Perexod
public Perexod()
{
mydisplay = Display.getDisplay(this);
// выход из приложения
exitMidlet = new Command("Выход", Command.EXIT, 1);
// переход TextBox
perexodTextBox = new Command("Перейти", Command.SCREEN, 2);
}
```

```
// входная точка всей программы
public void startApp()
{
    // создаем объект класса Form
    Form myform = new Form(" Это объект класса Form " ) ;
    // добавляем команду выхода из программы
    myform.addCommand(exitMidlet);
    // добавляем команду перехода в TextBox
    myform.addCommand(perexodTextBox);
    // устанавливаем обработчик событий для команд
    // объекта класса Form
    myform.setCommandListener(this);
    // отражаем текущий экран
    mydisplay.setCurrent(myform);

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}
// обработчик событий в программе
public void commandAction(Command c, Displayable d)
{
    // обработка команды выход
    if (c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();

// обработка команды перехода в TextBox
    if (c == perexodTextBox)
    {
        TextBox tb = new TextBox("TextBox ", "Текст ", 256, 0);
        perexodList = new Command("Перейти",Command.SCREEN, 2);
        tb.addCommand(exitMidlet);
        tb.addCommand(perexodList);
        tb.setCommandListener(this);
        Display.getDisplay(this).setCurrent(tb);
    }
    // обработка команды перехода в List
    if (c == perexodList)
    {
        List mylist = new List("List", List.IMPLICIT);
        perexodAlert = new Command("Перейти",Command.SCREEN,
        mylist.addCommand(exitMidlet);
        mylist.addCommand(perexodAlert);
        mylist.setCommandListener(this);
```

```
        Display.getDisplay(this).setCurrent(mylist);
    }
    // обработка команды перехода в Alert
    if (c == perexodAlert)
    {
        Alert myalert = new Alert("Alert", "Текст", null, null);
        Display.getDisplay(this).setCurrent(myalert);
    }
}
}
```

После компиляции примера пройдите по всей программе и убедитесь, что вам понятны общая идея и принцип работы смены экранов, на которых строятся все приложения в Java 2 ME.

5.4. Навигация

Рассмотренный пример из листинга 5.2 раскрыл суть перехода с одного экрана на другой. Как вы заметили, принцип смены экранов довольно прост: достаточно добавить необходимую команду с помощью метода `addCommand()`, установить обработчик событий для этого экрана и создать код в методе `commandAction()`, адекватно реагирующий на заданные действия.

Познакомившись с моделью смены экранов и закрепив свои знания на практике, можно переходить к более осмысленной *навигации* в приложении. В предыдущем примере происходила последовательная смена экранов без возможности возврата либо перехода на необходимый экран. Такая структура хороша для изучения, но абсолютно не годится для серьезного приложения. Телефоны различных марок имеют собственные механизмы перехода, предоставляемые операционной системой или прошивкой мобильного телефона. Механизм, использованный в Java 2 ME для приложений, созданных на этом языке, предоставляет не менее мощные, а главное, простые средства для навигации в программе. Самый простой и, как мне кажется, эффективный способ - это использовать автоматически созданное меню при помощи сервиса телефона. Когда вы добавляете к заданному экрану с присоединенным к нему объектом `Displayable` команды обработки в виде двух подэкранных клавиш, вы имеете всего два видимых варианта клавиш - слева и справа. Как только вы добавите с помощью функции `addCommand()` более двух команд, сервис телефона автоматически создаст на правой или левой подэкранной клавише телефона (в зависимости от марки производителя) выпадающее меню. При нажатии клавиши, отвечающей за меню, появится меню, отображающее полный список имеющихся команд. На рис. 5.7 изображено контекстное меню, созданное эмулятором телефона среды программирования J2ME Wireless Toolkit.

Задав нужные команды для конкретного объекта `Displayable` и создав код их обработки, вы получаете отличный механизм навигации. Но это не единственный способ перехода с экрана на экран. Каждый из четырех классов - `Form`, `List`, `TextBox` и `Alert` - имеет свои встроенные средства для создания списков, меню,

Рис. 5.7. Автоматически созданное меню

таблиц, полей, загрузки изображений и т. д. При знакомстве с каждым из классов мы обязательно рассмотрим имеющиеся возможности. А пока давайте разберем механизм автоматического создания меню и обработки имеющихся команд.

Итак, к каждому из задействованных классов нам надо добавить набор команд для перехода на нужный экран и обработать, а точнее создать код, реагирующий на назначенные команды. Рисунок 5.8 живописно изображает и отчасти решает поставленную перед нами задачу.

Теперь сосредоточимся на одном из вариантов программного кода, решающего проблему с навигацией. Первым делом создадим класс, назвав его Navigator.

```
public class Navigator extends MIDlet implements
CommandListener
```

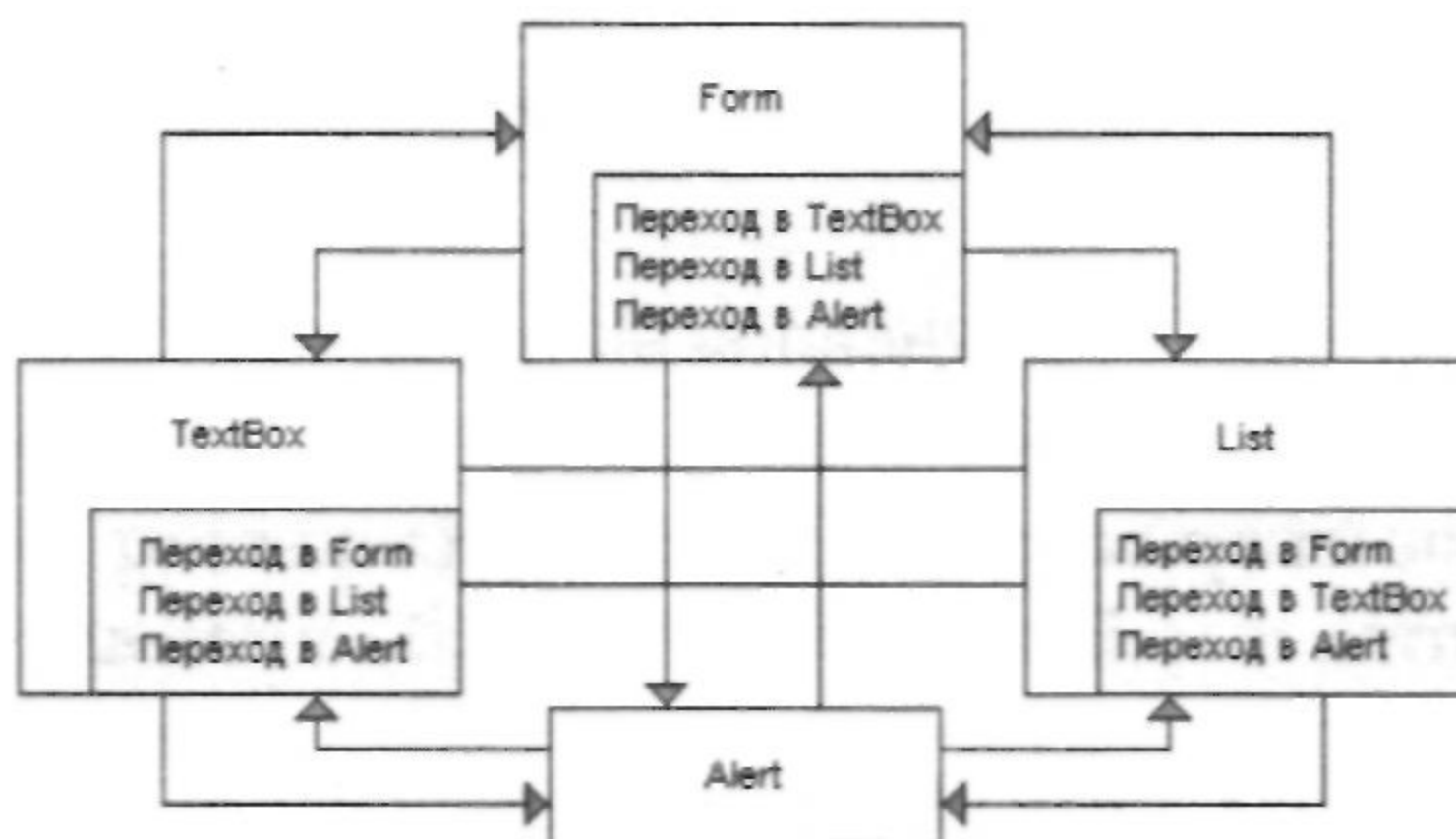


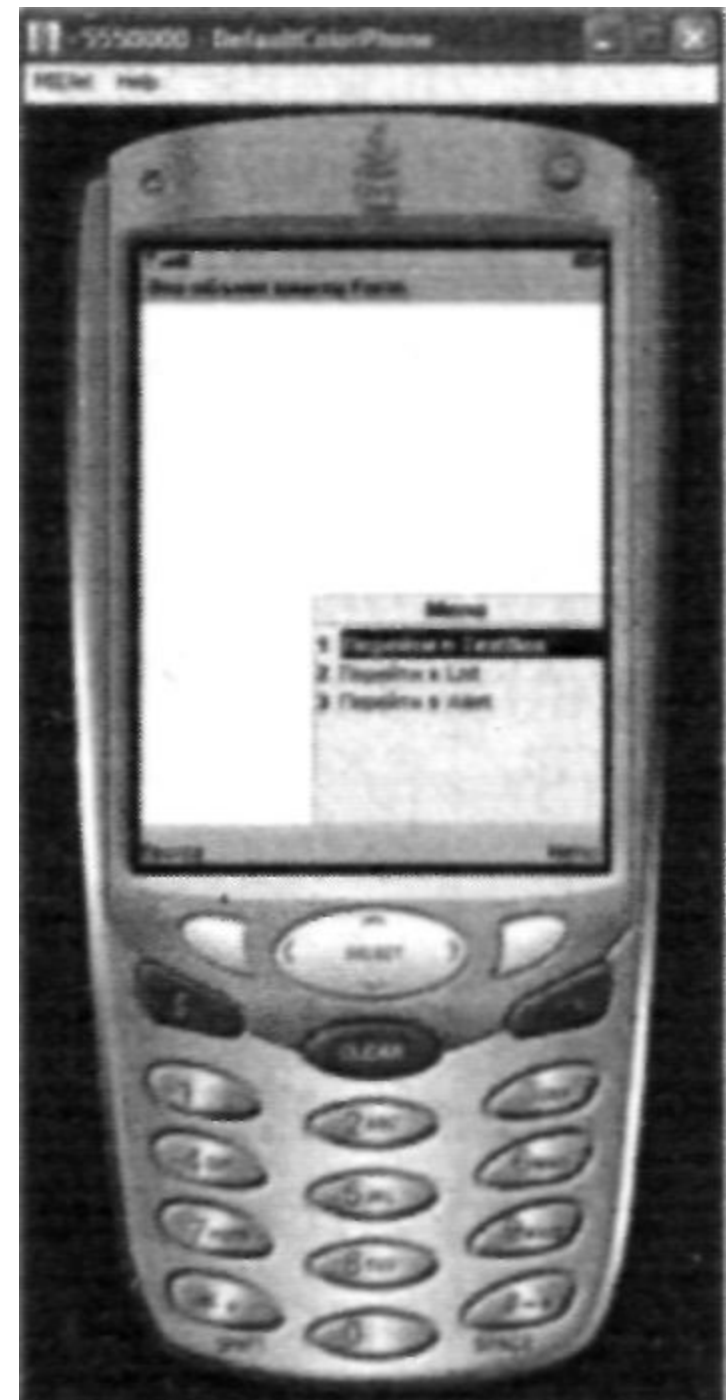
Рис. 5.8. Схема перехода с экрана на экран

В исходном коде до строк вызова конструктора класса Navigator добавим объект, содержащий команду **Выход**.

```
private Command exitMidlet = new Command("Выход",
Command.EXIT, 1);
```

Потом необходимо создать четыре объекта для каждого из задействованных классов: Form, TextBox, List и Alert. Созданные объекты будут отвечать за обработку команд перехода на экраны, представленные соответствующими классами.

```
private Command perexodTextBox = new Command("B
TextBox", Command.SCREEN, 2);
private Command perexodList = new Command("B List",
Command.SCREEN, 2);
private Command perexodAlert = new Command("B Alert",
Command.SCREEN, 2);
private Command perexodForm = new Command("B Form",
Command.SCREEN, 2);
```



Информативные названия всех объектов понятны, но, естественно, выбранные мною названия ни к чему вас не обязывают. Созданные объекты являются объектами класса `Command`, отвечающего за создание команд, которые впоследствии можно определить для каждого из классов `Form`, `TextBox`, `List` и `Alert`. Позже, в коде мидлета, мы будем задавать соответствующие блоки обработки событий непосредственно в методе `commandAction()` при помощи оператора `if` и созданных объектов.

Теперь нам нужно объявить и инициализировать объекты четырех классов `Form`, `TextBox`, `List` и `Alert`.

```
private Form myform = new Form(" Это объект класса Form " ) ;
private List mylist = new List(" Это объект класса List ",
List.IMPLICIT);
private TextBox mytextbox = new TextBox(" Это TextBox ",
" Текст ", 256, 0 );
private Alert myalert = new Alert("Это Alert", "Alert
исчезнет", null, null);
private Display mydisplay;
```

В конструкторе класса `Navigator` происходит инициализация объекта `mydisplay`.

```
public Navigator()
{
    mydisplay = Display.getDisplay(this);
}
```

Следующая наша задача - это реализация *метода* `startApp()`. Сейчас необходимо решить, какой из классов будет первым появляться на экране, и добавить к нему команды перехода в другие классы, а также команду выхода из приложения. В предыдущем примере первым появлялся класс `Form`. Его и определим как основной объект, в который попадет пользователь, запускающий приложение.

```
public void startApp()
{
    myform.addCommand(exitMidlet);
    myform.addCommand(perexodTextBox);
    myform.addCommand(perexodList);
    myform.addCommand(perexodAlert);
    myform.setCommandListener(this);
    mydisplay.setCurrent(myform);
}
```

Последняя строка метода `startApp()` отображает объект `myform` на дисплее со всеми имеющимися командами. Как уже говорилось, командам, которым не хватит телефонных клавиш, будет автоматически создано свое собственное меню.

После того как вы попадете в основное окно приложения, которое мы определили для объекта `myform`, над левой или правой подэкранной клавишей появятся

команда выхода из приложения и команда **Menu**. При нажатии на клавишу **Menu** на экране телефона появится всплывающее меню с добавленными ранее командами перехода на экраны, представленные классами `TextBox`, `List` и `Alert`.

Следующей нашей задачей является написание кода для обработки событий созданных команд в методе `commandAction()`. Исходный код, обрабатывающий команду **Выход** из приложения, идентичен коду из примера в листинге 5.2 и в большинстве рассматриваемых впоследствии примеров останется таковым. Дальнейшие действия состоят в обработке команды перехода на экран, представленный классом `TextBox`.

```
if (c == perexodTextBox)
{
    mytextbox.addCommand(exitMidlet);
    mytextbox.addCommand(perexodForm);
    mytextbox.addCommand(perexodList);
    mytextbox.addCommand(perexodAlert);
    mytextbox.setCommandListener(this);
    mydisplay.setCurrent(mytextbox);
}
```

Сразу после того, как пользователь попадет на экран, представленный классом `Form`, и в контекстном меню выберет команду **Перейти в TextBox**, произойдет обработка блока команд `perexodTextBox`. Добавляются все команды к объекту `mytextbox`, устанавливается обработчик событий, и в итоге отображается текущий экран, содержащий все созданные компоненты объекта `mytextbox`. Точно так же как и на экране с объектом `myform`, существуют меню перехода и кнопка выхода.

Обработка событий для объекта `mylist` происходит с помощью команды `perexodList` и аналогично рассмотренному коду для объекта `mytextbox`. С той лишь разницей, что используются соответствующие команды для объекта `mylist`. В итоге листинг 5.3 связывает все разрозненные фрагменты кода этого подраздела, собирая воедино очень простую и в то же время мощную программу отличной системы навигации.

```
/**
```

```
Листинг 5.3
```

```
Навигация в приложении
```

```
*/
```

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

```
public class Navigator extends MIDlet implements
CommandListener
```

```
{
// команда выхода из приложения
```



```
private Command exitMidlet = new Command("Выход",
Command.EXIT, 1);
// команда перехода в TextBox
private Command perexodTextBox = new Command("В
TextBox",Command.SCREEN, 2);
// команда перехода в List
private Command perexodList = new Command("В List",
Command.SCREEN, 2);
// команда перехода в Alert
private Command perexodAlert = new Command("В Alert",
Command.SCREEN, 2);
// команда перехода в Form
private Command perexodForm = new Command("В Form",
Command.SCREEN, 2);
// объект класса Form
private Form myform = new Form(" Это объект класса Form " );
// объект класса List
private List mylist = new List(" Это объект класса List",
List.IMPLICIT);
// объект класса TextBox
private TextBox mytextbox = new TextBox(" Это TextBox ",
" Текст ", 256, 0);
// объект класса Alert
private Alert myalert = new Alert("Это Alert","Alert
исчезнет",null,null);
// объект mydisplay представляет экран телефона
private Display mydisplay;

public Navigator()
{
mydisplay = Display.getDisplay(this);
}

public void startApp()
{
    // добавить команды перехода в Form
    myform.addCommand(exitMidlet);
    myform.addCommand(perexodTextBox);
    myform.addCommand(perexodList);
    myform.addCommand(perexodAlert);
    // установка обработчика событий для Form
    myform.setCommandListener(this);
    // отразить текущий дисплей
    mydisplay.setCurrent(myform);
}
```

```
}

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}
public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if (c == exitMidlet)
    {
        destroyApp(false) ;
        notifyDestroyed() ;
    }
    // переход в TextBox
    if (c == perexodTextBox)
    {
        mytextbox.addCommand(exitMidlet) ;
        mytextbox.addCommand(perexodForm) ;
        mytextbox.addCommand(perexodList) ;
        mytextbox.addCommand(perexodAlert) ;
        mytextbox.setCommandListener(this) ;
        mydisplay.setCurrent(mytextbox) ;
    }
    // переход в List
    if (c == perexodList)
    {
        mylist.addCommand(exitMidlet) ;
        mylist.addCommand(perexodForm) ;
        mylist.addCommand(perexodAlert) ;
        mylist.addCommand(perexodTextBox) ;
        mylist.setCommandListener(this) ;
        mydisplay.setCurrent(mylist) ;
    }
    // переход в Alert
    if (c == perexodAlert)
    {
        mydisplay.setCurrent(myalert) ;
    }
    // переход в Form
    if (c == perexodForm) mydisplay.setCurrent(myform) ;
}
}
```

В следующей главе будут изучаться классы высокоуровневого интерфейса, с помощью которых создаются списки, группы элементов, текстовые поля и множество других элементов пользовательского интерфейса.

<http://palata-x.narod.ru>

Глава 6. Классы

пользовательского интерфейса

В Java 2 ME имеется пакет `javax.microedition.lcdui`, определенный для классов пользовательского интерфейса. Как уже отмечалось в главе 5, классы пользовательского интерфейса разделены на высокоуровневый и низкоуровневый интерфейсы. В этой главе будут последовательно рассмотрены все классы высокоуровневого пользовательского интерфейса. Каждый из разделов содержит информацию об одном конкретном классе, предоставляющем ряд возможностей в оформлении интерфейса пользователя. Используя возможности этих классов, вы сможете создавать в приложении списки, группы элементов, загружать в программу изображения, использовать бегущую строку, назначать шрифт текста и многое другое.

В разделах по каждому классу пользовательского интерфейса анализируются конструкторы, основные методы и константы класса, с помощью которых в конце каждого раздела создается приложение, иллюстрирующее работу этого класса, и приводится листинг всей программы. При рассмотрении методов и констант классов используются только основные компоненты. Для детального анализа составляющих одного из классов обратитесь к *приложению 2*, которое выполнено в виде справочника по Java 2 ME.

Единственное, о чем необходимо помнить при проектировании и создании пользовательского интерфейса приложения, - это об используемом профиле MIDP. В некоторые классы, имеющиеся в составе профиля MIDP 1.0, были добавлены новые методы, константы и даже классы из профиля MIDP 2.0. Например, в классе `ChoiceGroup` для профиля MIDP 1.0 доступны 13 методов, а уже в профиле MIDP 2.0 их насчитывается 17, то есть добавлено еще четыре новых. В той же документации по Java 2 ME какой-то четкой грани, разделяющей два профиля, не существует, но при рассмотрении методов и констант упоминается о принадлежности к одному из профилей. Поэтому при создании приложений, например, под профиль MIDP 1.0 необходимо внимательно планировать разработку программ и не задействовать компоненты профиля MIDP 2.0. Если вы разрабатываете программу для профиля MIDP 2.0, то можете пользоваться всеми имеющимися компонентами вне зависимости от принадлежности к одному из профилей.

6.1. Класс `Form`

Основным экранным классом в примерах из главы 5 служил экран, представленный *классом* `Form`. Как вы понимаете, это не обязательное условие, но я выбрал класс `Form` не случайно. Дело в том, что реализация класса `Form` выполнена в виде

контейнера, позволяющего встраивать в себя различные компоненты пользовательского интерфейса. Это, пожалуй, единственный и самый мощный по своим возможностям класс. Само по себе название `Form` подразумевает *создание некой формы* для заполнения экрана телефона. В предыдущих примерах мы использовали так называемую пустую форму этого класса в виде чистого экрана. Впоследствии при упоминании термина «форма» будет подразумеваться экран телефона, представленный объектом класса `Form` для интеграции классов пользовательского интерфейса.

Класс `Form` имеет два конструктора, необходимых при создании объекта этого класса. Первым конструктором мы уже пользовались, и выглядит он достаточно просто:

```
public Form (String title)
```

Параметры конструктора класса `Form`:

- `title` - заголовок появляется в верхней части созданного окна.

Второй конструктор класса `Form` имеет уже два параметра и позволяет встроить компоненты интерфейса в пустую форму.

```
public Form (String title, Item[] items)
```

Параметры конструктора класса `Form`:

- `title` - заголовок окна;
- `items` - массив компонентов, размещаемых в классе `Form`.

В классе `Form` существует набор методов, с помощью которых можно добавить, удалить или вставить компоненты интерфейса. Класс `Form` имеет 12 методов, с помощью которых можно манипулировать компонентами абстрактного класса `Item`. Всего насчитывается 8 компонентов пользовательского интерфейса в иерархии класса `Item`. То есть вы создаете экран, за который отвечает класс `Form`, и интегрируете имеющиеся в вашем распоряжении компоненты класса `Item`. О самом классе `Item` и его иерархии мы поговорим в следующем разделе этой главы, после анализа класса `Form`.

5.1.7. Методы класса *Form*

- `int append (Image img)` - добавляет в форму одно изображение. Класс `Image` дает возможность загрузить изображение на экран телефона, это может быть фон дисплея, элемент интерфейса;
- `int append (Item item)` - этот метод добавляет любой из доступных компонентов класса `Item` в созданную форму;
- `int append (String str)` - добавляет в форму строку текста;
- `void delete (int itemNum)` - удаляет компонент, ссылающийся на параметр `itemNum`;
- `void deleteAll ()` - удаляет все компоненты из имеющейся формы;
- `Item get (int itemNum)` - получает позицию выбранного компонента;
- `int get Height ()` - возвращает высоту экрана в пикселях, доступную для встраиваемых компонентов;

- `int getWidth ()` - возвращает ширину экрана в пикселях, доступную для встраиваемых компонентов;
- `void insert (int itemNum, Item item)` - вставляет компонент в форму до определенного компонента;
- `void set (int itemNum, Item item)` - устанавливает компонент, ссылающийся на компонент `itemNum`, заменяя при этом предшествующий компонент;
- `void setItemStateListener (ItemStateListener iListener)` - устанавливает переменную `iListener` для формы, заменяя при этом предыдущую переменную `iListener`;
- `int size ()` - получает количество компонентов в форме.

Благодаря вышеперечисленным методам все компоненты, находящиеся в форме, могут быть отредактированы надлежащим образом, например:

```
Form myform = new Form ("Пример" );
myform.append (item1);
myform.append (item2);
```

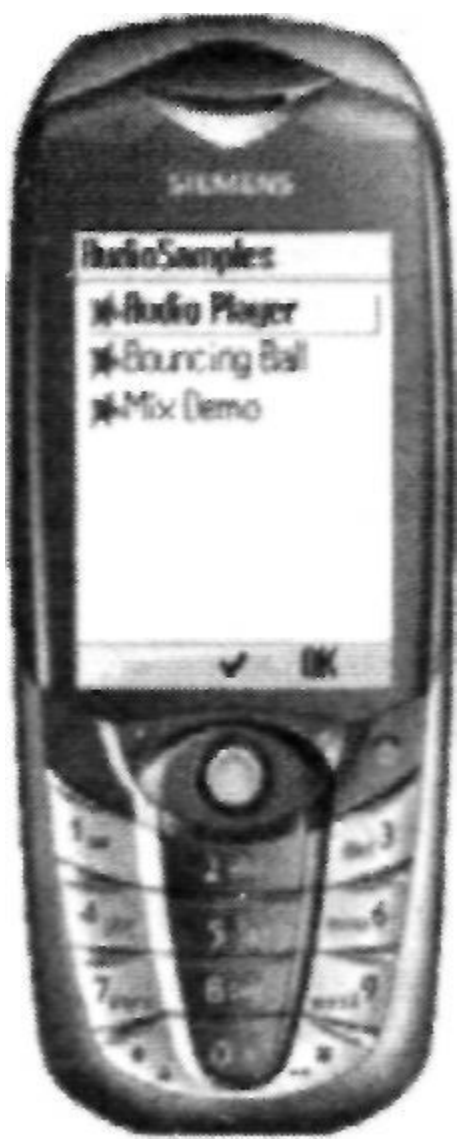


Рис. 6.1. Расположение элементов в форме

В этом примере в созданную пустую форму добавляются два объекта. Оба объекта, разумеется, должны быть созданы в коде. Точно так же можно воспользоваться всеми методами класса `Form` для редактирования создаваемой формы. Добавленные в форму компоненты организованы в виде колонок и обычно располагаются по ширине всего экрана. На рис. 6.1 изображен эмулятор с несколькими компонентами интерфейса.

Все компоненты, встроенные в форму, жестко закреплены и не перемещаются. Редактировать компоненты можно при помощи методов класса `Form`, причем присоединенные компоненты располагаются друг под другом, выравниваясь горизонтально. Пользователь может перемещаться по компонентам формы с помощью клавиш **Вверх** и **Вниз**. Когда количество добавленных компонентов больше видимой части экрана телефона, то автоматически создается

прокрутка. Внизу или вверху экрана появляется стрелочка, сигнализирующая об имеющихся компонентах, выпадающих из зоны видимости. При переходе в нижнюю часть экрана, как только верхний компонент выйдет из зоны видимости, стрелочка автоматически развернется на 180 градусов, указывая в направлении новых компонентов, выпадающих из зоны видимости. Такой механизм реализован в любом телефоне вне зависимости от производителя. Можно добавлять любое количество компонентов в форму, но очевидно, что необходимо задуматься и о дизайне пользовательского интерфейса и не валить все «в кучу». Наилучшим решением будет продуманная структура переходов с экрана на экран.

6.2. Класс Item

Абстрактный *суперкласс* *Item* имеет иерархию из 8 подклассов. Каждый подкласс представляет один из элементов пользовательского интерфейса, например класс *TextField* создает текстовые поля для ввода пароля, адреса электронной почты или просто числовых значений. Все 8 классов, по сути, устанавливают компоненты пользовательского интерфейса, которые встраиваются в форму, определенную классом *Form*. На рис. 6.2 изображена иерархия абстрактного суперкласса *Item*:

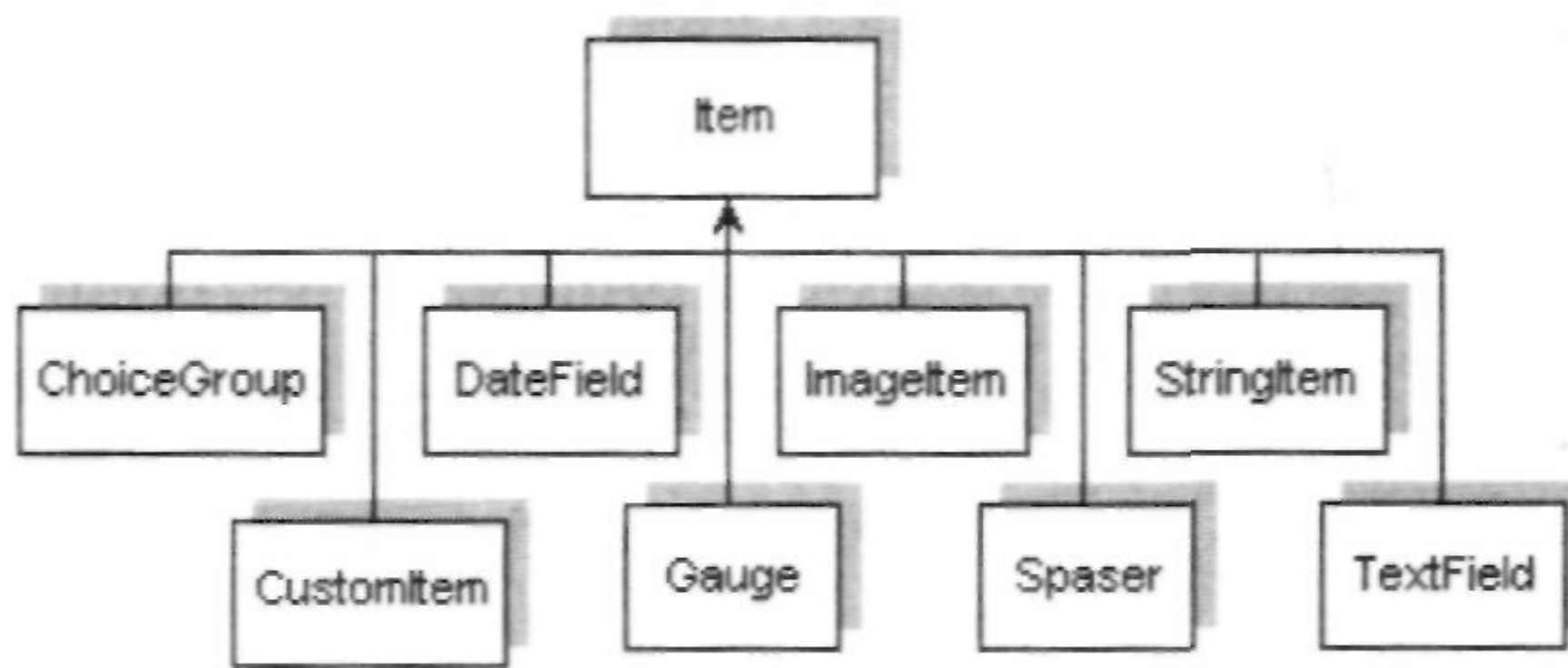


Рис. 6.2. Иерархия суперкласса *Item*

- *ChoiceGroup* - это группа связанных элементов для дальнейшего выбора предполагаемых действий;
- *CustomItem* - с помощью этого класса можно добавлять различные графические элементы в форму;
- *DateField* - класс, с помощью которого имеется возможность редактировать время и дату;
- *Gauge* - допускает графическое отображение диаграмм, процессов загрузки;
- *ImageItem* - осуществляет показ изображения на экране телефона;
- *Spacer* - задает определенное по размеру пространство;
- *StringItem* - с помощью этого класса можно создать произвольный текст. Этот класс не допускает редактирования, он лишь отображает информацию;
- *TextField* - предоставляет текстовые поля для редакции.

Любой из рассмотренных классов наследуется из суперкласса *Item* и может быть добавлен на экран, созданный классом *Form*. Каждый компонент класса *Item* содержит с левой стороны область, где при желании можно отобразить изображение в виде иконки. При перемещении компонента иконка также перемещается вместе с компонентом. Класс *Item* с помощью имеющихся в его составе директив задает в основном формат отображения для любого компонента. Формат определяет заданную ширину, высоту или выравнивание компонентов в форме, а также класс *Item* имеет множество методов, осуществляющих контроль над компонентами.

Методы класса *Item*

- `void addCommand (Command cmd)` - добавляет команду к компоненту;
- `String getLabel ()` - получает метку объекта *Item*;

- `int getLayout()` - использует следующие директивы для размещения компонентов в форме:
 - `LAYOUT_LEFT` - выравнивание по левой стороне;
 - `LAYOUT_RIGHT` - выравнивание по правой стороне;
 - `LAYOUT_CENTER` - выравнивание по центру;
 - `LAYOUT_TOP` - выравнивание к верхней области формы;
 - `LAYOUT_BOTTOM` - выравнивание по нижней стороне экрана;
 - `LAYOUT_VCENTER` - вертикальное выравнивание по центру. Горизонтальная и вертикальная директивы могут комбинироваться при помощи оператора «|»;
- `int getMinimumHeight()` - получает минимальную высоту для компонента;
- `int getMinimumWidth()` - получает минимальную ширину для компонента;
- `int getPreferredHeight()` - получает предпочтительную высоту компонента;
- `int getPreferredWidth()` - получает предпочтительную ширину компонента;
- `void notifyStateChanged()` - компонент, содержащийся в форме. Уведомляет объект `ItemStateListener` о своем состоянии;
- `void removeCommand(Command cmd)` - удаляет команду из компонента;
- `void setDefaultCommand(Command cmd)` - встроенная команда по умолчанию для данного компонента;
- `void setItemCommandListener(ItemCommandListener l)` - устанавливает обработку событий для компонента;
- `void setLabel(String label)` - устанавливает назначенную метку для компонента;
- `void setLayout(int layout)` - устанавливает директивы для форматирования компонента;
- `void setPreferredSize(int width, int height)` - устанавливает оптимальные высоту и ширину компонента.

При использовании вышеперечисленных методов можно настраивать и редактировать компоненты класса `Item`. В иерархии класса `Item` содержится ряд подклассов, обеспечивающих создание интуитивно понятного пользовательского интерфейса. Давайте рассмотрим эти подклассы.

6.2.1. Класс *ChoiceGroup*

С помощью класса *ChoiceGroup* можно встраивать в форму группу элементов. Группы элементов делятся на три типа: эксклюзивный (`EXCLUSIVE`), множественный (`MULTIPLE`) и всплывающий (`POPUP`). Посмотрите на рис. 6.3, где показан эмулятор мобильного телефона, показывающий все три группы элементов.

Первый тип группы элементов на рис. 6.3 выполнен в виде выпадающего меню и запрограммирован на основе типа `POPUP`. В данном случае это список из четырех флажков, с помощью которых можно выбрать заданные действия. Четыре флажка в меню были созданы абсолютно произвольно. Количество флажков и как следствие количество вариантов выбора зависят от задачи, поставленной перед программистом.

Рис. 6.3. Типы группы элементов ChoiceGroup

Следующая группа, изображенная на рис. 6.3, представлена типом MULTIPLE. В этой группе элементов пользователь имеет возможность многократного выбора, то есть можно выбрать сразу несколько вариантов. Обычно такая группа элементов используется при настройке различных опций, где возможно указать сразу несколько вариантов выбора. Третья и последняя группа элементов задается типом EXCLUSIVE, и возможен лишь один вариант выбора заданного флажка. Чтобы создать в приложении необходимую группу элементов, нужно воспользоваться конструктором класса ChoiceGroup. Всего имеется два конструктора. Первый конструктор - с двумя параметрами:

```
public ChoiceGroup(String  
label, int choiceType)
```

Параметры конструктора ChoiceGroup:

- label - это строка текста или информационная метка;
- choiceType - тип, указывающий на создаваемую группу элементов. Его можно задавать, например, следующим образом: Choice.EXCLUSIVE, Choice.MULTIPLE или Choice.POPUP.

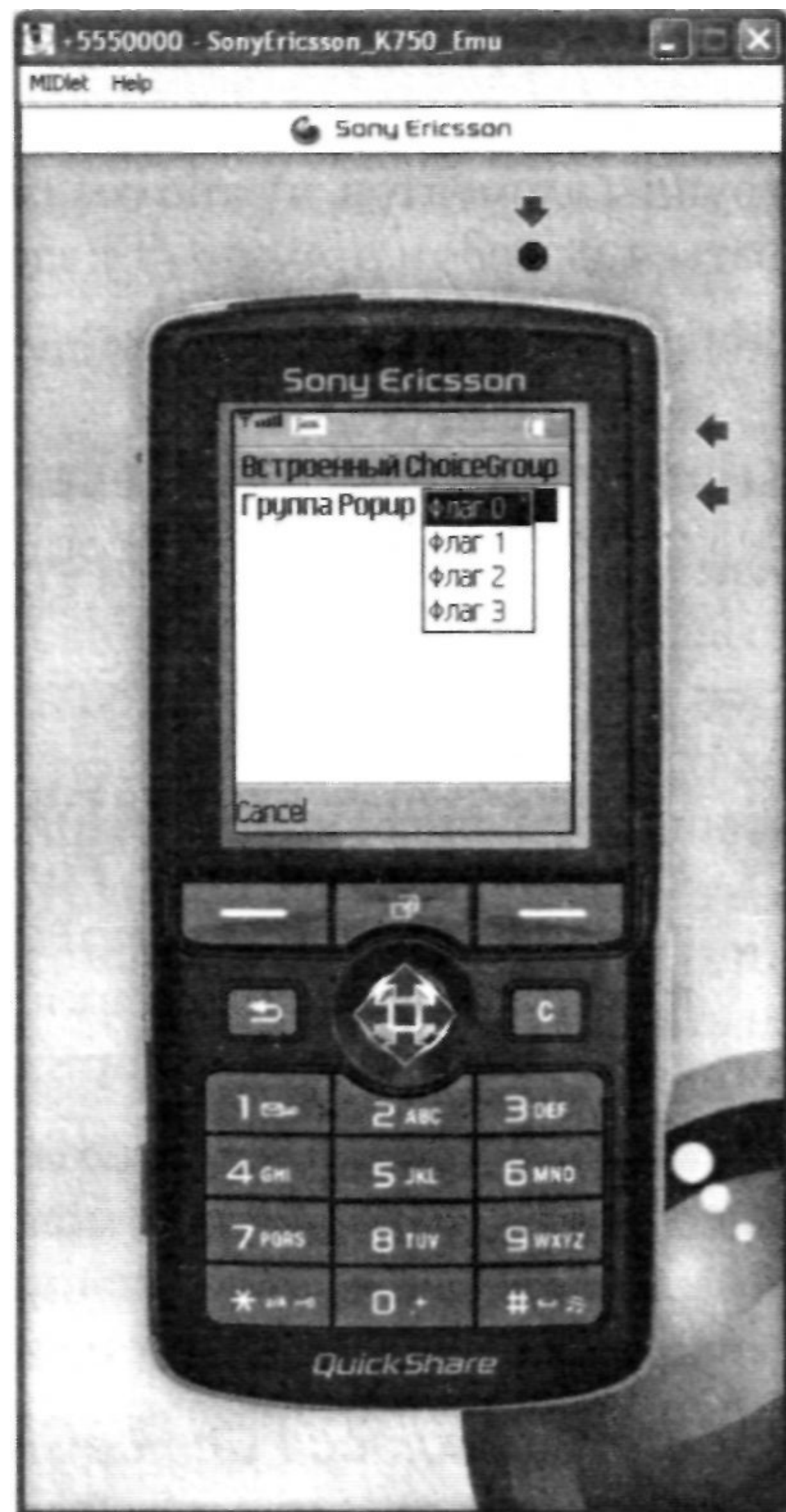
И второй конструктор с четырьмя параметрами, дающий программисту более интересный выбор в использовании графических изображений:

```
public ChoiceGroup(String label,  
int choiceType,  
String[] stringElements,  
Image[] imageElements)
```

Параметры конструктора ChoiceGroup:

- label - строка текста;
- choiceType - тип, указывающий на создаваемую группу элементов;
- stringElements - заданный массив текста для каждого элемента группы;
- imageElements - заданный массив изображений для каждого элемента группы.

Два последних параметра конструктора класса ChoiceGroup предназначены для создания массива названий и изображений для элементов группы, например таким образом:



```
String[] string = {"Флаг 0", "Флаг 1", "Флаг 2", "Флаг 3"}
```

Для того чтобы добавить в пустую форму класса `Form` все три имеющиеся группы элементов, нужно создать три объекта класса `ChoiceGroup` и воспользоваться *методом* `append()` класса `Form`, например:

```
ChoiceGroup groupMultiple = new ChoiceGroup("Группа
Multiple",
ChoiceGroup.MULTIPLE);
ChoiceGroup groupPopup = new ChoiceGroup("Группа
Popup", ChoiceGroup.POPUP);
ChoiceGroup groupExclusive = new ChoiceGroup("Группа
Exclusive",
ChoiceGroup.EXCLUSIVE);
Form myform = new Form("Встроенный ChoiceGroup");
myform.append(groupPopup);
myform.append(groupMultiple);
myform.append(groupExclusive);
```

Большой пользы простое статическое отображение элементов группы на дисплее телефона принести не может. Поэтому необходимо познакомиться с методами класса `ChoiceGroup`, с помощью которых можно удалять, добавлять и отслеживать состояние каждого элемента группы.

Методы класса *ChoiceGroup*

Всего имеется 17 методов, ознакомимся с основными и наиболее используемыми методами:

- `int append(String stringPart, Image imagePart)` - добавляет элемент в группу;
- `void delete (int elementNum)` - удаляет заданный элемент из группы;
- `void deleteAll()` - удаляет все элементы;
- `Font getFont(int elementNum)` - получает используемый шрифт элемента группы;
- `Image getImage (int elementNum)` - получает изображение для элемента группы;
- `int getSelectedFlags(boolean[] selectedArray_return)` - возвращает значение `Boolean` для группы элементов. Обычно эта функция используется с эксклюзивным типом элементов группы;
- `int getSelectedIndex()` - возвращает индекс выбранного элемента группы;
- `void insert(int elementNum, String stringPart, Image imagePart)` - вставляет элемент в группу;
- `boolean isSelected(int elementNum)` - получает выбранную логическую величину;
- `void set (int elementNum, String stringPart, Image imagePart)` -

устанавливает текст и изображения в заданный элемент группы, при этом удаляя предыдущую запись;

- `void setFont(int elementNum, Font font)` - устанавливает шрифт заданному элементу;
- `void setSelectedIndex(int elementNum, boolean selected)` - устанавливает особое состояние для элемента группы при использовании множественного типа;
- `int size()` - возвращает количество используемых элементов группы.

Прежде чем рассматривать практическую часть раздела, давайте разберемся, что именно от нас требуется, чтобы воспользоваться компонентами класса `ChoiceGroup`. Итак, сначала необходимо создать объект класса `Form` или пустую форму, куда можно встроить объекты класса `ChoiceGroup`. Далее необходимо определить, что именно будет происходить при выборе элемента группы. Я предлагаю рассмотреть вариант перехода в новое окно после выбора конкретного элемента группы, где мы выведем простую информационную надпись. Для этого необходимо создать две команды перехода. Одна из команд будет реагировать на выбранный элемент группы, перемещая пользователя в новое окно, а другая команда перехода - возвращать в окно выбора. Пожалуй, это все, что от нас сейчас требуется, поэтому давайте перейдем к реализации данного примера. Предлагаю не рассматривать по отдельности каждый кусок кода всей программы, а проанализировать весь пример целиком, поле чего остановиться на наиболее непонятных местах программного кода. В листинге 6.1 показан исходный код рассматриваемого примера, который находится на компакт-диске в папке `\Code\Chapter6\Listing6_1\src`.

```
/**
Листинг 6.1
Класс ChoiceGroup
*/

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MainClassChoiceGroup extends MIDlet
implements CommandListener
{
    // команда выхода из приложения
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0);
    // команда выбора элемента группы
    private Command vibor = new Command("Выбрать",
    Command.SCREEN, 1);
    // команда возврата в главное окно
    private Command vozvrat = new Command("Назад",
    Command.BACK, 0);
```

```
// объект класса ChoiceGroup
private ChoiceGroup groupPopup;
// объект класса Form
private Form myform;
// объект mydisplay представляет экран телефона
private Display mydisplay;

public MainClassChoiceGroup()
{
mydisplay = Display.getDisplay(this);
}
// текст для элементов группы
private String[] mygroup = {"Флаг 0", "Флаг 1", "Флаг 2",
"Флаг 3"};

public void startApp()
{
    // инициализируем объект groupPopup
    groupPopup = new ChoiceGroup(" Группа Popup",
    ChoiceGroup.POPUP, mygroup, null);
    // создаем форму при помощи объекта Form
    myform = new Form(" Встроенный ChoiceGroup " ) ;
    // добавляем группу элементов
    myform.append(groupPopup);
    myform.addCommand(exitMidlet);
    myform.addCommand(vibor);
    myform.setCommandListener(this);
    // отражаем текущий дисплей
    mydisplay.setCurrent(myform);
}

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if(c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
    // возврат в myform
    if(c == vozvrat)
```

```
{
mydisplay.setCurrent(myform);
}
// обработка выбранного элемента в группе
if (c == vibor)
{
    int i = groupPopup.getSelectedIndex();
    if(i == 0)
    {
        Form formPopup = new Form("Это formPopup "+ mygroup[0]);
        formPopup.append(mygroup[0]);
        formPopup.addCommand(vozvrat);
        formPopup.addCommand(exitMidlet);
        formPopup.setCommandListener(this);
        mydisplay.setCurrent(formPopup);
    }
    if(i == 1)
    {
        Form formPopup = new Form("Это formPopup "+ mygroup[1]);
        formPopup.addCommand(vozvrat);
        formPopup.append(mygroup[1]);
        formPopup.addCommand(exitMidlet);
        formPopup.setCommandListener(this);
        mydisplay.setCurrent(formPopup);
    }
    if(i == 2)
    {
        Form formPopup = new Form("Это formPopup "+ mygroup[2]);
        formPopup.append(mygroup[2]);
        formPopup.addCommand(vozvrat);
        formPopup.addCommand(exitMidlet);
        formPopup.setCommandListener(this);
        mydisplay.setCurrent(formPopup);
    }
    if(i == 3)
    {
        Form formPopup = new Form("Это formPopup "+ mygroup[3]);
        formPopup.append(mygroup [ 3 ] );
        formPopup.addCommand(vozvrat);
        formPopup.addCommand(exitMidlet);
        formPopup.setCommandListener(this);
        mydisplay.setCurrent(formPopup);
    }
}
```



```

}
}

```

Вся программа основывается на классе `MainClassChoiceGrop`. К команде выхода `exitMidlet` добавлены еще две команды обработки событий - это `vozvrat` и `vibor`. Команда `vozvrat` возвращает пользователя обратно в главное окно приложения, в которое он попадает при запуске программы. С помощью команды `vibor` происходит выбор заданных действий, то есть отклик программы на выбранный элемент группы. Как мы уже договорились, каждый элемент группы `POPUP` (всего их будет четыре), должен привести пользователя в свой экран, установленный с помощью класса `Form`. Далее в листинге 6.1 идет объявление необходимых объектов для классов `Form`, `ChoiceGrop` и `Display`. После конструктора идет строка кода, создающая текст для элементов группы:

```

private String[] mygroup = {"Флаг 0", "Флаг 1", "Флаг 2",
    "Флаг 3"};

```

С помощью переменной `mygroup` создается массив текстовых данных для инициализации всей группы элементов. После создания переменной `mygroup` следует код метода `startApp()`. Первой строкой кода в методе `startApp()` инициализируется объект `groupPopur` класса `ChoiceGrop`. Конструктор этого класса мы подробно уже рассматривали, но небольших пояснений требуют два последних параметра. Оба параметра могут быть представлены в виде массива данных. Предпоследний параметр конструктора класса `ChoiceGrop`, инициализирующий объект `mygroup`, создает четыре строки текста в виде выпадающего меню (поскольку мы использовали значение `POPUP` во втором параметре конструктора класса `ChoiceGrop`). Все четыре строки текста и есть группа элементов, дающая пользователю выбор конкретных действий. Последний параметр в конструкторе класса `ChoiceGrop` служит для загрузки каждому элементу группы своего изображения или иконки, которая будет отображаться слева от текста, назначенного для каждого элемента группы. Поскольку изображения вы еще загружать не умеете (чему мы, безусловно, научимся), то надо выставить это значение в `null`. После инициализации объекта `groupPopur` создается форма на основе класса `Form`, добавляются команды выхода и выбора, и самое главное, происходит встраивание объекта `groupPopur` класса `ChoiceGrop` в форму класса `Form`. После чего текущий экран дисплея отображается посредством строки кода:

```

mydisplay.setCurrent(myform);

```

Последующие действия всей программы сводятся к обработке событий, возникающих при нажатии клавиш телефона. Если посмотреть на эту программу на экране телефона, то мы увидим на дисплее строку текста и выпадающее меню с четырьмя элементами. Выбрав один элемент из группы с помощью кнопки **Select**, на правой клавише телефона мы получим команду **Выбрать**, благодаря которой можно будет перейти в новое окно, заданное для выбранного элемента группы.

После того как произведен выбор элемента группы и нажата клавиша с командой **Выбор**, программа попадает в обработчик событий этой команды, назначенный

для переменной `vibor`. Далее используется *метод* `getSelectedIndex()` класса `choiceGroup`, с помощью которого происходят получение индекса выбранного элемента группы и помещение результата в переменную `i`. После этого происходит сравнение полученного индекса с четырьмя значениями, заданными для каждого элемента. Соответственно после совпадения индекса и значения выбранного элемента происходят действия, заданные для этого выбора. В примере происходят создание нового экрана с информационной надписью о выбранном элементе группы, добавлением команд выхода из приложения и возврата в главное окно программы. В этой программе, в ответ на действия по выбору элемента группы, создается новый экран с объектом класса `Form`. В ваших программах это могут быть любые другие события, необходимые для решения конкретных задач.

6.2.2. Класс *DateField*

Это, пожалуй, самый простой класс из всех имеющихся в иерархии класса `Item`. С помощью класса *DateField* можно произвести *установку необходимой даты и времени*. Используемый интерфейс для отображения даты и времени элементарный, и практически все действия по установке заданных параметров даты и времени уже реализованы программно. На рис. 6.4 изображен эмулятор телефона, отображающий текущее время.

В составе класса `DateField` имеются в наличии два конструктора для создания объектов этого класса, рассмотрим их. Первый конструктор:

```
public DateField(String label,  
int mode)
```

Параметры конструктора класса `DateField`:

- `label` - строка текста;
- `mode` - с помощью этого параметра конструктора устанавливается, какой именно из компонентов класса `DateField` будет воссоздан на экране. Имеется возможность вывести дату с помощью значения `DATE` и время, задав значение `TIME`. Также можно пользоваться комбинированным способом `DATE_TIME` для отображения обоих компонентов вместе.

Второй конструктор содержит добавочный параметр и позволяет устанавливать время по часовому поясу.

```
public DateField(String label,  
int mode,  
TimeZone timeZone)
```

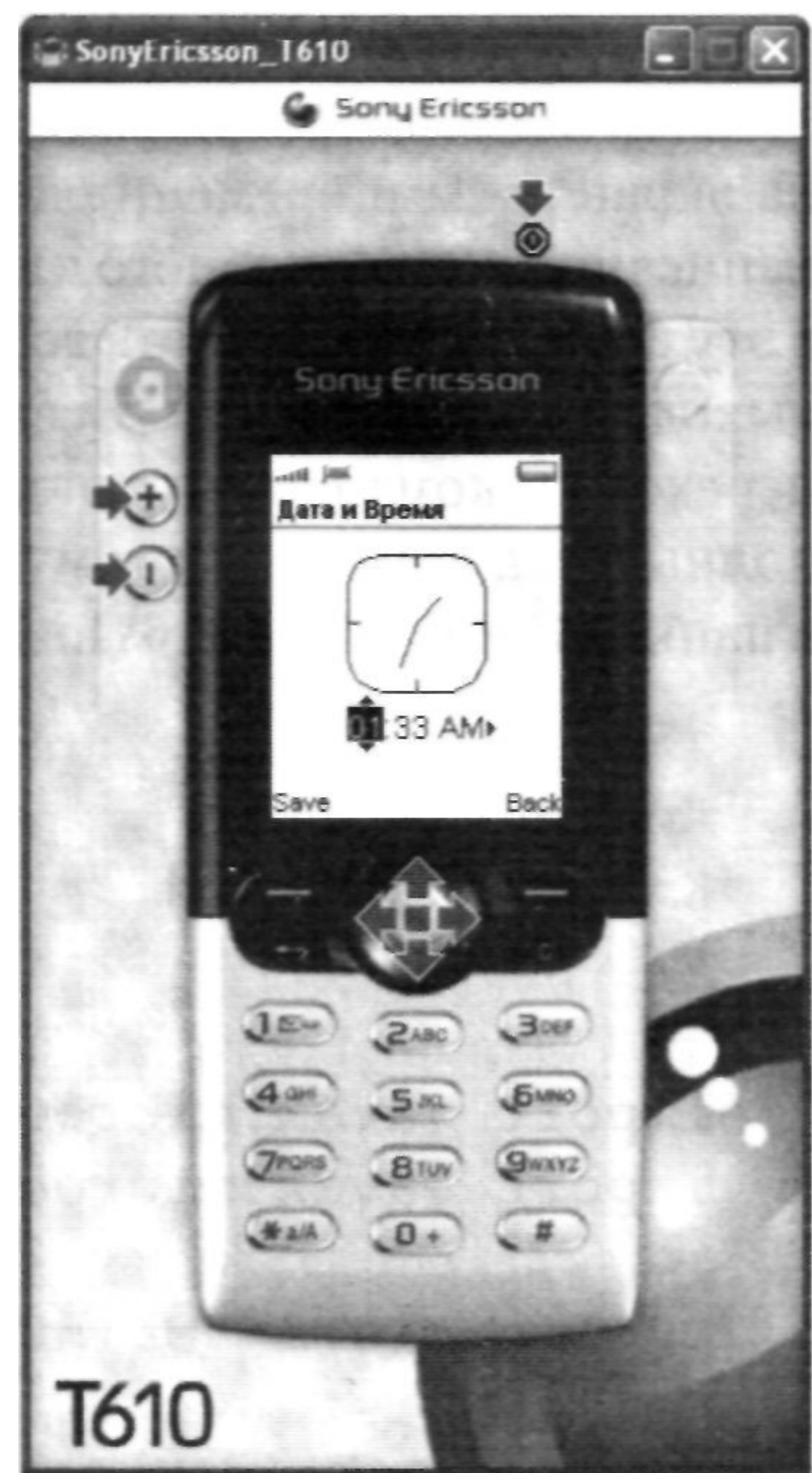


Рис. 6.4. Текущее время на экране телефона

Параметры конструктора класса `DateTimeField`:

- `label` - строка текста;
- `mode` - установка заданных компонентов класса `DateTimeField`;
- `timeZone` - это объект класса `TimeZone`, с помощью которого можно определить часовой пояс. Например:

```
TimeZone v = TimeZone.getTimeZone("GMT");
```

Класс `TimeZone` содержит всего четыре метода:

- `Date getDate()` - возвращает текущую дату;
- `void setDate(Date date)` - устанавливает новую дату;
- `int getInputMode()` - получает установленные компоненты `DATE`, `TIME` или `DATE_TIME`;
- `void setInputMode(int mode)` - устанавливает компоненты `DATE`, `TIME` или `DATE_TIME`.

Перейдем к программному коду и рассмотрим пример, реализующий вывод на экран даты и времени одновременно. Все, что сейчас от нас требуется, - это написание кода основного класса мидлета, создание пустой формы и встраивание в эту форму класса `DateTimeField`. Также необходимо проследить наличие команды выхода из приложения. Все остальное за нас сделает Java 2 ME, создав кнопки перехода и команду сохранения настроек даты и времени. В листинге 6.2 дается полный код примера к этому разделу, на компакт-диске исходный код находится в папке `\Code\Chapter6\Listing6_2\src`.

/** ЛИСТИНГ 6.2

Класс `DateTimeField`

*/

```
import javax.microedition.midlet.*;
```

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

```
public class MainClassDateTimeField extends MIDlet implements  
CommandListener
```

```
{
```

```
// команда выхода из приложения
```

```
private Command exitMidlet = new Command("Выход",
```

```
Command.EXIT, 0)
```

```
// объект класса DateTimeField
```

```
private DateTimeField dt;
```

```
// объект класса Form
```

```
private Form myform;
```

```
// объект mydisplay представляет экран телефона
```

```
private Display mydisplay;
```



```
public MainClassDateField()
{
    mydisplay = Display.getDisplay(this);
}

public void startApp()
{
    // инициализируем объект dt
    dt = new DateField(" Дата и Время ", DateField.DATE_TIME);
    // создаем форму при помощи объекта Form
    myform = new Form(" Встроенный DateField " );
    // добавить объект dt
    myform.append(dt);
    myform.addCommand(exitMidlet);
    myform.setCommandListener(this);
    // отразить текущий дисплей
    mydisplay.setCurrent(myform);
}

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if(c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
}
```

В примере создан класс `MainClassDateField`, соответствующий названию разбираемого класса. Сам по себе пример очень легкий в силу простоты реализации самого класса `DateField`. Первоначально создается объект `dt` для класса `DateField`, после этого происходит его инициализация в методе `startApp()`. Создается форма классом `Form`, и объект `dt` интегрируется в эту форму. Все остальное, а именно циферблат и календарь, показанные на рис. 6.4, создаются автоматически с помощью эмулятора телефона при выборе одного из элементов класса. В нашем примере был создан объект `dt` класса `DateField`, но можно было этого и не делать, а обойтись, например, такой простой записью:

```
Form f = new Form(new DateField(" Дата и Время ",
DateField.DATE_TIME);
```

Подобная запись используется иногда в профессиональных программах, где это действительно очевидно и не затруднит чтения и понимания всей программы в целом.

После того как вы откомпилируете этот пример и запустите приложение, на экране появятся два элемента с надписями **time** и **date**. Выбрав один из элементов и нажав на кнопку **Select**, вы попадете, в зависимости от выбора, на экран с календарем или временем, изображенный на рис. 6.4. С помощью джойстика или клавиш перемещения можно установить необходимые параметры для обоих элементов.

6.2.3. Класс *TextField*

С помощью этого класса можно создать заданный по размеру *контейнер*, в который помещается *редактируемый текст*. Этот класс обычно используется в создании адресных книг или полей для ввода текста. Кроме текста, также можно размещать любую числовую информацию. В *классе TextField* существует всего один конструктор с четырьмя параметрами, рассмотрим этот конструктор.

```
public TextField(String label,
                String text,
                int maxSize,
                int constraints)
```

Параметры конструктора класса `TextField`:

- `label` - метка, название для редактируемого поля;
- `text` - строка текста. Поле может и не содержать текст;
- `maxSize` - максимальное количество символов в поле;
- `constraints` - входное ограничение, с помощью которого можно задавать, что именно должно принимать данное поле, например цифры, буквы или символы, задается ограничение с помощью следующих констант:
 - `static int ANY` - можно вводить любой текст;
 - `static int DECIMAL` - можно вводить дробные числа;
 - `static int EMAILADDR` - используется для адреса электронной почты;
 - `static int NUMERIC` - для ввода только целого числа;
 - `static int PASSWORD` - используется при вводе пароля;
 - `static int PHONENUMBER` - для ввода телефонного номера;
 - `static int URL` - адрес сайта в Интернете.

Как видите, предусмотрены практически все варианты, остается только подставлять требуемые значения и наслаждаться простотой программирования под Java 2 ME. Использование вышеперечисленных директив в Java 2 ME традиционно, и, например, для ввода адреса сайта может быть следующая запись:

```
TextField tf = new TextField("Адрес", "", 20, TextField.URL);
```

Методы класса *TextField*

Класс `TextField` содержит 14 методов, некоторые из них мы сейчас рассмотрим.

- `void delete (int offset, int length)` - удаляет текст или заданный символ;
- `int getCaretPosition()` - получает позицию каретки для печати символов;
- `int getChars (char [] data)` - копирует текст в символьный массив данных;
- `int getMaxSize()` - определяет максимально доступное количество символов для размещения в классе `TextField`;
- `String getString()` - получает строку текста;
- `void insert(char[] data, int offset, int length, int position)` - вставляет в заданную позицию массив символьных данных;
- `void insert (String src, int position)` - вставляет в заданную позицию строку текста;
- `void setChars (char [] data, int offset, int length)` - устанавливает из символьного массива данные в заданную позицию, при этом заменяя предыдущие данные;
- `int size()` - определяет размер содержимого в `TextField` на данный момент.

Теперь перейдем непосредственно к примеру, реализующему возможности класса `TextField`. Создадим пустую форму и вставим в нее поля в виде адресной книги. В листинге 6.3 дается код всего примера.

/**ЛИСТИНГ 6.3

Класс `TextField`

*/

```
import javax.microedition.midlet.*;
```

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

```
public class MainClassTextField extends MIDlet  
implements CommandListener
```

```
{
```

```
// команда выхода из приложения
```

```
private Command exitMidlet = new  
Command("Выход", Command.EXIT, 0);
```

```
// объект класса Form
```

```
private Form myform;
```

```
// объект mydisplay представляет экран телефона
```

```
private Display mydisplay;
```

```
public MainClassTextField()
```

```
{
```

```
mydisplay = Display.getDisplay(this);
```

```
}
```



```

public void startApp()
{
    // создаем форму при помощи объекта Form
    myform = new Form(" Класс TextField " ) ;
    // добавить в форму поле для текста
    myform.append(new TextField(
        "Введите текст:", "", 20, TextField.ANY));
    // добавить в форму поле для пароля
    myform.append(new TextField(
        "Введите пароль:", "", 20, TextField.PASSWORD));
    // добавить в форму поле для e-mail
    myform.append(new TextField(
        "Введите E-mail:", "", 20, TextField.EMAILADDR));
    // добавить в форму поле для URL
    myform.append(new TextField(
        "Введите URL:", "", 20, TextField.URL));
    // добавить в форму поле для телефонного номера
    myform.append(new TextField(
        "Телефонный номер:", "", 20, TextField.PHONENUMBER));
    myform.addCommand(exitMidlet);
    myform.setCommandListener(this);
    mydisplay.setCurrent(myform);
}

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if (c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
}

```

Пример, представляющий возможности класса `TextField`, также находится на компакт-диске в папке `\Code\Chapter6\Listing6_3\src`. В листинге 6.3 создается пустая форма при помощи класса `Form` и вставляется несколько текстовых контейнеров для пароля, адреса электронной почты, Web-сайта и любой другой комбинации символов и цифр. Возьмем для наглядности первую строку кода, создающую текстовое поле для размещения символов и цифр:

Рис. 6.5. Поля класса TextField

```
myform.append(new TextField("Введите  
текст: ", "", 20, TextField.ANY));
```

Здесь используется упрощенная запись без создания объекта класса `TextField`. Первый параметр конструктора `TextField` задает информационную строку текста - метку, поясняющую назначение данного текстового поля. В следующий параметр конструктора класса `TextField`, а точнее в переменную, отвечающую за текстовый массив данных, пользователь будет вводить необходимую информацию. Значение этого параметра пустое, но возможно поместить любой текст, который впоследствии можно редактировать. Числовое значение 20 задает длину или количество введенных символов. Последний параметр использует *константу* `ANY`, дающую возможность вводить любую комбинацию символов и цифр.

Все созданные поля в листинге 6.3 используют рассмотренную выше конструкцию программного кода, и только в последнем параметре конструктора `TextField` значение варьируется для пароля, e-mail, Web-сайта и телефонного номера. Задавая различные значения последнему параметру при создании объекта этого класса, вы можете создать набор необходимых полей. На рис. 6.5 изображен эмулятор, показывающий несколько полей класса `TextField`.



6.2.4. Класс *StringItem*

Рассматриваемый класс позволяет *интегрировать в форму строку текста*, состоящую из двух частей - метки и заданного текста. Строка текста, выводимая на экран, не может быть изменена или отредактирована - это статический текст, жестко заданный в параметрах конструктора *класса* `StringItem` при создании объекта этого класса. Имеются два конструктора класса `StringItem`, разберем их устройство.

```
public StringItem(String label, String text)
```

Параметры конструктора класса `StringItem`:

- `label` - метка для строки текста;
- `text` - строка текста.

Второй конструктор класса `StringItem` имеет три параметра и позволяет выбирать способ отображения текстовой информации.

```
public StringItem(String label,
    String text,
    int appearanceMode)
```

Параметры конструктора `StringItem`:

- `label` - метка для строки текста;
- `text` - строка текста;
- `appearanceMode` - этот параметр содержит большое количество предустановленных значений, используя которые вы сможете отформатировать текст, например поместив его в кнопку и создав при этом команду, реагирующую на нажатие данной кнопки. Значения, устанавливающие вышеперечисленные действия, содержатся в пакете `javax.microedition.lcdui.Item`, рассмотрим несколько из них:
 - `BUTTON` - создает кнопку с текстом;
 - `HYPERLINK` - создает гиперссылку;
 - `LAYOUT_BOTTOM` - выравнивание к нижней части экрана;
 - `LAYOUT_CENTER` - выравнивание по центру экрана;
 - `LAYOUT_TOP` - выравнивание к верхней части экрана;
 - `LAYOUT_LEFT` - выравнивание к левой части экрана;
 - `LAYOUT_RIGHT` - выравнивание к правой части экрана.

При создании примера к классу `StringItem` обязательно воспользуемся некоторыми значениями для параметра `appearanceMode` в конструкторе класса `StringItem`.

Методы класса `StringItem`

- `int getAppearanceMode()` - возвращает заданный способ отображения текста на экране;
- `Font getFont()` - получает шрифт текста;
- `String getText()` - получает текст для класса `StringItem`;
- `void setFont(Font font)` - устанавливает шрифт текста;
- `void setPreferredSize(int width, int height)` - задает ширину и высоту текста;
- `void setText(String text)` - устанавливает текст для класса `StringItem`.

Пример, который будет предложен для класса `StringItem`, создаст форму при помощи класса `Form` и разместит в форме текст. Первая строка текста выполнена в виде простой статической надписи, вторая сделана как гиперссылка. Выделив эту строку текста и нажав кнопку на телефоне **Перейти**, вы попадете на экран с новой формой. А последняя, третья строка текста выполнена просто в виде кнопки. Рассмотрим листинг 6.4, иллюстрирующий работу данного примера.

```
/**
```

Листинг 6.4

Класс `StringItem`

```
*/
```



```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class MainClassStringItem extends MIDlet
implements CommandListener, ItemCommandListener
{
    // команда выхода из приложения
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0);
    // команда перехода по нажатию кнопки
    private Command perexodButton = new
    Command("Дальше", Command.ITEM, 1);
    // команда перехода по гиперссылке
    private Command perexodHyperlink = new Command("Перейти",
    Command.ITEM, 1);
    // команда возврата в основное окно
    private Command vozvrat = new Command("Назад",
    Command.BACK, 1);
    // объект класса Form
    private Form myform;
    // объект mydisplay представляет экран телефона
    private Display mydisplay;

    public void startApp()
    {
        mydisplay = Display.getDisplay(this);
        myform = new Form(" Класс StringItem");
        StringItem s1 = new StringItem("Метку", "Текст");
        myform.append(s1);
        // создать гиперссылку
        StringItem s2 = new StringItem("Гиперссылка",
        "www.dmk.ru", Item.HYPERLINK);
        s2.setDefaultCommand(perexodHyperlink);
        s2.setItemCommandListener(this);
        myform.append(s2);
        // создать текст в виде кнопки
        StringItem s3 = new
            StringItem("Кнопка", "Опций", Item.BUTTON);
        s3.setDefaultCommand(perexodButton);
        s3.setItemCommandListener(this);
        myform.append(s3);
        myform.addCommand(exitMidlet);
        myform.setCommandListener(this);
        mydisplay.setCurrent(myform);
    }
}
```

```
protected void destroyApp(boolean unconditional) {}
protected void pauseApp() {}
// обработчик класса ItemCommandListener
public void commandAction(Command c, Item i)
{
    // переход в окно опций
    if (c == perexodButton)
    {
        Form f1 = new Form(" Опции " ) ;
        f1.append(" Необходимые Опции " ) ;
        f1.addCommand(exitMidlet);
        f1.addCommand(vozvrat);
        f1.setCommandListener(this);
        mydisplay.setCurrent(f1);
    }
    // переход по гиперссылке
    if (c == perexodHyperlink)
    {
        Form f2 = new Form(" Издательство ДМК " ) ;
        f2.append(" Сайт издательства ДМК " ) ;
        f2.addCommand(exitMidlet);
        f2.addCommand(vozvrat);
        f2.setCommandListener(this);
        mydisplay.setCurrent(f2);
    }
}

public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if(c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
    // возврат в основную форму
    if(c == vozvrat) mydisplay.setCurrent(myform);
}
}
```

В коде листинга 6.4 для наглядности не использовался конструктор основного класса мидлета `MainClassStringItem`, но добавлялся, как уже упоминалось, новый интерфейс `ItemCommandListener` для установки обработки команд перехода в приложении. В методе `startApp()` происходят создание пустой формы для класса `Form` и интеграция класса `StringItem`. В строке кода

```
StringItem s1 = new StringItem("Метки", "Текст");
```

создается простой статический текст и выводится на дисплей телефона. Следующий блок кода:

```
StringItem s2 = new StringItem("Гиперссылка",  
                                "www.dmk.ru", Item.HYPERLINK);  
s2.setDefaultCommand(perexodHyperlink);  
s2.setItemCommandListener(this);  
myform.append(s2);  
StringItem s3 = new  
StringItem("Кнопка", "Опций", Item.BUTTON);  
s3.setDefaultCommand(perexodButton);  
s3.setItemCommandListener(this);  
myform.append(s3);
```

формирует текст на экране телефона, назначив для него обработчик событий при помощи метода `setItemCommandListener()`. Можно получить текст в виде активной ссылки. При создании объекта `s2` класса `StringItem` использовался конструктор с тремя параметрами. Последний параметр этого конструктора как раз и отвечает за вид создаваемой ссылки. Была создана гиперссылка с помощью константы `HYPERLINK`. Блоком кода с объектом `s3` уже создавалась кнопка. Эта кнопка является также простым статическим текстом, но оформленным в виде прямоугольной кнопки. Объекту `s3` также назначается обработчик событий методом `setItemCommandListener()`, благодаря чему и получается активная ссылка. Выбрав ее, можно перейти в нужное место в приложении.

Теперь наша программа имеет два одноименных обработчика событий с разными параметрами, представленными двумя интерфейсами - `CommandListener` и `ItemCommandListener`. Обработчик событий, созданный при помощи метода `commandAction(Command c, Item i)`, следит за двумя активными ссылками, выполненными в виде гиперссылки и кнопки. Выбрав одну из активных ссылок и воспользовавшись соответственной командой перехода `perexodButton` - для кнопки и `perexodHyperlink`, вы попадете на экран с новой формой и информационной надписью. Оба новых экрана созданы классом `Form`, где также имеются две команды: `exitMidlet` - для выхода из приложения и `vozvrat` - для возврата в основное окно. Эти две команды обрабатываются своим методом `commandAction(Command c, Displayable d)` интерфейса `CommandListener`. Для того чтобы создать активную ссылку, необходимо воспользоваться интерфейсом `ItemCommandListener`, реализовав метод `commandAction()` для обработки необходимых событий. Рисунок 6.6 показывает экран эмулятора с несколькими элементами класса `StringItem`.



Рис. 6.6. Элементы класса `StringItem`

6.2.5. Класс *Spacer*

Класс *Spacer* подвигает элемент на экране телефона, создавая тем самым свободное пространство с указанными размерами. Именно за *создание свободного пространства* на экране отвечает класс *Spacer*. При создании объекта класса используется один конструктор с двумя параметрами, при помощи которых задается создаваемое пространство на экране. Конструктор класса *Spacer* выглядит следующим образом:

```
public Spacer (int minWidth, int minHeight)
```

Параметры конструктора *Spacer*:

- `minWidth` - ширина в пикселях;
- `minHeight` - высота в пикселях.

Класс *Spacer* имеет четыре метода, все они просты и не нуждаются в пояснениях, в *приложении 2* находится справочник по платформе Java 2 ME, в котором вы сможете найти описание существующих методов класса *Spacer*. Чтобы показать работу класса *Spacer*, рассмотрим простой пример, где создается область в 50 пикселей по ширине и 0 по высоте, благодаря чему элемент, размещенный в форме, сдвигается на указанное пространство вправо. В качестве элемента, встроенного в форму, используется класс *Text Field*. В листинге 6.5 дается исходный код примера, который можно найти на компакт-диске в папке `\Code\ Chapter6\Listing6_5\src`.

```
/**
Листинг 6.5
Класс Spacer
*/

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MainClassSpacer extends MIDlet implements
CommandListener
{
    // команда выхода из приложения
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0);
    // объект класса DateField
    private Spacer sp;
    // объект класса Form
    private Form myform;
    // объект mydisplay представляет экран телефона
    private Display mydisplay;

    public MainClassSpacer()
```

```
{
mydisplay = Display.getDisplay(this);
}

public void startApp()
{
    // инициализируем объект sp
    sp = new Spacer(50,0);
    // создаем форму при помощи объекта Form
    myform = new Form(" Класс Spacer " );
    // добавить объект sp
    myform.append(sp);
    myform.append(new
        TextField("Метки", "Текст", 20, TextField.ANY));
    myform.addCommand(exitMidlet);
    myform.setCommandListener(this);
    // отразить текущий дисплей
    mydisplay.setCurrent(myform);
}

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if(c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
}
```

В листинге 6.5 создаются рабочий класс `MainClassSpacer` и форма на основе класса `Form`. Объявляется объект `sp` для класса `Spacer` и инициализируется в методе `startApp()`. При инициализации объекта `sp` используются два значения для параметров, создавая тем самым пустое пространство с левой стороны от текстового поля, созданного при помощи класса `TextField`. Эмулятор, изображенный на рис. 6.7, показывает работу программы из листинга 6.5.

Класс `Spacer` был добавлен в Java 2 ME для профиля MIDP 2.0, нельзя сказать, что этот элемент жизненно необходим, но бывают случаи, когда использование класса `Spacer` облегчает работу программиста.



Рис. 6.7. Пространство, созданное классом `Spacer`

6.2.6. Класс *ImageItem*

С помощью класса *ImageItem* возможна загрузка изображения в форму, представленную классом *Form*. Изображением может быть любая картинка формата PNG (Portable Network Graphics - формат портативной сетевой графики), выполненная в виде иконки, фотографии, заставки, фона и т. д. Имеются два конструктора класса *ImageItem*. Первый конструктор содержит четыре параметра, рассмотрим этот конструктор:

```
public ImageItem(String label,
Image img,
int layout,
String altText)
```

Параметры конструктора *ImageItem*:

- `label` - метка;
- `img` - объект класса *Image*, содержащий изображение;
- `layout` - форматирует загружаемое изображение на экране телефона с помощью использования следующих директив:

- `public static final int LAYOUT_DEFAULT` - размещение изображения по умолчанию;
- `public static final int LAYOUT_LEFT` - размещение изображения со сдвигом к левой стороне экрана;
- `public static final int LAYOUT_RIGHT` - размещение изображения со сдвигом к правой стороне экрана;
- `public static final int LAYOUT_CENTER` - размещение изображения со сдвигом к центру экрана;
- `altText` - информационный текст, используемый взамен загружаемого изображения. Если текст не используется - этот параметр нужно установить в значение `null`.

Второй конструктор класса `ImageItem` имеет на один параметр больше и выглядит следующим образом:

```
public ImageItem(String label,  
                 Image img,  
                 int layout,  
                 String altText  
                 int appearanceMode)
```

Параметры конструктора `ImageItem`:

- `label` - метка;
- `img` - объект класса `Image`, содержащий изображение;
- `layout` - форматирование загружаемого изображения на экране телефона;
- `altText` - текст, использующийся взамен загружаемого изображения;
- `appearanceMode` - этот параметр содержит ряд значений:
 - `BUTTON` - создает кнопку с текстом;
 - `HYPERLINK` - создает гиперссылку;
 - `LAYOUT_BOTTOM` - выравнивание к нижней части экрана;
 - `LAYOUT_CENTER` - выравнивание по центру экрана;
 - **`LAYOUT_TOP`** - выравнивание к верхней части экрана;
 - `LAYOUT_LEFT` - выравнивание к левой части экрана;
 - `LAYOUT_RIGHT` - выравнивание к правой части экрана.

С помощью этих значений можно создать активную ссылку и оформить изображение в виде кнопки или гиперссылки. В *разделе 4.9* при рассмотрении класса `StringItem` мы уже сталкивались с этими значениями, создавая статический текст в виде кнопки и гиперссылки.

При загрузке изображений с помощью класса `ImageItem` существует ряд нюансов, на которые необходимо обратить внимание. Класс `ImageItem` является подклассом класса `Image`, прежде чем воспользоваться классом `ImageItem`, необходимо создать объект класса `Image`. Затем поместить или загрузить в объект класса `Image` изображение и только потом воспользоваться классом `ImageItem` для размещения на экране изображения, представленного объектом класса `Form`. Создавая объект класса `ImageItem`, вы создаете своего рода контейнер для

содержания ссылки на объект `Image`. Рассмотрим небольшой фрагмент кода, иллюстрирующий создание и загрузку изображения:

```
Image a = Image.createImage("/risunok.png" ) ;
ImageItem b = new ImageItem("Рисунок", a,
ImageItem.LAYOUT_CENTER,null) ;
```

Первым делом создается объект `i` класса `Image`, после чего происходит загрузка необходимого изображения посредством вызова *метода* `createImage()` класса `Image`. Далее создается объект `im` класса `ImageItem`, который будет содержать ссылку на объект `image`.

Изображение, загружаемое в приложение, может находиться в любом месте рабочего каталога. При использовании, например, J2ME Wireless Toolkit изображение лучше поместить в папку `\res`. Данная папка по умолчанию для файлов ресурса к разрабатываемому приложению, и в этом случае запись `\risunok.png` будет обращаться к папке `\res`. Если вы хотите использовать другую папку, то необходимо указать весь путь при загрузке изображения, например:

```
Image ikon1 = Image.createImage("/Ikon/Level2/ikon1.png") ;
```

Рассмотрим пример загрузки изображения на экран в виде фона. В качестве изображения послужит фотография автора этой книги, которую мы загрузим и выведем на экран телефона. Код примера содержится в листинге 6.6 и папке `\Code\Chapter6\Listing6_6\src` на компакт-диске.

```
/**
```

```
ЛИСТИНГ 6.6
```

```
Класс ImageItem
```

```
*/
```

```
import javax.microedition.midlet.*;
```

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

```
public class MainClassImageItem extends MIDlet implements
CommandListener
```

```
{
```

```
// команда выхода из приложения
```

```
private Command exitMidlet = new Command("Выход",
Command.EXIT, 1) ;
```

```
// объект класса Form
```

```
private Form myform = new Form(" Изображение " ) ;
```

```
// объект mydisplay представляет экран телефона
```

```
private Display mydisplay;
```

```
public MainClassImageItem()
```

```
{
```

```
mydisplay = Display.getDisplay(this);
}

public void startApp()
{
    // перехватываем исключительную ситуацию
    try{
        // загрузка изображения
        Image image = Image.createImage("/gornakov.png");
        // создаем объект класса ImageItem
        ImageItem im = new ImageItem(" Фотография ",
            image, ImageItem.LAYOUT_CENTER, "");
        // добавляем изображение в форму
        myform.append(im);
    } catch(java.io.IOException ex){ }
    // установка обработчика событий для Form
    myform.addCommand(exitMidlet);
    myform.setCommandListener(this);
    // отразить текущий дисплей
    mydisplay.setCurrent(myform);
}

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if (c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
```

Пример достаточно прост: происходит загрузка на экран телефона изображения, представленного классом `Form`, с добавлением команды выхода из приложения. Но после компиляции листинга 6.6 и запуска приложения на эмуляторе J2ME Wireless Toolkit возникают цветовые дефекты в виде некачественного отображения фотографии. Это вызвано прежде всего минимальной цветовой гаммой, представляемой эмулятором J2ME Wireless Toolkit. Протестируйте код из листинга 6.6 на различных эмуляторах, рассмотренных в *главе 4*. На рис. 6.8 показан эмулятор с изображением на экране фотографии.

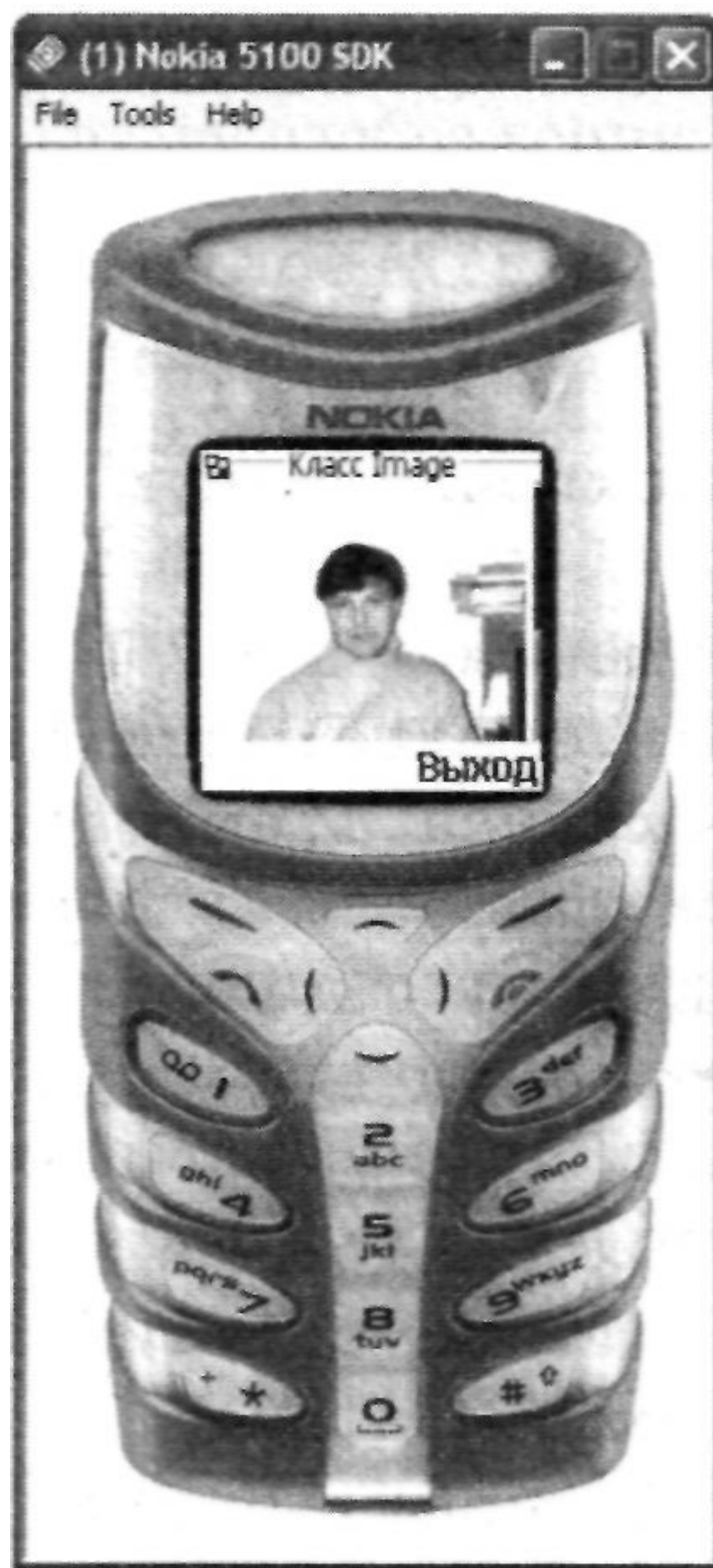


Рис. 6.8. Изображение, загруженное при помощи класса ImageItem

6.2.7. Класс Gauge

С помощью класса *Gauge* создается графический измеритель различных процессов. То есть можно осуществить графическое отображение, например, процесса загрузки файла, сохранения игры, поиска информации и т. д. Представление любого из процессов в графическом виде дает возможность создать красивое интерактивное приложение. Визуальное отображение процесса осуществляется в виде заданного по размеру горизонтального столбца, который закрашивается слева направо по мере выполнения процесса. К сожалению, определенного стандарта в графическом представлении, скажем того же столбца, не существует, и каждый из производителей представляет свой разработанный вид графического контекста. На рис. 6.9 изображен эмулятор телефона с графическим измерителем процесса.

Класс *Gauge* имеет всего один конструктор, необходимый при создании объекта этого класса. Разберем конструктор класса *Gauge*:



Рис. 6.9. Эмуляторы телефонов, показывающие использование класса Gauge

```
public Gauge(String label,  
             boolean interactive,  
             int maxValue,  
             int initialValue)
```

Параметры конструктора Gauge:

- `label` - метка или название процесса, связанного с объектом Gauge;
- `interactive` - имеются два значения: `true` для интерактивного режима и `false` - для неинтерактивного режима;
- `maxValue` - максимальное значение, задающее диапазон длительности всего процесса. Может быть установлено при помощи значения `INDEFINITE`;
 - `static int INDEFINITE`-специальное значение, устанавливающее максимальную величину при неизвестном диапазоне течения всего процесса;
- `initialValue` - параметр может быть инициализирован значением от нуля и до значения в параметре `maxValue`. Этим значением инициализируется начальный отсчет, от которого происходит увеличение визуального представления работы процесса. Кроме числовых значений, возможно применение заданных констант:
 - `static int CONTINUOUS_IDLE` - задает непрерывное циклическое течение процесса для неинтерактивного режима при неопределенном диапазоне;
 - `static int CONTINUOUS_RUNNING` - задает непрерывное бегущее течение процесса для неинтерактивного режима при неопределенном диапазоне;
 - `static int INCREMENTAL_IDLE` - задает пошаговое циклическое течение процесса для неинтерактивного режима при неопределенном диапазоне;
 - `static int INCREMENTAL_UPDATING` - задает пошаговое обновление течения процесса для неинтерактивного режима при неопределенном диапазоне.

Методы класса Gauge

Методы, имеющиеся в составе класса Gauge, позволяют настраивать графическое отображение течения процесса на экране телефона, рассмотрим некоторые из методов:

- `void addCommand (Command cmd)` - добавляет команду;
- `int getMaxValue ()` - получает значение максимального диапазона работы процесса;
- `int getValue()` - получает текущее значение в процессе работы;
- `void setItemCommandListener (ItemCommandListener l)` - устанавливает обработчик событий;
- `void setLabel (String label)` - устанавливает метку для элемента;
- `void setLayout (int layout)` - устанавливает директивы для элемента;

- `void setMaxValue (int maxValue)` - устанавливает максимальное значение течения процесса;
- `void setPreferredSize (int width, int height)` - задает ширину и высоту для графического представления всего течения процесса;
- `void setValue(int value)` - устанавливает текущее значение процесса.

В примере создается простой измеритель течения процесса в виде прямоугольника, максимальный диапазон задан значением 10. В листинге 6.7 показано использование класса `Gauge`. Исходный код примера также находится на компакт-диске в папке `\Code\Chapter6\Listing6_7\src`.

```
/** ЛИСТИНГ 6.7
```

```
Класс Gauge
```

```
*/
```

```
import javax.microedition.midlet.*;
```

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

```
public class MainClassGauge extends MIDlet implements  
CommandListener
```

```
{
```

```
// команда выхода из приложения
```

```
private Command exitMidlet = new Command("Выход",  
Command.EXIT, 1);
```

```
// объект класса Form
```

```
private Form myform = new Form(" Класс Gauge " );
```

```
// объект mydisplay представляет экран телефона
```

```
private Display mydisplay;
```

```
public MainClassGauge()
```

```
{
```

```
    mydisplay = Display.getDisplay(this);
```

```
}
```

```
public void startApp()
```

```
    // добавить объект класса Gauge
```

```
    myform.append(new Gauge(" Прогресс: ", true, 10, 5 ));
```

```
    // установка обработчика событий для Form
```

```
    myform.addCommand(exitMidlet);
```

```
    myform.setCommandListener(this);
```

```
    // отразить текущий дисплей
```

```
    mydisplay.setCurrent(myform);
```

```
}
```



```
public void pauseApp() {}
public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if (c == exitMidlet)
    {
        destroyApp(false) ;
        notifyDestroyed() ;
    }
}
```

Откомпилировав этот пример, вы увидите на экране прямоугольник, наполовину закрашенный темным цветом. Использование класса `Gauge` позволяет создавать экранные заставки, отображающие, например, процесс загрузки приложения. Для того чтобы пользователь не наблюдал за черным экраном телефона в момент загрузки очередного процесса, создайте класс `Gauge` и используйте его по назначению.

6.3. Класс Alert

Использование *класса Alert* в приложениях Java 2 ME обусловлено возникновением различных *внештатных ситуаций*. В основном класс `Alert` применяется для создания экрана, который информирует пользователя об ошибке, произошедшей в приложении или любом другом уведомлении информационного характера. Экран, определенный классом `Alert`, может содержать строковое уведомление о произошедшей ошибке либо текстовую строку с заданным изображением. В связи с этим класс `Alert` имеет два конструктора, использующихся в создании объектов этого класса. Первый конструктор содержит один параметр типа `String`, задавая строку текста для уведомления. Рассмотрим первый конструктор класса `Alert`.

```
public Alert(String title)
```

Параметры конструктора `public Alert`:

- `title` - строка текста.

Второй конструктор класса `Alert` имеет уже четыре параметра, представляя более интересный вид создаваемого экрана:

```
public Alert(String title,
             String alertText,
             Image alertImage,
             AlertType alertType)
```

Параметры конструктора `public Alert`:

- `title` - название созданного экрана;
- `alertText` - текст уведомления;
- `alertImage` - изображение;
- `alertType` - тип уведомления, определяемый классом `AlertType`. Существуют пять типов уведомлений:
 - `static AlertType ALARM` - тревога;
 - `static AlertType CONFIRMATION` - предупреждение о возможном действии, которое пользователь должен произвести;
 - `static AlertType ERROR` - ошибка;
 - `static AlertType INFO` - информационное сообщение;
 - `static AlertType WARNING` - предупреждение.

Создавая объект класса `Alert`, вы можете выбрать необходимый тип уведомлений или информационных сообщений, формируя органичные, удобные приложения, предусматривающие любые варианты развития событий.

6.3.1. Методы класса `Alert`

Существует множество методов класса `Alert`, все они призваны создавать более насыщенные и информационные сообщения. Рассмотрим методы класса `Alert`.

- `void addCommand (Command cmd)` - добавляет команду;
- `int getDefaultTimeout()` - получает время для представления уведомления. Можно воспользоваться переменной `FOREVER` для постоянного представления экрана с объектом класса `Alert`;
- `Image getImage()` - получает изображение для экрана, представленного классом `Alert`;
- `Gauge getIndicator()` - этот метод позволяет воспользоваться графическим измерителем класса `Gauge`;
- `String getString()` - получает текстовую строку;
- `int getTimeout()` - получает заданное время для представления уведомления;
- `AlertType getType()` - определяет тип используемого уведомления;
- `void removeCommand (Command cmd)` - удаляет команду;
- `void setCommandListener (CommandListener l)` - устанавливает обработчик событий;
- `void setImage(Image img)` - устанавливает изображение;
- `void setIndicator(Gauge indicator)` - устанавливает индикатор измерителя для использования класса `Gauge`;
- `void setString (String str)` - устанавливает строку текста;
- `void setTimeout (int time)` - устанавливает время;
- `void setType (AlertType type)` - устанавливает тип уведомлений или информационных сообщений.

Использовать возможности класса `Alert` в приложении необходимо. Уведомления об ошибках и различные информационные сообщения улучшают

пользовательский интерфейс разрабатываемой программы. В листинге 6.8 приводится простой пример, иллюстрирующий создание и отображение класса Alert на экране телефона. Исходный код примера находится в папке `\Code\Chapter6\Listing6_8\src`.

```
/**
Листинг 6.8
Класс Alert
*/

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MainClassAlert extends MIDlet implements
CommandListener

// команда выхода из приложения
private Command exitMidlet = new Command("Выход",
Command.EXIT, 1);
// объект класса Alert
Alert al;
// объект mydisplay представляет экран телефона
private Display mydisplay;

public MainClassAlert()

    mydisplay = Display.getDisplay(this);
}

public void startApp()

    // перехватываем исключительную ситуацию
    try{
        // загрузка изображения
        Image image = Image.createImage("/error.png" );
        // объект класса Alert
        al = new Alert(" Класс Alert ",null, image,
            AlertType.ERROR);
    } catch(Java.io.IOException ex){ }
    al.addCommand(exitMidlet);
    al.setCommandListener(this);
    mydisplay.setCurrent(al);
}

public void pauseApp() {}
```



```
public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if (c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
}
```

В листинге 6.8 создается класс `MainClassAlert`, являющийся основным классом мидлета. В самом начале всего кода происходит объявление необходимых переменных и, в частности, объекта `al` класса `Alert`. В методе `startApp()` создается объект класса `Image`, в котором будет содержаться загружаемое изображение. Изображение выполнено в виде информационной надписи об ошибке и находится в папке `\Code\Chapter6\Listing6_8\res` на прилагаемом к книге компакт-диске. На рис. 6.10 изображена работа класса `Alert`.



Рис. 6.10. Информационное уведомление, созданное при помощи класса `Alert`

6.4. Класс List

Класс List не входит в иерархию класса Item. Использование класса List дает возможность *создавать выбираемый список элементов*, отображаемый на экране в виде одной или нескольких строк текста. Класс List наследуется от класса Screen и реализует возможности интерфейса Choice. При создании выбираемого списка элементов необходимо указать тип создаваемого списка. Существует всего три типа списков, реализация которых основана на использовании интерфейса Choice:

- EXCLUSIVE - предоставляет эксклюзивный выбор элемента в списке;
- MULTIPLE - множественный выбор элементов из списка;
- IMPLICIT - выбирает из списка только один элемент, на котором сфокусировал свое внимание пользователь.

Конструкция применения типов EXCLUSIVE и MULTIPLE напоминает использование этих типов в классе ChoiceGroup, а вот применение типа IMPLICIT возможно только с использованием класса List. При создании объекта класса List можно воспользоваться двумя видами конструкторов. Рассмотрим их более подробно.

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

Параметры конструктора List:

- title - название создаваемого списка элементов;
- listType - тип создаваемого списка, может быть одним из трех значений: IMPLICIT, EXCLUSIVE и MULTIPLE.

Этот конструктор с двумя параметрами создает пустой список с заданным типом в параметре listType. Второй конструктор класса List несколько сложнее. Он состоит из четырех параметров и создает многострочный список элементов с загрузкой иконки или изображения для каждого элемента.

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

Параметры конструктора List:

- title - название создаваемого списка элементов;
- listType - может быть одним из трех значений: IMPLICIT, EXCLUSIVE и MULTIPLE - для определения типа создаваемого списка элементов;
- stringElements - в этом параметре используется массив строк для создания списка элементов;
- imageElements - с помощью этого параметра каждому из элементов можно загрузить свое изображение, чаще всего используются иконки маленьких размеров, например 10x10 пикселей.

6.4.1. Методы класса *List*

Класс `List` имеет множество методов, с помощью которых можно производить редакцию списка элементов, выбор заданного элемента и многое другое. Разберем часть методов класса `List`:

- `int append(String stringPart, Image imagePart)` - добавление списка элементов;
- `void delete (int elementNum)` - удаление заданного элемента из списка;
- `void deleteAll ()` - удаление всех элементов;
- `Font getFont (int elementNum)` - получает шрифт для заданного элемента в списке;
- `Image getImage (int elementNum)` - получает изображение для заданного элемента в списке;
- `int getSelectedFlags (boolean[] selectedArray_return)` - возвращает состояние всех элементов в виде массива данных;
- `int getSelectedIndex()` - получает выбранный индекс элемента в списке;
- `String getString (int elementNum)` - получает строку текста для выбранного элемента из списка;
- `void insert (int elementNum, String stringPart, Image imagePart)` - вставляет элемент в список до указанного номера элемента в списке;
- `boolean isSelected (int elementNum)` - получает выбранный элемент из списка;
- `void removeCommand (Command cmd)` - удаляет команду для списка;
- `void set (int elementNum, String stringPart, Image imagePart)` - вставляет новый элемент в список взамен предшествующего;
- `void setFont (int elementNum, Font font)` - устанавливает шрифт заданному элементу в списке;
- `void setSelectCommand (Command command)` - этот метод предназначен для работы с типом `IMPLICIT`. Когда используется такой тип списка, то выбирается элемент, на котором сфокусирована в данный момент строка состояния. Этот метод позволяет определить, на каком элементе сфокусировано внимание пользователя. При этом используется такая запись: `List.SELECT_COMMAND` для определения выбранного элемента в списке;
- `void setSelectedFlags (boolean[] selectedArray)` - устанавливает состояние выбранных элементов;
- `void setSelectedIndex (int elementNum, boolean selected)` - устанавливает индекс выбранного элемента в списке;
- `void setTitle (String s)` - добавляет название в список элементов;
- `int size()` - с помощью этого метода можно узнать количество элементов в списке.

Теперь давайте создадим пример, описывающий основные возможности класса `List`. Класс `List` может создавать три списка элементов: `Exclusive`, `Multiple` и `Implicit`. Используем эту возможность и создадим код, реализующий

все три типа. Основная идея создания примера для класса `List` сводится к следующему: при входе в приложение пользователь попадает в главное окно со списком из двух элементов `Multiple` и `Implicit`, а сам список этих двух элементов будет создан на основе типа `Exclusive`. Ко всем элементам списка будут загружаться свои иконки.

Выбрав один из двух элементов списка курсором, пользователь должен нажать клавишу команды **Выбор** для перехода в программе. Оба элемента списка `Multiple` и `Implicit` будут представлять два разных типа списка. Выбрав один из элементов `Multiple` или `Implicit`, пользователь попадает на новый экран. Каждый из выбранных списков будет содержать ряд элементов, иллюстрирующих работу типов `Multiple` и `Implicit`. Выбирая элементы из этих списков, пользователь будет получать информационное сообщение. В листинге 6.9 показан исходный код примера, а на компакт-диске исходный код размещается в папке `\Code\Chapter6\Listing6_9\src`.

```
/**
Листинг 6.9
Класс List
*/

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MainClassList extends MIDlet implements
CommandListener
{
    // команда выхода из приложения
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0);
    // команда выбора элемента из списка
    private Command vibor = new Command("Выбор",
    Command.SCREEN, 1);
    // команда возврата в главное окно
    private Command vozvrat = new Command("Назад",
    Command.BACK, 1);
    // команда выбора элемента для типов Implicit и Multiple
    private Command OK = new Command("OK", Command.OK, 1);
    // массив иконок для типа EXCLUSIVE
    Image[] iconEx = null;
    // массив иконок для типа Multiple
    Image[] iconMu = null;
    // массив иконок для типа Implicit
    Image[] iconIm = null;
    // объект класса List для типа EXCLUSIVE
```

```
private List mylistEx;
// объект класса List для типа Multiple
private List mylistMu;
// объект класса List для типа Implicit
private List mylistIm;
// объект mydisplay представляет экран телефона
private Display mydisplay;

public MainClassList()
{
mydisplay = Display.getDisplay(this);
}

public void startApp()
{
    // перехватываем исключительную ситуацию
    try{
        // загрузка изображения
        Image image1 = Image.createImage("/iconMu.png " ) ;
        Image image2 = Image.createImage("/iconIm.png " ) ;
        // поместить загруженные изображения в массив iconEx
        iconEx = new Image[]
        {
            image1, image2
        };
        // загрузка изображения
        Image image3 = Image.createImage("/Multiple.png " ) ;
        // поместить загруженные изображения в массив iconMu
        iconMu = new Image[]{image3, image3, image3, image3};
        // загрузка изображения
        Image image4 = Image.createImage("/Implicit.png " ) ;
        // поместить загруженные изображения в массив iconIm
        iconIm = new Image[]{image4, image4, image4};
    } catch (Java.io.IOException ex){    }
    // текст для двух элементов списка
    String[] st = {" Тип Multiple", " Тип Implicit"};
    // инициализация объекта mylistEx
    mylistEx = new List(" Тип EXCLUSIVE ",
        Choice.EXCLUSIVE, st, iconEx);
    // добавить команды
    mylistEx.addCommand(exitMidlet);
    mylistEx.addCommand(vibor);
    mylistEx.setCommandListener(this);
    // отразить текущий дисплей
```

```
mydisplay.setCurrent(mylistEx);
}

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if (c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
    // возврат в главное окно
    if (c == vozvrat)
        Display.getDisplay(this).setCurrent(mylistEx);
    // обработка команды OK
    if (c == OK)
    {
        Alert al = new Alert(null, " Информационное уведомле-
            ние", null, null);
        mydisplay.setCurrent(al);
    }
    // обработка команды vibor
    if (c == vibor)
    {
        // взять индекс выбранного элемента
        int i = mylistEx.getSelectedIndex();
        // события для элемента "Тип Multiple"
        if (i == 0)
        {
            // текст для элементов списка
            String[] string = {" Меч", " Щит", " Нож", " Копье" };
            // инициализация объекта mylistMu
            mylistMu = new List(" Тип MULTIPLE ", Choice.MULTIPLE,
                string, iconMu);
            // добавить команду возврата .
            mylistMu.addCommand(vozvrat);
            // добавить команду OK
            mylistMu.addCommand(OK);
            mylistMu.setCommandListener(this);
            // отразить текущий дисплей
            mydisplay.setCurrent(mylistMu);
        }
    }
}
```



```

    }
    // события для элемента "Тип Implicit"
    if(i == 1)
    {
        // текст для элементов списка
        String[] string = {" Звук", " Видео", " Управление"};
        // инициализация объекта mylistIm
        mylistIm = new List(" Тип IMPLICIT ", Choice.IMPLICIT,
                           string, iconIm);
        // добавить команду возврата
        mylistIm.addCommand(vozvrat);
        // добавить команду ОК
        mylistIm.addCommand(OK);
        mylistIm.setCommandListener(this);
        // отразить текущий дисплей
        mydisplay.setCurrent(mylistIm);
    }
}
}

```

В листинге 6.9 создан класс `MainClassList`, являющийся основным классом мидлета программы. В начале исходного кода создаются команды для выхода из приложения - `exitMidlet`, для выбора элемента из списка - `vibor`, для возврата в главное окно приложения - `vozvrat` и команда ОК, обрабатывающая выбранный элемент из группы. За командами обработки событий следует объявление трех переменных: `iconEx`, `iconMu` и `iconIm`. Все три переменные будут содержать массив изображений или иконок для трех рассматриваемых в этом примере типов: `Exclusive`, `Multiple` и `Implicit` класса `List`. Затем в коде

```

private List mylistEx;
private List mylistMu;
private List mylistIm;
private Display mydisplay;

```

создаются три объекта класса `List`, представляющие три имеющихся типа элементов списка и объект `mydisplay` класса `Display`. Метод `startApp()` производит загрузку всех имеющихся иконок из папки `\Code\Chapter6\Listing6_9\res` с помощью метода `createImage` класса `Image`. Все загруженные иконки содержатся в переменных `image1`, `image2`, `image3` и `image4`. При загрузке изображений используется конструкция `try{}catch(){}` для перехвата исключительной ситуации. Все иконки размещаются в массивах `iconEx`, `iconMu` и `iconIm` для каждого типа элементов списка. В строке кода

```

mylistEx = new List(" Тип EXCLUSIVE ", Choice.EXCLUSIVE,
st, iconEx);

```

происходит инициализация объекта `mylistEx`. Используется конструктор класса из четырех параметров. Первый параметр конструктора класса `List` создает заголовок для всего экрана. Во втором параметре конструктора используется значение `Choice.EXCLUSIVE`. С помощью этого значения создается список элементов типа `Exclusive`, позволяющий выбрать только один элемент из всего списка. Третий параметр в конструкторе класса `List` принимает значение переменной `st`. Эта переменная содержит две строки текста, создавая тем самым только два элемента списка. Последний параметр загружает две иконки для обоих элементов списка.

В методе `commandAction()` происходит обработка всех имеющихся команд, созданных в приложении. Команда `exitMidlet` производит выход из приложения. Команда `vozvrat` возвращает пользователя в главное окно программы. Команда `ОК` показывает информационное сообщение, выполненное на основе класса `Alert`. Команда `vibor` осуществляет переход в выбранный экран, представленный списком элементов двух различных типов `Multiple` и `Implicit` класса `List`. С помощью метода `getSelectedIndex()` берется индекс выбранного элемента из списка, и на его основе в конструкции `if/else` происходит обработка выбранных событий. Два типа списков `Multiple` и `Implicit` создаются подобно списку типа `Exclusive`. Рисунок 6.11 изображает эмулятор, на экране которого воспроизводится список элементов, организованный с помощью класса `List`.

В мобильных приложениях очень часто используются различные списки элементов, поэтому необходимо изучить возможности класса `List` более внимательно.

6.5. Класс Ticker

Объект класса *Ticker* служит для создания в приложении подобия *бегущей строки*, располагающейся в верхней части экрана. Текст, выводимый на экран объектом класса `Ticker`, перемещается справа налево с одинаковой скоростью. При достижении конца текста бегущая строка появляется заново, обеспечивая тем самым цикличность перемещения текста. На рис. 6.12 изображен эмулятор с бегущей строкой в верхней части экрана.

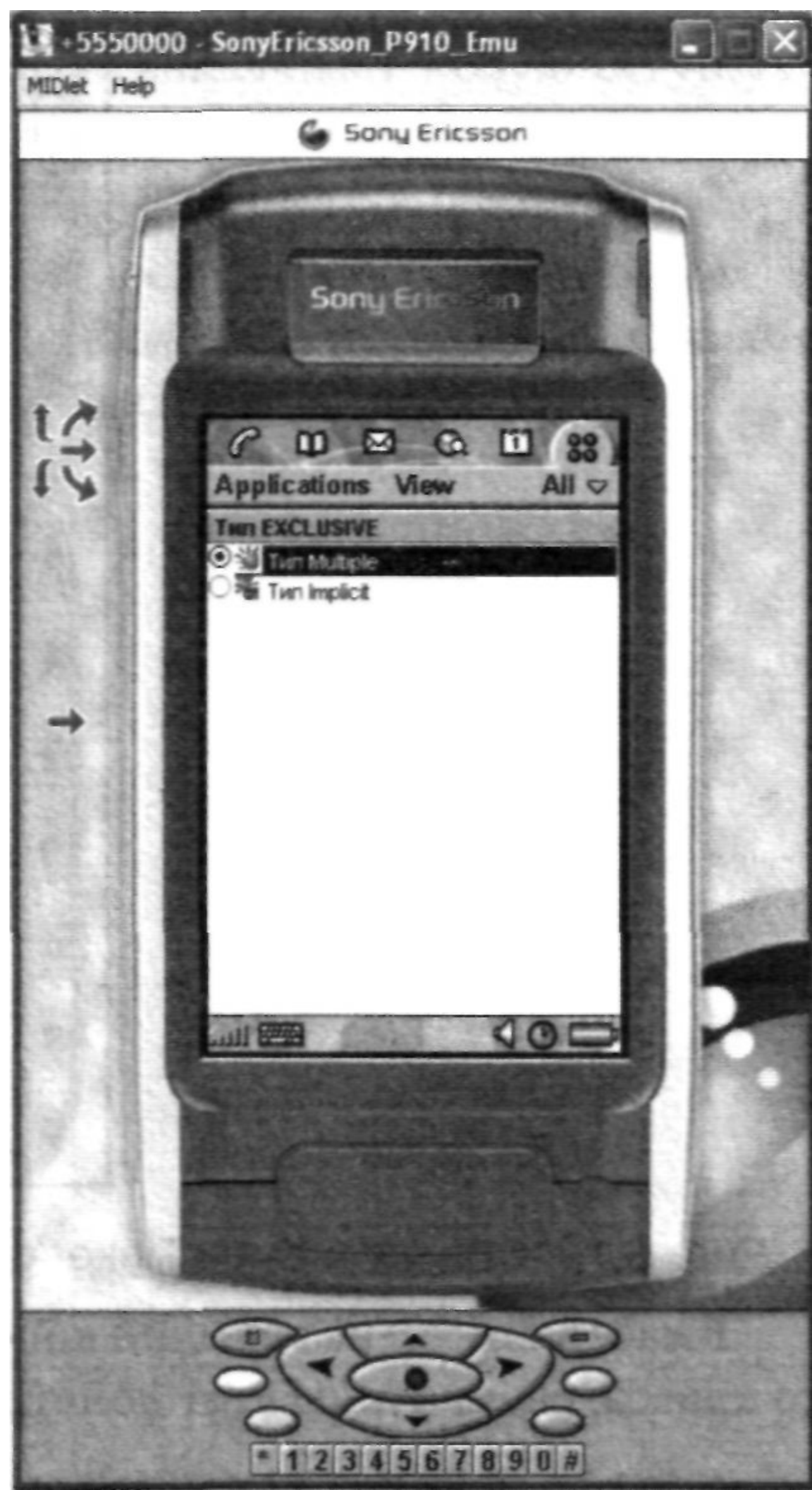


Рис. 6.11. Список элементов, созданный классом `List`



Рис. 6.12. Объект класса `Ticker` создает в верхней части экрана бегущую строку

Класс `Ticker` имеет один конструктор, необходимый в создании объекта этого класса, рассмотрим этот конструктор:

```
public Ticker(String str)
```

Параметры конструктора класса `Ticker`:

- `str` - строка текста, появляющаяся в виде бегущей строки.

Создавая объект класса `Ticker` с помощью рассмотренного конструктора, вы задаете значение для параметра `str`, и эта строка текста будет циклично прокручиваться в программе.

5.5.7. Методы класса *Ticker*

В составе класса `Ticker` существуют всего два метода для получения и установки необходимой строки текста для приложения:

- `String getString ()` - получает строку текста, заданную для объекта класса `Ticker`;
- `void setString (String str)` - устанавливает строку текста для отображения ее на экране телефона с помощью объекта класса `Ticker`, заменяя ее новой строкой.

Также имеется возможность воспользоваться еще двумя методами абстрактного класса `Displayable`. Оба метода выполняют аналогичные действия методом класса `Ticker`, но при этом позволяют встраивать объект класса `Ticker` непосредственно в форму, то есть экран, представленный классом `Form`. Разберем эти два метода:

- `void setTicker (Ticker ticker)` - устанавливает новую бегущую строку, заменяя предыдущую;
- `Ticker getTicker ()` - получает используемую строку текста.

Оба этих метода дублируют по сути методы класса `Ticker`. В листинге 6.10 приводится образец применения класса `Ticker`, также весь исходный код находится на компакт-диске в папке `\Code\Chapter6\Listing6_10\src`.

```
/**
Листинг 6.10
Класс Ticker
*/

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MainClassTicker extends MIDlet
implements CommandListener
{
    // команда выхода из приложения
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0);
    // объект класса Form
    private Form myform;
    // объект mydisplay представляет экран телефона
    private Display mydisplay;

    public MainClassTicker()
    {
        mydisplay = Display.getDisplay(this);
    }

    public void startApp()
    {
        // создаем форму при помощи объекта Form
        myform = new Form(" Класс Ticker");
        // создаем объект класса Ticker
        Ticker myticker = new Ticker(" Бегущая строка " );
        // добавляем бегущую строку в форму
        myform.setTicker(myticker);
    }
}
```

```
// добавить команду выхода
myform.addCommand(exitMidlet);
myform.setCommandListener(this);
mydisplay.setCurrent(myform);
}

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable d)

    // выход из приложения
    if(c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
}
```

В листинге 6.10 создаются пустая форма с помощью класса `Form` и объект класса `Ticker` с заданным текстом. Методом `setTicker()` объект класса `Ticker` добавляется в форму, организовывая тем самым бегущую строку в верхней части экрана телефона.

6.6. Класс `Image`

При рассмотрении класса `ImageItem` мы уже использовали объекты класса `Image`, но тогда был рассмотрен только один метод и способ работы с классом `Image`. В этом разделе вы более подробно познакомитесь с этим классом. Класс `Image` необходим для загрузки и хранения изображений в формате PNG. Чаще всего загружаемые изображения находятся в рабочем каталоге приложения. Но могут находиться и где угодно, надо только правильно указать путь местонахождения для загрузки. При упаковке приложения в JAR-файл все имеющиеся изображения автоматически копируются в архив, и при работе программы на телефоне загрузка уже осуществляется из JAR-файла. Загружаемые изображения могут использоваться во время работы с классами `Alert`, `Choice`, `ChoiceGroup`, `Form`, `ImageItem` и `Graphics`. Качество воспроизведения изображения на экране всецело зависит от возможностей используемого телефона. Если изображение больше фактического размера дисплея, то сервис телефона организует прокрутку изображения, и если это не входит в ваши планы, то следует придерживаться минимальных размеров ширины и высоты при создании изображений.

6.6.1. Методы класса *Image*

Все методы класса `Image` служат для загрузки изображений из файлов, ресурсов, потоков, а в некоторых методах можно задавать размеры и трансформацию изображений. Проанализируем основные методы класса `Image`:

- `static Image createImage(byte[] imageData, int imageOffset, int imageLength)` - загружает изображение, учитывая смещение и длину в байтах;
- `static Image createImage (Image source)` - загружает изображение из файла;
- `static Image createImage (Image image, int x, int y, int width, int height, int transform)` - загружает изображение в заданное место, определенное координатами, с возможностью трансформации изображения. Параметр `transform` устанавливает необходимую трансформацию с помощью класса `Sprite` и константных значений:
 - `Sprite.TRANS_NONE` - изображение копируется без трансформации;
 - `Sprite.TRANS_ROT90` - трансформирует изображение по часовой стрелке на 90°;
 - `Sprite.TRANS_ROT180` - трансформирует изображение по часовой стрелке на 180°;
 - `Sprite.TRANS_ROT270` - трансформирует изображение по часовой стрелке на 270°;
- `static Image createImage (InputStream stream)` - загружает изображение из потока;
- `static Image createImage(int width, int height)` - загружает изображение в заданные размеры;
- `static Image createImage (String name)` - загружает изображение из ресурса;
- `static Image createRGBImage(int[] rgb, int width, int height, boolean processAlpha)` - загружает изображение, учитывая цветовую компоненту **ARGB**;
- `Graphics getGraphics()` - создает графический объект для представления изображения;
- `int getHeight()` - получает высоту изображения;
- `int getWidth()` - получает ширину изображения.

В листинге 6.11 происходит загрузка изображения в приложение, но без использования объекта класса `ImageItem`, который использовался при рассмотрении примера в листинге 6.6 из раздела 6.8. Применялась ссылка на объект класса `Image`, в этом примере объект класса `Image` используется напрямую. Исходный код примера можно найти на компакт-диске в папке `\Code\Chapter6\Listing6_11\src`.

Класс Image

```
*/

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MainClassImage extends MIDlet implements
CommandListener
{
    // команда выхода из приложения
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 1);
    // объект класса Form
    private Form myform = new Form("Класс Image");
    // объект mydisplay представляет экран телефона
    private Display mydisplay;

    public MainClassImage()
    {
        mydisplay = Display.getDisplay(this);
    }

    public void startApp()
    {
        // перехватываем исключительную ситуацию
        try{
            // загрузка изображения
            Image im = Image.createImage("/gornakov.png " ) ;
            // добавляем загруженный файл в форму
            myform.append(im);
        } catch (java.io.IOException ex) { }
        // Установка обработчика событий для Form
        myform.addCommand(exitMidlet);
        myform.setCommandListener(this);
        // Отобразить текущий дисплей
        mydisplay.setCurrent(myform);
    }

    public void pauseApp() {}
    public void destroyApp(boolean unconditional) {}

    public void commandAction(Command c, Displayable d)
    {
        // Выход из приложения
    }
}
```

```
if (c == exitMidlet)
{
    destroyApp(false) ;
    notifyDestroyed() ;
}
}
```

На рис. 6.13 показан эмулятор телефона, воссоздающий загруженное изображение, посмотрите, насколько больше экран телефона и как разместилось меньшее по размеру изображение на дисплее.



Рис. 6.13. Загрузка изображения классом Image

6.7. Класс Font

При формировании приложения программисту всегда хочется улучшить его внешний вид. Кроме обилия компонентов, создающих списки, таблицы, бегущие строки, существует *класс Font*, с помощью которого можно задавать шрифт для текста. Телефоны имеют ограниченные системные ресурсы, поэтому доступно всего несколько шрифтов, которые отличаются по размеру, начертанию, стилю и задаются при помощи констант.

Размер шрифта устанавливается при помощи трех констант:

- `int SIZE_LARGE` - большой шрифт;
- `static int SIZE_MEDIUM` - средний шрифт;
- `static int SIZE_SMALL` - маленький шрифт.

Стиль можно задавать четырьмя константами:

- `static int STYLE_BOLD` - жирный шрифт;
- `static int STYLE_ITALIC` - курсив;
- `static int STYLE_PLAIN` - обычный шрифт;
- `static int STYLE_UNDERLINED` - подчеркнутый шрифт.

Начертание шрифта определяется тремя константами:

- `static int FACE_MONOSPACE` - шрифте небольшим интервалом;
- `static int FACE_PROPORTIONAL` - пропорциональный шрифт;
- `static int FACE_SYSTEM` - системный шрифт.

В профиле **MIDP 1.0** возможность установки различных шрифтов в приложении имела только при использовании класса `Graphics` и метода `setFont()`. В профиле **MIDP 2.0** уже имеется возможность установки шрифта без использования класса `Graphics`, только при помощи методов из состава классов пользовательского интерфейса. Процесс назначения шрифта текста в программе происходит следующим образом. Вначале создается переменная, которая будет содержать размер, стиль и начертание шрифта, установленные при помощи *метода* `getFont()` класса `Font`, например:

```
Font myFont =
Font.getFont(Font.FACE_SYSTEM,Font.STYLE_BOLD,Font.SIZE_LARGE);
```

Переменная `myFont` теперь содержит шрифт, который можно назначить любому тексту в программе. В профиле **MIDP 2.0** для этого достаточно вызвать *метод* `setFont()` с необходимыми параметрами. В профиле **MIDP 1.0** для назначения шрифта тексту необходимо использовать класс `Graphics`, в *главе 6* рассматривается эта возможность.

В примере к этому разделу будет задействован класс `List`, создающий список элементов. При создании на экране списка из четырех элементов каждому элементу будет назначен свой шрифт. В листинге 6.12 содержится код примера, создающего различные шрифты элементам списка. Весь исходный код также содержится на компакт-диске в папке `\Code\Chapter6\Listing6_12\src`.

```
/**
Листинг 6.12
Класс Font
*/

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



```
public class MainClassFont extends MIDlet implements
CommandListener
{
    // команда выхода из приложения
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0);
    // массив иконок
    Image[] icon = null;
    // объект класса List
    private List mylist;
    // объект mydisplay представляет экран телефона
    private Display mydisplay;

    public MainClassFont()
    {
        mydisplay = Display.getDisplay(this);
    }

    public void startApp()
    {
        // перехватываем исключительную ситуацию
        try{
            // загрузка изображения
            Image image0 = Image.createImage("/icon0.png " );
            Image image1 = Image.createImage("/icon1.png " );
            Image image2 = Image.createImage("/icon2.png " );
            Image image3 = Image.createImage("/icon3.png " );
            // поместить загруженные изображения в массив icon
            icon = new Image[]{ image0, image1, image2, image3};
        } catch(java.io.IOException ex){ }
        // текст для четырех элементов списка
        String[] stroka = {"Синий", "Красный", "Зеленый", "
            Оранжевый"};
        // назначается шрифт нулевому элементу списка
        Font f0 = Font.getFont(Font.FACE_PROPORTIONAL,
            Font.STYLE_PLAIN, Font.SIZE_SMALL);
        // назначается шрифт первому элементу списка
        Font f1 =
        Font.getFont (Font.FACE_SYSTEM, Font.STYLE_BOLD, Font.SIZE_MEDIUM);
        // назначается шрифт второму элементу списка
        Font f2 =
            Font.getFont (Font.FACE_MONOSPACE, Font.STYLE_ITALIC,
```

```
Font.SIZE_LARGE);
// назначается шрифт третьему элементу списка
Font f3 =
    Font.getFont(Font.FACE_SYSTEM,Font.STYLE_UNDERLINED,
        Font.SIZE_LARGE);
// инициализация объекта mylist
mylist = new List(" Класс List ", Choice.EXCLUSIVE,
    stroka, icon);
// устанавливается шрифт нулевому элементу списка
mylist.setFont(0,f0);
// устанавливается шрифт первому элементу списка
mylist.setFont(1,f1);
// устанавливается шрифт второму элементу списка
mylist.setFont(2,f2);
// устанавливается шрифт третьему элементу списка
mylist.setFont(3, f3);
// добавить команду выхода
mylist.addCommand(exitMidlet);
mylist.setCommandListener(this);
// отразить текущий дисплей
mydisplay.setCurrent(mylist);
}

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable d)
{
    // выход из приложения
    if(c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
}
```

Основным классом в программе из листинга 6.12, иллюстрирующей работу со шрифтом, является *класс* *MainClassFont*. Весь код программы построен на использовании класса *List*, создающего список элементов. В строке кода

```
Image[] icon = null;
```

создается переменная для хранения массива изображений. Конкретно в этом примере будут использованы маленькие иконки, загружаемые каждому элементу

списка. Всю группу элементов представляет объект `mylist` класса `List`. В методе `startApp()` происходят основные действия по созданию списка элементов, загрузке изображения и назначению шрифта каждому элементу списка. В четырех строках кода:

```
Image image0 = Image.createImage("/icon0.png" );
Image image1 = Image.createImage("/icon1.png" );
Image image2 = Image.createImage("/icon2.png" );
Image image3 = Image.createImage("/icon3.png" );
```

загружаются четыре различные иконки в виде шариков, окрашенных в синий, красный, зеленый и оранжевый цвета. Все они помещаются в массив:

```
icon = new Image[] {image0, image1, image2, image3};
```

Далее в программном коде создается массив строковых значений:

```
String[] stroka = {" Синий", " Красный", " Зеленый", " Оранжевый" };
```

Теперь пришло время создать шрифты:

```
Font f0 = Font.getFont(Font.FACE_PROPORTIONAL,
Font.STYLE_PLAIN, Font.SIZE_SMALL);
Font f1 =
Font.getFont(Font.FACE_SYSTEM,Font.STYLE_BOLD,Font.SIZE_MEDIUM);
Font f2 =
Font.getFont(Font.FACE_MONOSPACE,Font.STYLE_ITALIC,
Font.SIZE_LARGE);
Font f3 =
Font.getFont(Font.FACE_SYSTEM,Font.STYLE_UNDERLINED,
Font.SIZE_LARGE);
```

В этих строках создаются четыре переменные `f0...f3`, содержащие различные по стилю, размеру и начертанию шрифты. С помощью созданных переменных впоследствии будет производиться установка шрифтов для каждого элемента списка. Список элементов представлен объектом `mylist` и выполнен по типу `Exclusive` (четыре элемента со своими иконками). В классе `List` имеется метод `setFont()`, доступный в профиле **MIDP 2.0**, он и используется в примере.

Создав объект класса `List`, установим шрифт всем элементам списка:

```
mylist.setFont(0,f0);
mylist.setFont(1,f1);
mylist.setFont(2,f2);
mylist.setFont(3,f3);
```

С помощью метода `setFont()` происходит установка заданного шрифта, содержащегося в переменных `f0...f3`. Назначение шрифта происходит по индексам от 0 до 3 в массиве `stroka[]`.

В конце кода происходят добавление команды выхода и отображение текущего экрана на дисплее телефона. На рис. 6.14 представлена работа программы из листинга 6.12.

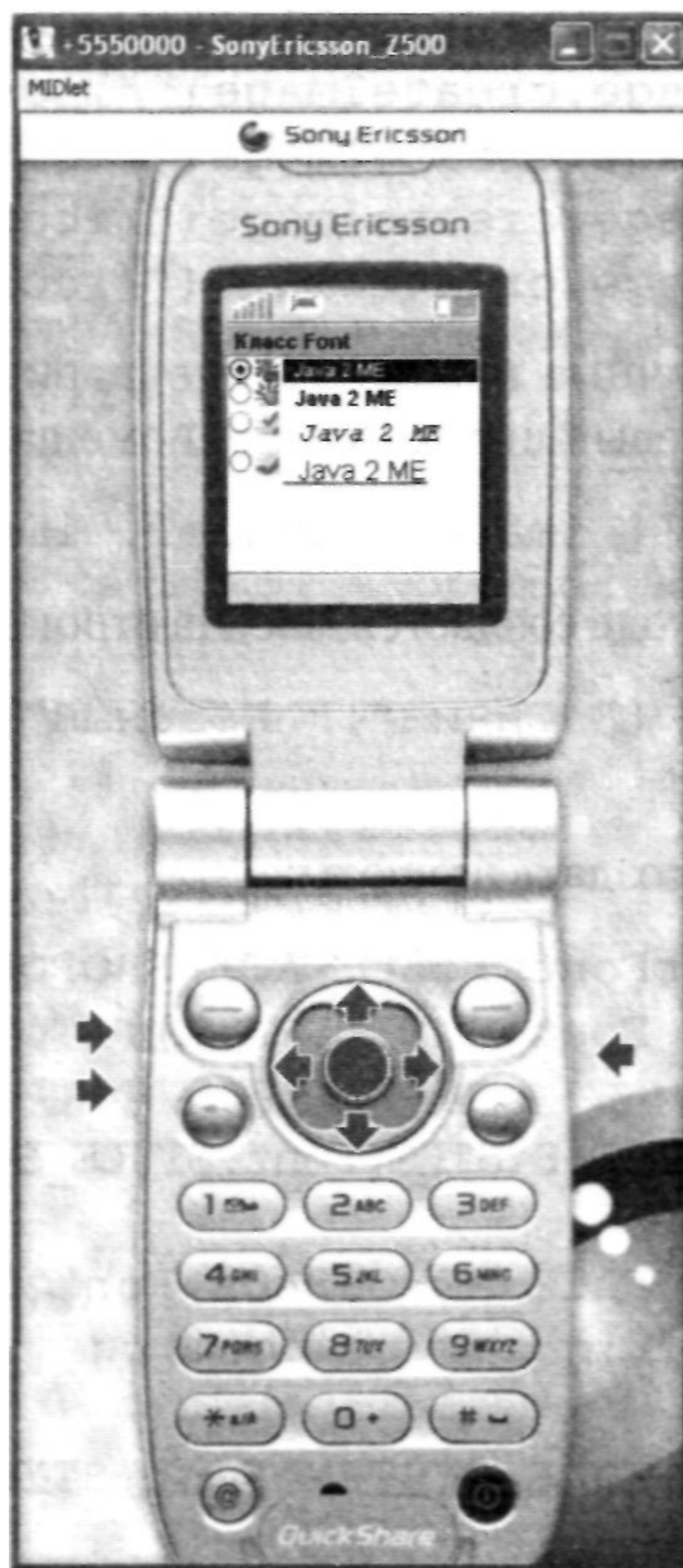


Рис. 6.14. Эмулятор телефона, на экране которого представлены разные шрифты

К сожалению, ресурсы мобильных телефонов не позволяют воспользоваться обилием шрифтов компьютерной платформы, но и имеющейся базы вполне достаточно для создания разнообразных текстовых элементов. Использование шрифтов в программах значительно улучшает пользовательский интерфейс и придает приложению более изящный вид. В следующей главе будет рассматриваться программирование графики в Java 2 ME с помощью классов низкоуровневого интерфейса.

Глава 7. Программирование графики

Высокоуровневые классы, изученные в *главе 6*, дают возможность создавать пользовательский интерфейс приложения. По сути, эти классы выполнены в виде шаблонов, используя которые вы можете создавать списки, формы, шрифт, группы элементов, бегущие строки. Но использование таких классов-шаблонов несколько упрощает интерфейс программы, лишая возможности использовать графику в программах на Java 2 ME. Иногда в приложении необходимо нарисовать таблицу, линию, квадрат, то есть воспользоваться графикой для создания насыщенной и красочной программы. Для этих целей в платформе Java 2 ME существуют так называемые классы низкоуровневого интерфейса - это классы `Canvas` и `GameCanvas` (класс `GameCanvas` будет рассмотрен в следующей главе), а также класс `Graphics`, с помощью которого осуществляется непосредственная прорисовка графики на экране телефона. В самом начале этой главы дается характеристика классам `Canvas` и `Graphics`, попутно рассматриваются имеющиеся в составе обоих классов методы, после чего мы приступим к практической части и создадим ряд примеров, иллюстрирующих работу обоих классов.

7.1. Класс `Canvas`

Класс `Canvas` - это абстрактный класс, поэтому необходимо создавать подклассы для работы с классом `Canvas`. Абстрактный класс `Canvas` представляет некий обобщенный графический контекст, что позволяет программе производить прорисовку графики при помощи класса `Graphics`. Кроме этого, класс `Canvas` предоставляет возможность в обработке событий, полученных с клавиш телефона. Если классы высокоуровневого интерфейса, рассмотренные в *главах 5 и 6*, обрабатывают команды перехода, то с помощью класса `Canvas` можно получать события с любой нажатой клавиши телефона.

Существует ряд так называемых «*ключевых кодов*» в виде заданных констант, с помощью которых можно назначать игровые действия для клавиш телефона. Все ключевые коды соответствуют стандарту ITU-T и заданы в виде следующих констант:

- `static int DOWN` - движение вниз;
- `static int FIRE` - обычно используется в играх и реализует стрельбу из оружия;
- `static int GAME_A` - игровая клавиша A;
- `static int GAME_B` - игровая клавиша B;
- `static int GAME_C` - игровая клавиша C;
- `static int GAME_D` - игровая клавиша D;

- `static int KEY_NUM0` - клавиша 0;
- `static int KEY_NUM1` - клавиша 1;
- `static int KEY_NUM2` - клавиша 2;
- `static int KEY_NUM3` - клавиша 3;
- `static int KEY_NUM4` - клавиша 4;
- `static int KEY_NUM5` - клавиша 5;
- `static int KEY_NUM6` - клавиша 6;
- `static int KEY_NUM7` - клавиша 7;
- `static int KEY_NUM8` - клавиша 8;
- `static int KEY_NUM9` - клавиша 9;
- `static int KEY_POUND` - клавиша #;
- `static int KEY_STAR` - клавиша *;
- `static int LEFT` - движение влево;
- `static int RIGHT` - движение вправо;
- `static int UP` - движение вверх.

Ключевые коды `GAME_A`, `GAME_B`, `GAME_C`, `GAME_D` и `FIRE` предназначены специально для игровых действий и обычно задаются клавишам с цифрами соответственно 2, 4, 8, 6 и 5, но зависят от реализации конкретных моделей телефонов.

7.1.1. Методы класса *Canvas*

Большинство методов класса `Canvas` обеспечивают обработку низкоуровневых событий. Абстрактный метод `void paint(Graphics g)` является основным методом, с помощью которого происходит прорисовка графики на экране телефона. Класс `Graphics` определяет, что именно необходимо рисовать на экране телефона. Разберем основную часть методов класса `Canvas`:

- `int getGameAction(int keyCode)` - связывает игровые действия с заданным ключевым кодом;
- `int getKeyCode(int gameAction)` - получает ключевой код игровых действий;
- `String getKeyName(int keyCode)` - получает ключевой код для клавиши;
- `boolean hasPointerMotionEvents()` - проверяет поддержку устройством перемещение указателя;
- `protected void keyPressed(int keyCode)` - вызывается при нажатии клавиши;
- `protected void keyReleased(int keyCode)` - вызывается при отпускании нажатой клавиши;
- `protected void keyRepeated(int keyCode)` - повторное нажатие клавиши;
- `protected abstract void paint(Graphics g)` - прорисовка графики на экране телефона;
- `protected void pointerDragged(int x, int y)` - определяет перемещение курсора;

- `protected void pointerPressed(int x,int y)` - определяет позицию курсора, при которой должно производиться нажатие определенной клавиши;
- `protected void pointerReleased(int x,int y)` - определяет позицию курсора в момент отпускания определенной клавиши;
- `void repaint()` - повторяет прорисовку;
- `void repaint(int x, int y, int width, int height)` - повторяет прорисовку заданной области.

7.2. Класс Graphics

При помощи *класса Graphics* осуществляется двухмерное представление графики на экране телефона. Класс Graphics существует также в составе Java 2 SE, но в платформе Java 2 ME он сильно урезан в связи с ограниченными системными ресурсами телефонов. Поэтому имеется возможность рисовать только линии, прямоугольники, дуги и текст. Для окрашивания этих примитивов предусмотрена работа с цветовой компонентой.

Система координат, используемая для представления графики в Java 2 ME, перевернута по отношению к обычной декартовой системе координат. Начало системы координат находится в левой верхней точке экрана телефона. Положительная ось X проходит по верхней кромке экрана слева направо, а положительная ось Y - сверху вниз по левой стороне экрана, как изображено на рис. 7.1.

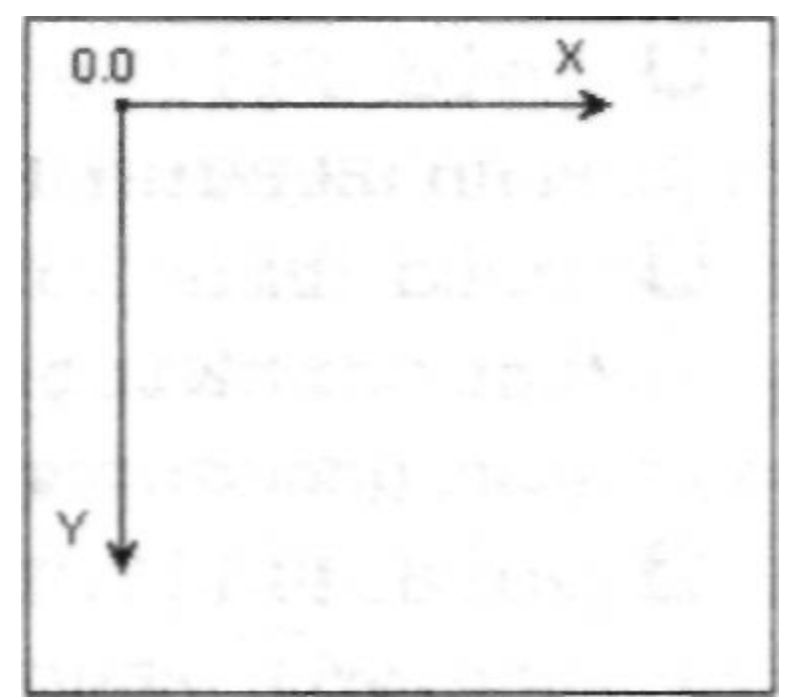


РИС. 7.1. Система координат в Java 2 ME

Такая система координат - отнюдь не новшество в программировании двухмерной графики. Идентичная модель координат применяется также в DirectX и OpenGL для представления двухмерных сцен, но уже в компьютерной графике.

7.2.1. Методы класса Graphics

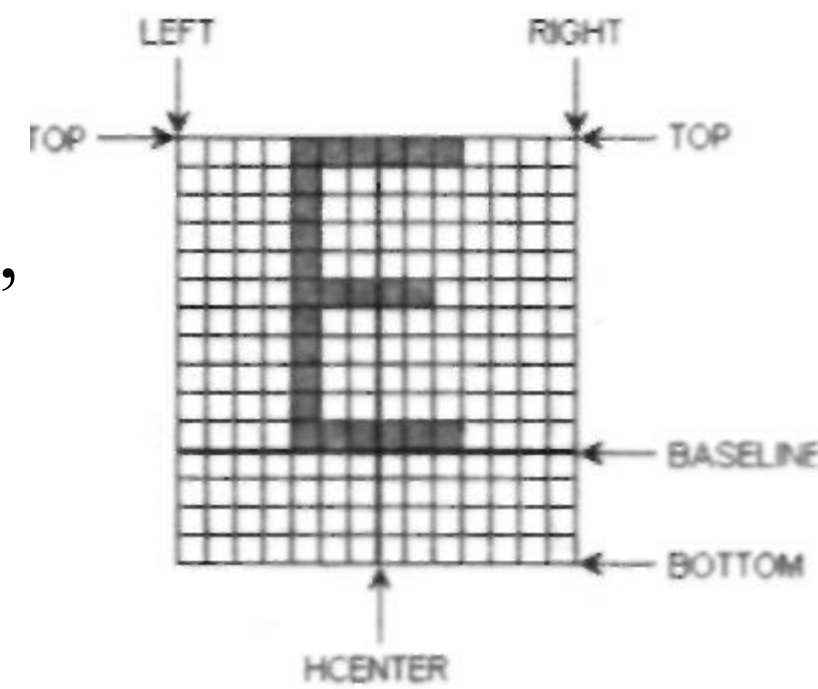
Основные методы класса Graphics обеспечивают прорисовку двухмерной графики. Есть еще несколько методов, с помощью которых можно произвести перемещение системы координат и произвести *отсечение* (clipping). Основные методы класса Graphics:

- `void copyArea(int x_src,int y_src,int width,int height, int x_dest,int y_dest,int anchor)` - копирует прямоугольную область из установленных значений в параметрах (`x_src`, `y_src`, `width`, `height`) в новую область (`x_dest`, `y_dest`);
- `void drawArc(int x,int y,int width,int height,int startAngle, int arcAngle)` - рисует контур дуги в виде эллипса;
- `void drawChar(char character,int x,int y,int anchor)` - рисует символ;

- `void drawChars (char[] data, int offset, int length, int x, int y, int anchor)` - **рисует массив символов**;
- `void drawImage (Image img, int x, int y, int anchor)` - **рисует изображение**;
- `void drawLine (int x1, int y1, int x2, int y2)` - **рисует линию из точки x1 и y1 до точки x2 и y2**;
- `void drawRegion (Image src, int x_src, int y_src, int width, int height, int transform, int x_dest, int y_dest, int anchor)` - **копирует изображения в заданную область на экран телефона**;
- `void drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)` - **рисует контур прямоугольника, используя закругленные углы**;
- `void fillArc (int x, int y, int width, int height, int startAngle, int arcAngle)` - **рисует заполненную цветом дугу**;
- `void fillRect (int x, int y, int width, int height)` - **рисует заполненный цветом прямоугольник**;
- `void fillRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)` - **рисует заполненный прямоугольник, используя закругленные углы**;
- `void fillTriangle (int x1, int y1, int x2, int y2, int x3, int y3)` - **рисует заполненный цветом треугольник**;
- `int getBlueComponent ()` - **получает синий компонент цвета**;
- `int getClipHeight ()` - **получает высоту для текущей области отсечения**;
- `int getClipWidth ()` - **получает ширину для текущей области отсечения**;
- `int getColor ()` - **получает текущий цвет**;
- `Font getFont ()` - **получает текущий шрифт**;
- `int getGreenComponent ()` - **получает зеленый компонент цвета**;
- `int getRedComponent ()` - **получает красный компонент цвета**;
- `void setClip (int x, int y, int width, int height)` - **устанавливает отсечение заданной области экрана**;
- `void setColor (int RGB)` - **устанавливает цвет при помощи значения RGB**;
- `void setColor (int red, int green, int blue)` - **назначает цвет при помощи трех цветовых компонентов: red, green и blue**;
- `void setFont (Font font)` - **устанавливает заданный шрифт**;
- `void setStrokeStyle (int style)` - **задает штриховой стиль рисуемому контексту, используя константы SOLID и DOTTED**;
- `void translate (int x, int y)` - **перемещает систему координат в точку x и y**.

При использовании некоторых методов очень часто используется параметр `int anchor`. С помощью этого параметра задаются различные значения для выбора позиции. Посмотрите на рис. 7.2, где изображен механизм прорисовки текста с выбором определенной позиции.

РИС. 7.2. Техника прорисовки текста



Для этих целей в классе `Graphics` имеются константы, с помощью которых происходит выбор позиции:

- `static int BASELINE` - задает базовую линию;
- `static int BOTTOM` - сдвигает вниз;
- `static int HCENTER` - центрирует;
- `static int LEFT` - сдвигает влево;
- `static int RIGHT` - сдвигает вправо;
- `static int TOP` - сдвигает вверх;
- `static int VCENTER` - используется только при прорисовке изображений, производит вертикальную центровку всего изображения.

Можно использовать две константы для выбора позиции. Например, для того чтобы сдвинуть текст влево и вверх, используется комбинация `Graphics.LEFT | Graphics.TOP`.

Далее мы перейдем к практике и изучим модель программирования графики в приложении на Java 2 ME, рассмотрим создание и отрисовку линий, прямоугольников, дуг и текста. Главное, о чем надо помнить при использовании графических элементов, - это о размере экрана телефона. Разные модели телефонов имеют свои размеры дисплея, и если вы будете использовать большой по площади экран, например 240 x 320 пикселей, то на экране с разрешением 176 x 208 некоторые части графических элементов будут срезаны. Чтобы этого избежать, надо использовать методы класса `Canvas`, `getWidth()` и `getHeight()`, которые возвращают размеры ширины и высоты экрана, и уже на основании этих данных производить построение графических элементов, реализуя тем самым адаптацию графического контекста к конкретной модели телефона. Например, чтобы нарисовать горизонтальную линию, не выходящую из зоны видимости, можно воспользоваться следующим кодом:

```
int w = getWidth();
drawLine(20, 20, w-20, w-20);
```

В своих примерах к этой главе я специально не использую оптимизации графики, для того чтобы исходный код был более понятен. Поэтому после компиляции всех примеров обязательно запустите получившиеся программы на максимальном количестве имеющихся эмуляторов. Особенно попробуйте работу этих программ на эмуляторе `DefaultColorPhone` из состава среды программирования J2ME Wireless Toolkit, меня лично очень сильно позабавил результат работы этого эмулятора, а вам, я думаю, он предоставит некоторую пищу для размышлений.

7.3. Рисование линий

Для того чтобы *нарисовать линию*, нужно воспользоваться *методом* `drawLine()` класса `Graphics`. Рассмотрим прототип этого метода:

```
public void drawLine(int x1,
                    int y1,
```



```
int x2,
int y2)
```

Параметры метода `drawLine()` :

- `x1` - координата начальной точки отрезка по оси X;
- `y1` - координата начальной точки отрезка по оси Y;
- `x2` - координата конечной точки отрезка по оси X;
- `y2` - координата конечной точки отрезка по оси Y.

Задавая целочисленные координаты точек в методе `drawLine()`, можно нарисовать любую линию заданного размера. Также можно воспользоваться *методом* `setColor()` для закрашивания линий цветом и *методом* `setStrokeStyle()` - для рисования линии в виде пунктира.

Рассмотрим пример кода, находящийся в листинге 7.1, где с помощью линий рисуется система координат, используемая в Java 2 ME. Исходный код листинга 7.1 находится на компакт-диске в папке `\Code\Chapter7\Listing7_1\src`.

```
/**
```

ЛИСТИНГ 7.1

Класс Main и класс Line*/

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

public class Main extends MIDlet implements
CommandListener
{
    // команда выхода из программы
    private Command exitMidlet = new Command("Выход",
        Command.EXIT, 0);

    public void startApp()
    {
        // создаем объект класса Line
        Line myline = new Line();
        // добавляем команду выхода
        myline.addCommand(exitMidlet);
        myline.setCommandListener(this);
        Display.getDisplay(this).setCurrent(myline);

        public void pauseApp() {}
        public void destroyApp(boolean unconditional){}

        public void commandAction(Command c, Displayable d)
```

```
{
    if (c == exitMidlet)
    {
        destroyApp(false) ;
        notifyDestroyed() ;
    }
}
}

/**
класс Line определен в файле Line.Java
*/

import javax.microedition.lcdui.*;

public class Line extends Canvas
{
    // конструктор
    public Line(){ super (); }

    public void paint(Graphics g)
    {
        // устанавливается синий цвет для линий
        g.setColor(0x0000ff);
        // рисуется система координат
        g.drawLine(20, 20, 90, 20);
        g.drawLine(20, 20, 20, 90);
        g.drawLine(90, 20, 80, 10);
        g.drawLine(90, 20, 80, 30);
        g.drawLine(20, 90, 10, 80);
        g.drawLine(20, 90, 30, 80);
        // устанавливается красный цвет для трех линий
        g.setColor(0xffff0000);
        // рисуем толстую линию в три пикселя
        g.drawLine(30, 30, 70, 30);
        g.drawLine(30, 31, 70, 31);
        g.drawLine(30, 32, 70, 32);
        // устанавливаем пунктир
        g.setStrokeStyle(Graphics.DOTTED) ;
        // рисуем линию в два пикселя пунктиром
        g.drawLine(30, 50, 70, 50);
        g.drawLine(30, 51, 70, 51);
    }
}
```

В листинге 7.1 используются два класса: `Main`, являющийся основным классом мидлета приложения, и класс `Line`, в котором происходит работа с графикой. Подразумевается, что оба класса находятся в файлах `Main.java` и `Line.java`, это характерно для объектно-ориентированного программирования, и в дальнейшем мы будем придерживаться именно такой модели построения изучаемых примеров. Основным классом мидлета `Main` очень прост. В этом классе создается объект класса `Line`, добавляется команда выхода из приложения и отражается текущий экран. Класс `Line`, находящийся в файле `Line.java` листинга 7.1, рисует набор различных линий.

Сам класс `Line` наследуется от абстрактного класса `Canvas`. В более сложных программах могут использоваться интерфейс `Runnable` и метод `run()`. Такая техника программирования обычно используется при создании игр и будет обсуждаться в конце этой главы.

Конструктор класса `Line` использует метод `super()`, позволяющий обратиться к конструктору своего суперкласса `Canvas`. Основные же события происходят в методе `paint()` класса `Canvas`.

В строке кода

```
g.setColor(0x0000ff);
```

происходит назначение цвета для любого последующего отрисованного примитива. То есть вызов метода `setColor()` с заданным цветом действителен до момента последующего вызова метода `setColor()`, устанавливающего другой цвет. В этом примере используется метод `setColor(RGB)`, поэтому значение цвета задается восьмеричным значением с помощью нулей и букв. В следующем примере из *раздела 7.4* при рисовании прямоугольников будет показана работа метода `setColor` с тремя целочисленными параметрами, задающими значение цвета.

В строках кода

```
g.drawLine(20, 20, 90, 20);
g.drawLine(20, 20, 20, 90);
g.drawLine(90, 20, 80, 10);
g.drawLine(90, 20, 80, 30);
g.drawLine(20, 90, 10, 80);
g.drawLine(20, 90, 30, 80);
```

рисуются шесть синих линий, образующих систему координат. Толщина всех линий равна одному пикселю - это значение по умолчанию, и изменить его нельзя. Для того чтобы нарисовать широкую линию, придется рисовать несколько соприкасающихся одинаковых по размеру линий.

```
g.drawLine(30, 30, 70, 30);
g.drawLine(30, 31, 70, 31);
g.drawLine(30, 32, 70, 32);
```

Этими строками кода рисуется одна толстая линия шириной в три пикселя.

В конце в методе `paint()` рисуется линия толщиной в два пикселя в виде пунктирной линии. Для этого используются метод `setStrokeStyle()` и константа `DOTTED`. На рис. 7.3 изображен эмулятор телефона с результатом работы программы из листинга 7.1.

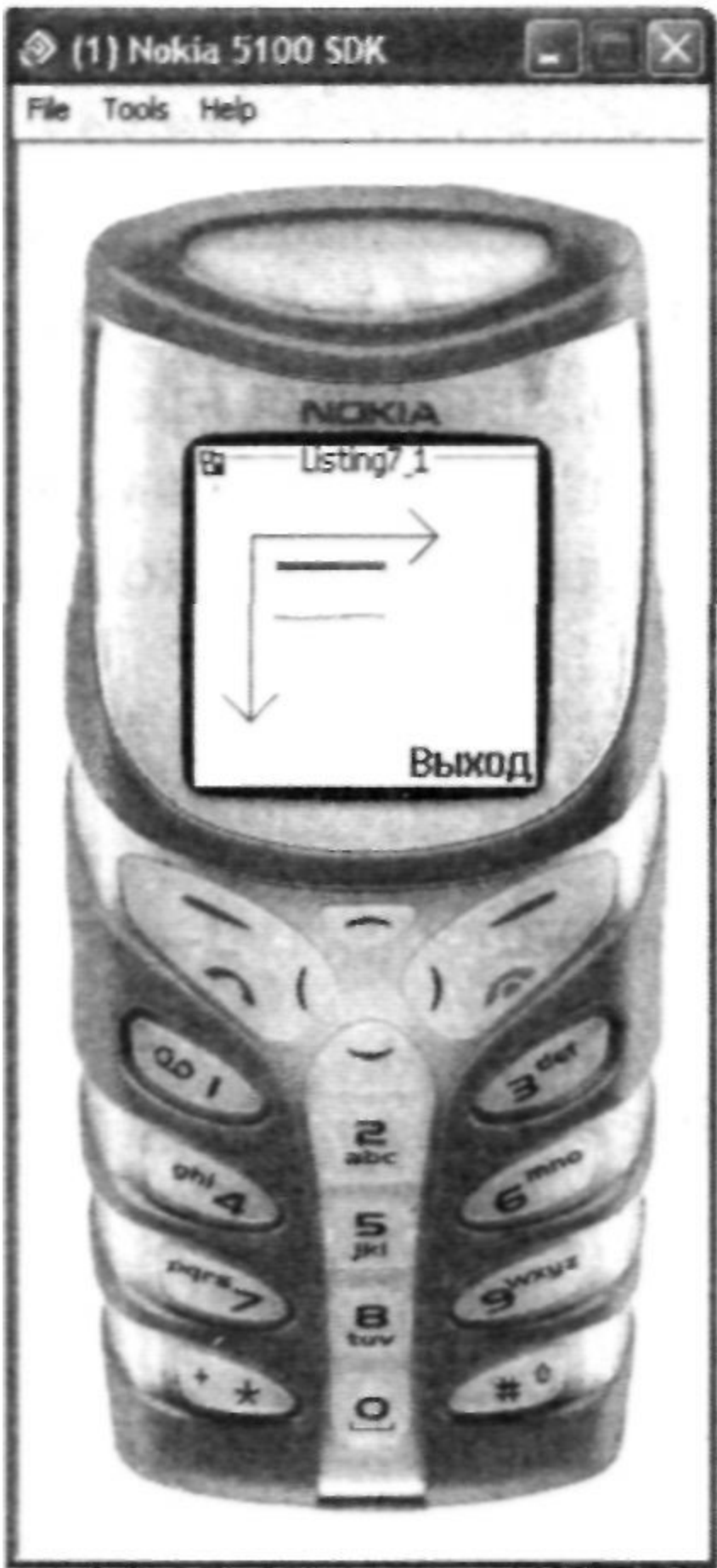


Рис. 7.3. Рисование разноцветных линий

7.4. Рисование прямоугольников

При создании *прямоугольников* можно использовать два метода класса Graphics - это `drawRect()` и `fillRect()`. При помощи метода `drawRect()` рисуется только контур прямоугольника, а метод `fillRect()` позволяет нарисовать прямоугольник уже закрашенным каким-либо цветом (изначально по умолчанию цвет черный). Оба метода абсолютно идентичны по количеству и назначению параметров, поэтому рассмотрим прототип одного из них, а именно *метода* `drawRect()`.

```
public void drawRect(int x,
                    int y,
                    int width,
                    int height)
```

Параметры метода `drawRect()`:

- `x` - координата точки по оси X для левого верхнего угла прямоугольника;
- `y` - координата точки по оси Y для левого верхнего угла прямоугольника;
- `width` - ширина рисуемого прямоугольника;
- `height` - высота рисуемого прямоугольника.

В составе класса Graphics имеется еще один метод, рисующий прямоугольник, но с закругленными углами, - `drawRoundRect()`. Этот метод имеет уже шесть параметров, где первые четыре параметра работают в том же ключе, что и методы `drawRect()` и `fillRect()`, а два последних параметра определяют степень закругленности углов. В листинге 7.2 рисуется три разных по размеру

прямоугольника, и закрашиваются они тремя цветами: красным, зеленым и синим. Для закрашивания прямоугольников применяется метод `setColor (int red, int green, int blue)` с тремя параметрами. Выставляя любое целочисленное значение от 0 до 255 для каждого параметра, можно создавать разнообразную цветовую гамму, естественно, учитывая при этом цветовые возможности телефона, на котором будет работать эта программа. Для того чтобы определить доступную цветность дисплея телефона, необходимо воспользоваться методами класса *Display*.

- `isColor()` - если телефон поддерживает цветовую гамму, то возвращает значение `true`;
- `numColor()` - определяет количество доступных цветов.

На компакт-диске листинг 7.2 находится в папке **\Code\Chapter7\Listing7Listing7_2\src**.

```
/**
Листинг 7.2
Класс Main и класс Rectangles
*/
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class Main extends MIDlet implements
CommandListener
{
    // команда выхода из программы
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0);

    public void startApp()
    {
        // создаем объект класса Rectangles
        Rectangles myrec = new Rectangles();
        // добавляем команду выхода
        myrec.addCommand(exitMidlet);
        myrec.setCommandListener(this);
        Display.getDisplay(this).setCurrent(myrec);
    }

    public void pauseApp() {}
    public void destroyApp(boolean unconditional){}

    public void commandAction(Command c, Displayable d)
    {
        if (c == exitMidlet)
```

```
{
    destroyApp(false);
    notifyDestroyed();
}
}

/**
класс Rectangles определен в файле Rectangles.Java
*/

import javax.microedition.lcdui.*;

public class Rectangles extends Canvas
{
    // конструктор
    public Rectangles() { super(); }

    public void paint(Graphics g)
    {
        // устанавливается красный цвет
        g.setColor(255, 0, 0);
        // рисуем первый прямоугольник
        g.fillRect(/*x*/ 15, /*y*/ 30, /*ширина*/ 15, /*высота*/ 20);
        // устанавливается зеленый цвет
        g.setColor(0, 255, 0);
        // рисуем второй прямоугольник
        g.fillRect(30, 30, 15, 45);
        // устанавливается синий цвет
        g.setColor(0, 0, 255);
        // рисуем третий прямоугольник
        g.fillRect(45, 30, 15, 60);
        // устанавливается синий цвет
        g.setColor(255, 0, 0);
        // рисуем прямоугольник с закругленными углами
        g.drawRoundRect(70, 30, 40, 40, 10, 10);
    }
}
```

В этом примере также используются два класса - класс `Main`, играющий роль основного класса мидлета, и класс `Rectangles`, где происходит отрисовка графики. Оба класса разделены на два файла `Main.java` и `Rectangles.java`. В классе `Main` создается объект класса `Rectangles`, добавляется команда выхода и показывается текущий экран. Класс `Rectangles` является подклассом класса `Canvas`. Прорисовка прямоугольников происходит в методе `paint()` класса `Graphics`.


```
g.setColor(255, 0, 0);
```

В этой строке кода устанавливается красный цвет для прямоугольника размером 15 x 20 пикселей, который рисуется с помощью метода `fillRect()`.

```
g.fillRect(/*x*/ 15, /*y*/ 30, /*ширина*/ 15, /*высота*/ 20);
```

Прямоугольник рисуется закрашенным в красный цвет. Далее происходит прорисовка еще двух закрашенных в зеленый и синий цвета прямоугольников с размерами соответственно 15x45 и 15x60 пикселей.

В конце всего кода в классе `Rectangles` рисуется контур прямоугольника с закругленными углами. На рис. 7.4 изображен эмулятор телефона с четырьмя нарисованными прямоугольниками.



Рис. 7.4. Рисование четырех прямоугольников

7.5. Рисование дуг

В английском языке слово *arc* означает дугу, и именно это слово применяется в документации по Java 2 ME. Используя *методы* `drawArc()` и `fillArc()`, можно нарисовать как *дугу*, так и полноценную *окружность*. Используя оба метода, как вы уже, наверное, заметили, можно нарисовать контур дуги и закрашенную цветом дугу. Методы `drawArc()` и `fillArc()` имеют одинаковое количество параметров со сходными действиями. Рассмотрим один из *методов* - `fillArc()`.

```
public void fillArc(int x,
                    int y,
                    int width,
                    int height,
                    int startAngle,
                    int arcAngle)
```

Параметры метода `fillArc()`:

- `x` - расстояние, откладываемое от оси `X` до мнимой вертикальной касательной к окружности;
- `y` - расстояние, откладываемое от оси `Y` до мнимой горизонтальной касательной к окружности;
- `width` - ширина рисуемой дуги (горизонтальный радиус);
- `height` - высота рисуемой дуги (вертикальный диаметр);
- `startAngle` - стартовый угол для начала рисования дуги;
- `arcAngle` - протяженность дуги (значение 360 замкнет дугу в окружность).

На рис. 7.5 схематично изображена техника создания дуги.

В листинге 7.3 приведен пример кода, создающего три разноцветных сегмента круга, наложенных друг на друга, и дугу в виде контура.

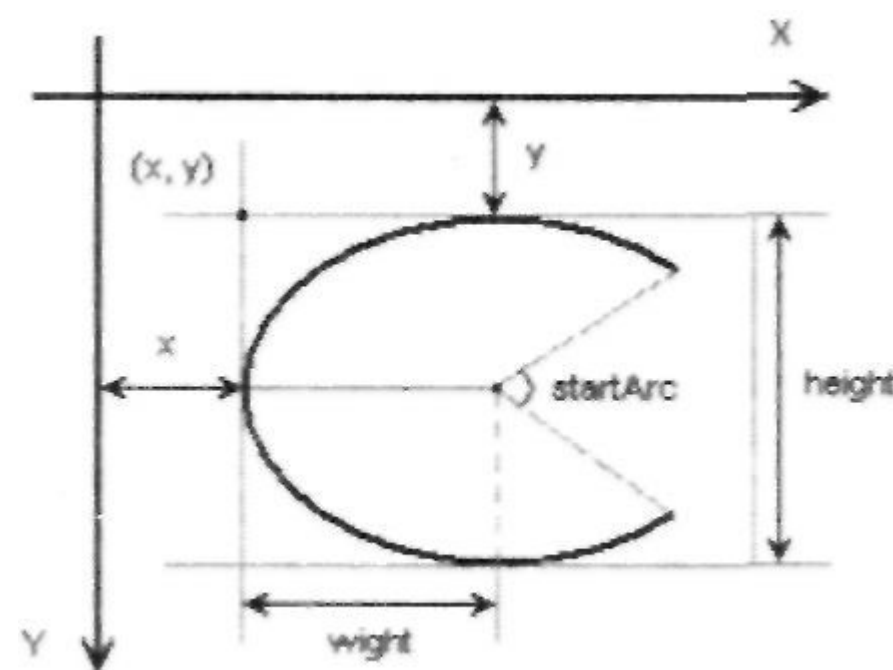


Рис. 7.5. Техника создания дуги

```
/**
Листинг 7.3
Класс Main и класс Arc
*/

import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class Main extends MIDlet implements
CommandListener
{
    // команда выхода из программы
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0);

    public void startApp()
    {
        // создаем объект класса Arc
        Arc myarc = new Arc();
        // добавляем команду выхода
        myarc.addCommand(exitMidlet);
        myarc.setCommandListener(this);
        Display.getDisplay(this).setCurrent(myarc);
    }

    public void pauseApp() {}
    public void destroyApp(boolean unconditional){}
```

```
public void commandAction(Command c, Displayable d)
{
    if (c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}

/
**класс Arc определен в файле Arc.Java
рисует дуги
*/
import javax.microedition.lcdui.*;

public class Arc extends Canvas
{
    // конструктор
    public Arc(){ super(); }

    public void paint(Graphics g)
    {
        // устанавливается красный цвет
        g.setColor(255, 0, 0);
        // рисуем первую заполненную цветом дугу
        g.fillArc(15, 15, 60, 60, 45, 360);
        // устанавливается зеленый цвет
        g.setColor(0, 255, 0);
        // накладываем вторую дугу поверх первой
        g.fillArc(15, 15, 60, 60, 45, 180);
        // устанавливается синий цвет
        g.setColor(0, 0, 255);
        // накладываем третью дугу поверх первых двух
        g.fillArc(15, 15, 60, 60, 45, 90);
        // устанавливается синий цвет для дуги в виде контура
        g.setColor(0, 0, 255);
        // рисуем контур дуги
        g.drawArc(5, 5, 80, 80, 30, 180);
    }
}
```

В листинге 7.3 используется тот же самый механизм, что и в примерах из листингов 7.2 и 7.1, - создаются два класса Main и Arc, находящиеся в файлах

Main.java и Arc.java. Код этого примера вы найдете на компакт-диске в папке \Code\Listing7_3\src. Все действия по прорисовке дуг осуществляются в методе `paint()`.

```
g.setColor(255, 0, 0) ;
g.fillArc(15, 15, 60, 60, 45, 360) ;
```

В этих строках кода происходят установка цвета для рисуемой дуги и прорисовка самой дуги. Первые два значения в методе `fillArc()` - 15 и 15 пикселей задают координаты точки в пространстве, относительно которой будет происходить прорисовка дуги. Значения 60 и 60 пикселей задают ширину и высоту дуги. Значением 45 устанавливается угол для начала рисования дуги (со значением 360 будет нарисована замкнутая окружность). Затем в примере рисуются еще два сегмента зеленого и синего цвета, наложенные поверх первой нарисованной дуги.

7.6. Вывод текста

Для *вывода текста* на экран телефона можно воспользоваться методами `drawString()` и `drawChar()`, рисующими соответственно строку текста и любой назначенный символ. Текст можно выводить с любым цветом, а также использовать стили начертания, изученные в *главе 6*. Прототип *метода* `drawString()` выглядит следующим образом:

```
public void drawstring (String str,
                        int x,
                        int y,
                        int anchor)
```

Параметры метода `drawString()`:

- `str` - строка текста;
- `x` и `y` - задают размер невидимого прямоугольника, в котором происходит расположение текста;
- `anchor` - в этом параметре задается выбор позиции текста внутри невидимого прямоугольника. Здесь используются константы класса `Graphics`, рассмотренные в разделе 7.2.

В листинге 7.4 показан пример вывода текста на экран телефона. Код довольно прост, и я думаю, вам не составит труда разобраться в нем самостоятельно. На компакт-диске код примера находится в папке \Code\Chapter7\Listing7Listing7_4\src.

```
/**
```

Листинг 7.4

Класс Main и класс Text

```
*/
```

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
public class Main extends MIDlet implements
CommandListener
```

```
{
// команда выхода из программы
private Command exitMidlet = new Command("Выход",
Command.EXIT, 0);

public void startApp()
{
    // создаем объект класса Text
    Text mytext = new Text();
    // добавляем команду выхода
    mytext.addCommand(exitMidlet);
    mytext.setCommandListener(this);
    Display.getDisplay(this).setCurrent(mytext);
}

public void pauseApp () {}

public void destroyApp(boolean unconditional){}

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

/**
класс Text определен в файле Text.Java
рисует текст
*/
import javax.microedition.lcdui.*;
public class Text extends Canvas
{
    // конструктор
    public Text(){super();}

    public void paint(Graphics g)
    {
        // устанавливается цвет
        g.setColor(10, 80, 200);
    }
}
```

```
// рисуем строку текста
g.drawString("Java 2 Micro Edition",
            80, 40, Graphics.TOP|Graphics.HCENTER);
}
}
```

7.7. Механизм создания игрового цикла

Для *создания цикла* классом Canvas используется *интерфейс Runnable* и его единственный *метод run()*, в котором реализуется цикл прорисовки графики. Рассмотрим в качестве примера *класс DemoGraphics* и проанализируем его.

```
public class DemoGraphics extends Canvas implements
Runnable
{
public void run()
{
    while(true)
    {
        // обновление графических элементов
        repaint();
        // задержка цикла на 20 миллисекунд для обновления
        // состояния дисплея
        Thread.sleep(20);
    }
}

public void paint( Graphics g )

{

    // код, создающий графические элементы

}
```

```
public void keyPressed( int keyCode )
```

В классе DemoGraphics для упрощения опущен основной код, связанный с прорисовкой графики и обработкой событий, поступающих с клавиш телефона. // обработка событий с клавиш телефона при помощи *Весь цикл прорисовки графики состоит из трех методов: run(), paint() и keyPressed().* ключевых кодов

В методе paint() происходят создание и отрисовка графических элементов программы на экране телефона, а метод run() создает цикл, в котором происходит постоянное обновление экрана телефона, связанное, например, с движением графического элемента. События, полученные в результате нажатия клавиш,

поступают в метод `keyPressed()`, где обрабатываются заданным вами способом. На основании этих событий также может происходить обновление цикла прорисовки графики.

В методе `run()` создается бесконечный цикл `while (true)`. В реальной программе необходимо, конечно, предусмотреть выход из бесконечного цикла. Метод `repaint()` постоянно обновляет графические элементы. Метод `sleep()` класса `Thread` останавливает системный поток на двадцать миллисекунд, для того чтобы отреагировать на все произошедшие изменения, а именно произвести обработку событий с клавиш телефона и перерисовать графический элемент на экране телефона. Механизм прорисовки графики в профиле MIDP 1.0 строится именно по такому принципу, связанному с изменением состояния графических элементов и называемому игровым циклом.

Теперь давайте рассмотрим как можно больше примеров, реализующих вывод графики на экран, перемещение графических элементов, обработку событий с клавиш телефона, столкновение графического элемента с препятствием и других хитростей, из которых состоят мобильные игры.

7.8. Перемещение квадрата

Начнем с самого простого - выведем на экран синий квадрат, прорисованный с помощью метода `fillRect()`, и заставим переместиться его через весь экран по горизонтали слева направо. Посмотрите на код из листинга 7.5, производящий перемещение квадрата на экране, также пример можно найти на компакт-диске в папке `\Code\Chapter7\Listing7Listing7_5\src`.

```
/**
Листинг 7.5
Класс Main и класс Draw
*/
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class Main extends MIDlet implements
CommandListener
{
    // команда выхода из программы
    private Command exitMidlet = new Command("Выход",
        Command.EXIT, 0);

    public void startApp()
    {
        // создаем объект класса Draw
        Draw dr = new Draw();
        // запускаем поток
        dr.start();
    }
}
```

```
// добавляем команду выхода
dr.addCommand(exitMidlet);
dr.setCommandListener(this);
Display.getDisplay(this).setCurrent(dr);
}

public void pauseApp() {}
public void destroyApp(boolean unconditional){}

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

/**
класс Draw определен в файле Draw.Java
перемещает квадрат по экрану
*/
import javax.microedition.lcdui.*;
public class Draw extends Canvas implements Runnable
{
    // позиция для перемещения квадрата
    int position = 10;
    // конструктор
    public Draw()
    { super(); }

    public void start()
    {
        // создаем и запускаем поток
        Thread t = new Thread(this);
        t.start();
    }
    // метод run() интерфейса Runnable
    public void run()
    {
        // бесконечный цикл
```

```
while (true)
{
    // увеличиваем позицию на 1
    position ++;
    // обновляем экран
    repaint();
    // останавливаем цикл на 20 миллисекунд
    try { Thread.sleep(20); }
    catch (Java.lang.InterruptedException zxz) {}
}

public void paint(Graphics g)
{
    // вычисляем область для перерисовки экрана
    int x = g.getClipWidth();
    int y = g.getClipHeight();
    // устанавливаем белый цвет фона
    g.setColor(0xffffffff);
    // назначаем перерисовку всему экрану
    . g.fillRect(0,0,x,y);
    // устанавливается синий цвет квадрата
    g.setColor(0,0,200);
    // рисуем квадрат
    g.fillRect(position,40,20,20);
}
}
```

Листинг 7.5 содержит два класса `Main` и `Draw`, находящихся в файлах `Main.java` и `Draw.java`. Основной класс мидлета `Main` содержит код, создающий объект `dr` класса `Draw`. Затем он запускает системный поток с помощью метода `start()`, добавляет команду выхода из программы и отображает текущий дисплей с объектом `dr`. Класс `Draw` содержит код, создающий синий квадрат и перемещающий его по экрану. Прежде чем мы начнем рассмотрение кода класса `Draw`, давайте подумаем, как можно произвести перемещение объекта на экране. Самый простейший способ перемещения объекта по экрану вдоль оси X заключается в постоянном увеличении, допустим на единицу, позиции этого объекта по оси X . При создании квадрата методом `fillRect()` задаются две координаты по оси X и по оси Y для первоначального вывода квадрата на экран. Поэтому достаточно создать переменную для координаты по оси X и затем в цикле прорисовки увеличивать ее на единицу, перемещая тем самым объект по экрану.

В листинге 7.5 класс `Draw` наследуется от класса `Canvas`, что дает возможность воспользоваться методом `paint()` для рисования графических элементов на экране и реализации метода `run()` интерфейса `Runnable`. В методе `run()` создается цикл, постоянно обновляющий состояние графических элементов.

В начале кода класса `Draw` создается переменная `position`, отвечающая за координату по оси `X` для точки вывода квадрата на экран. Конструктор класса `Draw` вызывает метод `super()` для использования конструктора своего супер-класса `Canvas`.

В методе `start()` создается системный поток, который будет запущен в методе `run()`. Кстати, было бы неплохо предусмотреть выход из потока. Этот пример небольшой, и проблем с ним не возникнет, а после разбора листинга 7.5 мы рассмотрим простой механизм, останавливающий системный поток и предусматривающий выход из бесконечного цикла `while` метода `run()`. В самом методе `run()` создается бесконечный цикл, в котором происходит постоянное увеличение переменной `position` на единицу, благодаря чему квадрат перемещается по оси `X` слева направо.

В методе `paint()` вас должны заинтересовать следующие строки кода:

```
int x = g.getClipWidth();
int y = g.getClipHeight();
g.setColor(0xffffffff);
g.fillRect(0,0,x,y);
```

Здесь используется *отсечение* (clipping), необходимое для корректной прорисовки графики. Если не использовать отсечения, то после того как вы нарисуете квадрат и начнете передвигать его, на экране будет рисоваться слева направо одна толстая синяя линия. То есть будет происходить постоянная перерисовка квадрата в новой позиции по оси `X`, но при этом будет оставаться и предыдущий нарисованный квадрат. Чтобы этого избежать, имеются два способа: можно стирать определенный участок экрана, а можно просто перерисовывать цвет фона всего экрана.

Такая операция в Java 2 ME называется отсечением, и для произведения этой операции используются *методы* `getClipWigth()` и `getClipHeight()`, производящие отсечение поверхности всего экрана, и *методы* `getClipX()` и `getClipY()` для более точного указания координат заданной области экрана для отсечения. По сути, использование этих методов приводит к простому обновлению всего экрана либо заданной области экрана. В этом примере мы будем использовать перерисовку всего фона экрана. Ширина и высота экрана узнаются с помощью методов `getClipWigth()` и `getClipHeight()`, полученные значения сохраняются в переменных `x` и `y`. Вызовом метода `setColor()` устанавливается белый цвет фона, и в следующей строке кода задается прямоугольник для области отсечения:

```
g.fillRect(0, 0, x, y);
```

В конце метода `paint()` рисуем синий квадрат:

```
g.fillRect(position, 40, 20, 20);
```

С каждым проходом через графический цикл цвет фона будет перерисовываться и стирать прошлую позицию квадрата, а квадрат постоянно перерисовывается

на новом месте в соответствии со значением переменной `position`, что и создает иллюзию передвижения объекта по экрану телефона.

После компиляции примера из листинга 7.5 на экране появится синий квадрат и пересечет один раз дисплей телефона слева направо. Теперь что касается бесконечного цикла `while`. В методе `run()` в реальном приложении необходимо предусмотреть выход из него, а также позаботиться о прекращении работы потока. Самый простой способ заключается в создании переменной, отвечающей за состояние цикла в начале класса `Draw`, например:

```
boolean z;
```

Дальше в методе `start()` присвоить этой переменной значение `true`.

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

А в методе `run()` использовать значение переменной `z` для входа в цикл `while`.

```
while(z)  
{  
    // код цикла  
}
```

Для логического конца создается новый *метод* `stop()` в классе `Draw`, назначая переменной `z` значения `false`.

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

Вызовем этот метод в классе `Main` в методе `destroyApp()`:

```
public void destroyApp(boolean unconditional)  
{  
    z.stop();  
}
```

7.9. Циклическое передвижение объекта по экрану

В листинге 7.5 был нарисован синий квадрат и перемещен один раз вдоль оси `X` горизонтально слева направо. Но иногда в играх необходимо *циклично передвигать* объект, используя его, например, в качестве мишени.

Возьмем за основу код из листинга 7.5, где перемещается квадрат, и сделаем так, чтобы после пересечения всего экрана и исчезновения он снова появлялся с другой стороны, создавая подобие циклического перемещения. Алгоритм решения этой задачи заключается в том, чтобы узнать, когда квадрат выйдет из области

видимости, и в тот же момент нарисовать его с другой стороны экрана и вновь переместить по экрану. Для этого создадим переменную `end` и присвоим ей значение окончания экрана, найденное методом `getWidth()` (движение происходит по ширине экрана).

```
int end = getWidth();
```

В методе `run()` в самом начале цикла `while` будем производить постоянное сравнение позиции квадрата с окончанием экрана:

```
if (position > end)
{
    position = 0;
}
```

Как только квадрат будет выходить из области экрана, его позиция обнулится и квадрат снова будет нарисован в первоначальной позиции, что зациклит движение квадрата. В листинге 7.6 представлен исходный код, решающий эту задачу, на компакт-диске пример находится в папке `\Code\Chapter7\Listing7Listing7_6`.

```
/**
```

Листинг 7.6

Класс `Main` и класс `Draw`

```
*/
```

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

```
public class Main extends MIDlet implements
CommandListener
```

```
{
    // команда выхода из программы
    private Command exitMidlet = new Command("Выход",
        Command.EXIT, 0);
```

```
public void startApp()
```

```
{
    // создаем объект класса Draw
    Draw dr = new Draw();
    // запускаем поток
    dr.start();
    // добавляем команду выхода
    dr.addCommand(exitMidlet);
    dr.setCommandListener(this);
    Display.getDisplay(this).setCurrent(dr);
}
```

```
public void pauseApp() {}
```

```
public void destroyApp(boolean unconditional) {}
```



```
public void commandAction(Command c, Displayable d)
{
    if (c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}

/**
класс Draw определен в файле Draw.Java
циклическое появление квадрата
*/

import javax.microedition.lcdui.*;

public class Draw extends Canvas implements Runnable
{
    // позиция для перемещения квадрата
    int position = 10;
    // узнаем ширину экрана
    int end = getWidth();
    // конструктор
    public Draw()
    { super (); }

    public void start ()
    {
        // создаем и запускаем поток
        Thread t = new Thread(this) ;
        t.start () ;
    }
    // метод run интерфейса Runnable

    public void run()
    {
        // бесконечный цикл
        while (true)
        {
            // сравниваем позицию квадрата
            if (position > end)
            {
                // обнуляем позицию квадрата
            }
        }
    }
}
```

```

        position = 0;
    }
    // увеличиваем позицию на 1
    position++;
    // обновляем экран
    repaint();
    // останавливаем цикл на 20 миллисекунд
    try { Thread.sleep(20); }
    catch (Java.lang.InterruptedException zxz) {}
}

public void paint(Graphics g)
{
    // вычисляем область для перерисовки экрана
    int x = g.getClipWidth();
    int y = g.getClipHeight();
    // устанавливаем белый цвет фона
    g.setColor(0xffffffff);
    // назначаем перерисовку всему экрану
    g.fillRect(0,0,x,y);
    // устанавливается синий цвет квадрата
    g.setColor(0, 0, 200);
    // рисуем квадрат
    g.fillRect(position, 40, 20, 20);
}
}

```

7.10. Столкновение

В предыдущем *разделе 7.9* мы вывели на экран синий квадрат, задали ему вектор движения и перемещали квадрат горизонтально через весь экран. После того как квадрат исчезал, достигнув конца экрана, он циклично появлялся вновь с другой стороны. Следующий пример иллюстрирует *столкновение* круга и квадрата *с препятствием*, а именно окончанием экрана телефона. Оба объекта рисуются независимо друг от друга по высоте экрана и перемещаются параллельно по горизонтали слева направо. По достижении конца экрана оба объекта отталкиваются от конца экрана и начинают движение в обратном направлении. Посмотрите на листинг 7.7, где приводится код программы, осуществляющий эти действия. Эта программа состоит из двух файлов - Main.java и Draw.java.

Листинг 7.7

Класс Main и класс Draw

*/

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

public class Main extends MIDlet implements
CommandListener
{
    // команда выхода из программы
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0);

    public void startApp()
    {
        // создаем объект класса Draw
        Draw dr = new Draw();
        // запускаем поток
        dr.start();
        // добавляем команду выхода
        dr.addCommand(exitMidlet);
        dr.setCommandListener(this);
        Display.getDisplay(this).setCurrent(dr);
    }

    public void pauseApp() {}
    public void destroyApp(boolean unconditional){}

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

/**
класс Draw определен в файле Draw.Java
рисует круг и квадрат
*/

import javax.microedition.lcdui.*;
public class Draw extends Canvas implements Runnable
{
```



```
// позиция для перемещения квадрата и круга
int position = 0;
// узнаем ширину экрана
int endX = getWidth();
// конструктор
public Draw()
{ super(); }

public void start()
{
    // создаем и запускаем поток
    Thread t = new Thread(this);
    t.start();
}
// метод run интерфейса Runnable
public void run()
{
    // бесконечный цикл
    while (true)
    {
        // сравниваем позицию
        if (position > endX)
        {
            // уменьшаем позицию
            position = endX--;
        }
        // увеличиваем позицию на 1
        position++;
        // обновляем экран
        repaint();
        // останавливаем цикл на 20 миллисекунд
        try { Thread.sleep(20); }
        catch (Java.lang.InterruptedException zxz) {}
    }
}

public void paint(Graphics g)
{
    // вычисляем область для перерисовки экрана
    int x = g.getClipWidth();
    int y = g.getClipHeight();
    // устанавливаем желтый цвет фона
    g.setColor(0xffff00);
}
```

```
// назначаем перерисовку всему экрану
g.fillRect(0,0,x,y);
// устанавливается синий цвет квадрата
g.setColor(0, 0, 200);
// рисуем квадрат
g.fillRect(position, 40, 20, 20);
// устанавливается красный цвет круга
g.setColor(250, 0, 0);
// рисуем круг
g.fillArc(position, 10, 20, 20, 45, 360);
}
}
```

ИСХОДНЫЙ КОД листинга вы найдете на компакт-диске в папке **\Code\Chapter7\Listing7Listing7_7\src**. В файле `Main.java` создается объект класса `Draw`, запускается системный поток и отражается текущий экран. Файл `Draw.java` содержит полную реализацию класса `Draw`. Начальные позиции квадрата и круга определяются переменной `position`, которой задано нулевое значение. Для того чтобы определить момент столкновения, необходимо знать точку или координаты препятствия. В этом примере как препятствие применяется конец экрана с правой стороны, поэтому с помощью метода `getWidth()` находится конец экрана.

```
int endX = getWidth();
```

Впоследствии значение переменной `endX` будет использовано для определения столкновения.

Конструктор класса `Draw` обращается к конструктору своего суперкласса `Canvas`. В методе `run()` и цикле `while` с помощью оператора отношения `if` происходит сравнение текущей позиции двух объектов и конца экрана.

```
if (position > endX)
{
    position = endX--;
}
```

После того как позиция обоих объектов становится больше значения переменной `endX`, то есть квадрат и круг достигают конца экрана, происходит уменьшение переменной `position`. Это приводит к движению объектов в обратном направлении.

После компиляции примера на экране появятся два движущихся слева направо объекта. Мы использовали препятствие в виде одной стороны экрана, но точно таким же образом можно определить столкновение для всех сторон экрана или просто для статических объектов. Выполните эти упражнения в качестве домашнего задания, а также внимательно посмотрите на процесс работы примера из листинга 7.7, квадрат и круг все-таки не отталкиваются от кромки экрана, а исчезают. Это происходит, потому что опорная позиция объектов находится в левом верхнем углу

и столкновение происходит именно с позицией объектов. Вы знаете размер круга и квадрата, подумайте, как можно определить более точную точку столкновения объектов и кромки экрана.

7.11. Перемещение объекта с помощью клавиш

Перемещение объекта по экрану телефона *с помощью клавиш* телефона - это, пожалуй, самое главное действие в играх. Для передвижения нужно воспользоваться методом `keyPressed()` и описать код, производящий обработку событий, получаемых с клавиш телефона. При реализации метода `keyPressed()` сначала необходимо вызвать метод `getGameAction()` для получения ключевого кода с последующей его обработкой. Для обработки полученного ключевого кода можно применить оператор `switch`, и тогда *метод* `keyPressed()` может принять следующий вид:

```
protected void keyPressed(int keyCode)
{
    // получаем игровые события
    int act = getGameAction(keyCode);
    // обработка событий
    switch(act)
    {
        case Canvas.LEFT:
            // движение влево
            break;
        case Canvas.RIGHT:
            // движение вправо
            break;
        case Canvas.UP:
            // движение вверх
            break;
        case Canvas.DOWN:
            // движение вниз
            break;
        default:
            break;
    }
}
```

Код обработки для клавиш зависит от того, что именно вы желаете сделать с объектом. Давайте возьмем за основу код из листинга 7.5, где был нарисован квадрат, и модернизируем его. Выведем синий квадрат на экран, для этого создадим

две переменные `positionX` и `positionY`, отвечающие за точку вывода квадрата на экран, и присвоим им значения, близкие к центру экрана.

```
int positionX = getWidth()/2;
int positionY = getHeight()/2;
```

При нажатии клавиш можно воспользоваться значениями переменных `positionX` и `positionY` для перемещения квадрата по экрану, увеличивая или уменьшая значения этих двух переменных. Значение, на которое вы будете увеличивать или уменьшать переменные, фактически будет обозначать скорость перемещения квадрата по экрану. В листинге 7.8 дается полный код программы перемещения элемента по экрану с помощью клавиш, который можно также найти на компакт-диске в папке `\Code\Chapter7\Listing7Listing7_8\src`.

```
/**
```

Листинг 7.8

Класс `Main` и класс `Draw`

```
*/
```

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

```
public class Main extends MIDlet implements
CommandListener
```

```
{
// команда выхода из программы
private Command exitMidlet = new Command("Выход",
Command.EXIT, 0);
```

```
public void startApp()
{
    // создаем объект класса Draw
    Draw dr = new Draw();
    // запускаем поток
    dr.start();
    // добавляем команду выхода
    dr.addCommand(exitMidlet);
    dr.setCommandListener(this);
    Display.getDisplay(this).setCurrent(dr);
}
```

```
public void pauseApp(){}
public void destroyApp(boolean unconditional){}
public void commandAction(Command c, Displayable d)
{
    if (c == exitMidlet)
```



```
{
    destroyApp(false);
    notifyDestroyed();
}
}
}

/**
класс Draw определен в файле Draw.Java
передвижение квадрата с помощью клавиш телефона
*/
import javax.microedition.lcdui.*;
public class Draw extends Canvas implements Runnable
{
    // устанавливаем квадрат в центр экрана
    int positionX = getWidth()/2;
    // устанавливаем квадрат в центр экрана
    int positionY = getHeight()/2;
    // конструктор
    public Draw()
    { super(); }

    public void start()
    {
        // создаем и запускаем поток
        Thread t = new Thread(this);
        t.start();
    }
    // метод run интерфейса Runnable
    public void run()
    {
        // бесконечный цикл
        while (true)
        {
            // обновляем экран
            repaint();
            // останавливаем цикл на 20 миллисекунд
            try { Thread.sleep(20); }
            catch (Java.lang.InterruptedException zxz) {}
        }
    }
    public void paint(Graphics g)
```

```
{
    // вычисляем область для перерисовки экрана
    int x = g.getClipWidth();
    int y = g.getClipHeight();
    // устанавливаем белый цвет фона
    g.setColor(0xffffffff);
    // назначаем перерисовку всему экрану
    g.fillRect(0,0,x,y);
    // устанавливается зеленый цвет квадрату
    g.setColor(0, 0, 200);
    // рисуем квадрат
    g.fillRect(positionX, positionY, 20, 20);
}
protected void keyPressed(int keyCode)
{
    // скорость передвижения
    int speed = 3;
    // получаем игровые события
    int act = getGameAction(keyCode);
    // обработка событий
    switch(act)
    {
        // движение влево
        case Canvas.LEFT:
            positionX -= speed;
            break;
        // движение вправо
        case Canvas.RIGHT:
            positionX += speed;
            break;
        // движение вверх
        case Canvas.UP:
            positionY -= speed;
            break;
        // движение вниз
        case Canvas.DOWN:
            positionY += speed;
            break;
        default:
            break;
    }
}
```

$$\}$$

Часть III

Пишем свою первую игру

Глава 8. Игровые классы

Глава 9. Формируем каркас классов

Глава 10. Добавляем в игру меню

Глава 11. Игровые карты

Глава 12. Перемещение корабля по экрану

Глава 13. Искусственный интеллект

Глава 14. Движение спрайтов

Глава 15. Игровые столкновения

Глава 16. Звуковые эффекты

Глава 17. Работа с памятью

[**http://palata-x.narod.ru**](http://palata-x.narod.ru)

<http://palata-x.narod.ru>

Глава 8. Игровые классы

С этой части книги мы начнем работу над своей игрой, но прежде чем переходить непосредственно к разработке, я предлагаю детально изучить все игровые классы платформы Java 2 ME, а также познакомиться с техникой программирования игр. Эта тема достаточно объемная, поэтому в ходе этой главы мы сначала освоим все начальные премудрости в программировании игр, а уже со следующей главы перейдем к написанию игры. В итоге к концу книги вы научитесь создавать полноценные игры для мобильных телефонов на языке программирования Java.

На сегодняшний день почти все телефоны, смартфоны и коммуникаторы имеют поддержку технологии Java 2 ME. Эта платформа позиционируется, конечно, в большей степени как игровая платформа для мобильных устройств, но немало создано и простых программ. При разработке игр под профиль MIDP 1.0 программист сталкивается с массой проблем в виде написания большого количества собственных классов для создания игрового процесса, рисования графики, уровней и многого другого. В профиль MIDP 2.0 добавлено пять игровых классов, значительно упрощающих создание мобильных игр, это:

- `GameCanvas` - абстрактный класс, содержащий основные элементы игрового интерфейса;
- `Layer` - абстрактный класс, отвечающий за уровни, представляемые в игре;
- `LayerManager` - менеджер уровней;
- `Sprite` - представляет на экране спрайтовое изображение;
- `TiledLayer` - отвечает за создание фоновых изображений.

Все вышеперечисленные классы доступны только в профиле MIDP 2.0. Также можно воспользоваться этими классами и в программировании телефонов Siemens под профиль MIDP 1.0, импортировав пакет `com.siemens.mp.colorgame`. Компания Siemens входит в экспертную группу MIDP Expert Group, и, по всей видимости, на базе уже имеющихся игровых классов от компании Siemens были созданы игровые классы для профиля MIDP 2.0.

При использовании игровых классов профиля MIDP 2.0 построение мобильной игры основано на системе уровней. Формируя игру, программист создает необходимое ему количество уровней. Каждый из уровней содержит набор графических элементов, например можно создать уровень с фоновым изображением, уровень с препятствиями, уровень с игровыми бонусами, уровень с главным персонажем игры. После этого все имеющиеся уровни komponуются воедино, накладываются один на другой и прорисовываются на экране телефона. Контроль над всеми уровнями осуществляется при помощи класса `LayerManager` - менеджера уровней.

Обилие методов, предоставляемых игровыми классами, позволяет отслеживать всевозможные столкновения, перемещения, анимацию и множество других функций, что значительно упрощает процесс создания мобильных игр. В этой главе сначала вы познакомитесь с устройством всех игровых классов, а затем будет создан ряд примеров, иллюстрирующих работу и взаимодействие игровых классов профиля MIDP 2.0.

8.1. Класс `GameCanvas`

Абстрактный класс *GameCanvas* составляет основу интерфейса всей создаваемой игры. Этот класс отвечает за прорисовку экрана, улучшая механизм работы игрового цикла и не применяя при этом входных системных потоков. Весь игровой процесс может быть сосредоточен в одном цикле *метода* `run()` *интерфейса* *Runnable*.

Игровой цикл, используемый классом `GameCanvas`, отличается от аналогичного цикла, применяемого классом `Canvas`. Типичная проблема игрового цикла класса `Canvas` заключается в том, что он разбит на несколько потоков. Вывод графики происходит в одном методе, обработка событий, получаемых с клавиш, - в другом, а изменение состояния игрового процесса - в третьем. Такая модель работы может давать сбои, поэтому в классе `GameCanvas` была придумана более изящная конструкция игрового цикла, располагающаяся в одном-единственном цикле.

```
public void run()
{
    Graphics g = getGraphics();
    while(true)
    {
        // метод, обрабатывающий нажатия клавиш с телефона
        inputKey();
        // метод, прорисовывающий графику
        GameGraphics();
        // копирует графические элементы на экран
        из внеэкранного буфера
        flushGraphics();
    }
}
```

Система прорисовки игровой графики построена на использовании *двойной буферизации*, без которой невозможно создать ни одной хорошей компьютерной игры. Смысл механизма двойной буферизации заключается в использовании *внеэкранного буфера*. Вся игровая графика рисуется во вторичном, или внеэкранном, буфере и только после полного цикла отрисовки копируется непосредственно на экран с помощью *метода* `flushGraphics()`. В этом случае улучшается качество самой графики и, что самое главное, до тех пор, пока игровой процесс не обновится, обработка событий с клавиш телефона является недоступной.

Обработка событий с клавиш упрощена с помощью *метода* `getKeyState()`. Вы просто определяете нажатую клавишу и описываете соответствующие игровые

действия для нее. Изменены некоторые названия констант для клавиш телефона. Обработка событий с клавиш телефона происходит с помощью следующих констант:

- `static int DOWN_PRESSED` – движение вниз;
- `static int FIRE_PRESSED` – реализует стрельбу из оружия;
- `static int GAME_A_PRESSED` – игровая клавиша A;
- `static int GAME_B_PRESSED` – игровая клавиша B;
- `static int GAME_C_PRESSED` – игровая клавиша C;
- `static int GAME_D_PRESSED` – игровая клавиша D;
- `static int LEFT_PRESSED` – движение влево;
- `static int RIGHT_PRESSED` – движение вправо;
- `static int UP_PRESSED` – движение вверх.

Использование констант значительно упрощает работу с объектом, а с помощью метода `getKeyStates()` класса `GameCanvas` можно определить, какая именно клавиша нажата в данный момент. Проанализируем методы класса `GameCanvas`:

- `void flushGraphics()` – копирует изображение из внеэкранного буфера на экран;
- `void flushGraphics(int x, int y, int width, int height)` – копирует изображение из внеэкранного буфера на экран в заданный по размеру прямоугольник;
- `protected Graphics getGraphics()` – получает графические элементы для представления их впоследствии классом `GameCanvas`;
- `int getKeyStates()` – определяет, какая из клавиш нажата;
- `void paint(Graphics g)` – рисует графические элементы, представленные классом `GameCanvas`.

Класс `GameCanvas` создает основной цикл игрового процесса в одном потоке, наблюдает за событиями, получаемыми с клавиш телефона, на основе которых производит постоянное обновление экрана.

8.2. Класс Layer

Абстрактный класс *Layer* задает основные свойства для всех созданных уровней игры. Класс `Layer` имеет два подкласса `TiledLayer` и `Sprite`. При создании любых других подклассов класса `Layer` необходимо реализовать метод `paint()` в этих классах, чтобы иметь возможность рисовать созданные уровни на экране телефона, представляемом классом `Graphics`. С помощью методов класса `Layer` можно задавать *размер и позицию уровня*.

- `int getHeight()` – получает высоту экрана;
- `int getWidth()` – получает ширину экрана;
- `int getX()` – получает горизонтальную координату уровня;
- `int getY()` – получает вертикальную координату уровня;
- `void move(int dx, int dy)` – перемещает уровень на координаты `dx` и `dy`;

- `abstract void paint (Graphics g)` - рисует уровень;
- `void setPosition(int x,int y)` - устанавливает уровень в позицию, обозначенную в координатах x и y.

8.3. Класс TiledLayer

С помощью класса *TiledLayer* создается *фон игровой сцены*. Фоновое изображение выполняется в виде одинаковых по размеру ячеек, как показано на рис. 8.1. Количество и расположение ячеек могут варьироваться как угодно, но нумерация ячеек следует от единицы слева направо и сверху вниз. Построение сцены происходит путем загрузки исходного изображения, разбитого на ячейки, и указания индекса необходимой ячейки на игровом поле. Но прежде нужно создать объект класса *TiledLayer* с помощью конструктора, прототип которого выглядит следующим образом:

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

- `columns` - количество столбцов;
- `rows` - количество строк;
- `image` - исходное изображение;
- `tileWidth` - размер ширины ячейки в пикселях;
- `tileHeight` - размер высоты ячейки в пикселях.

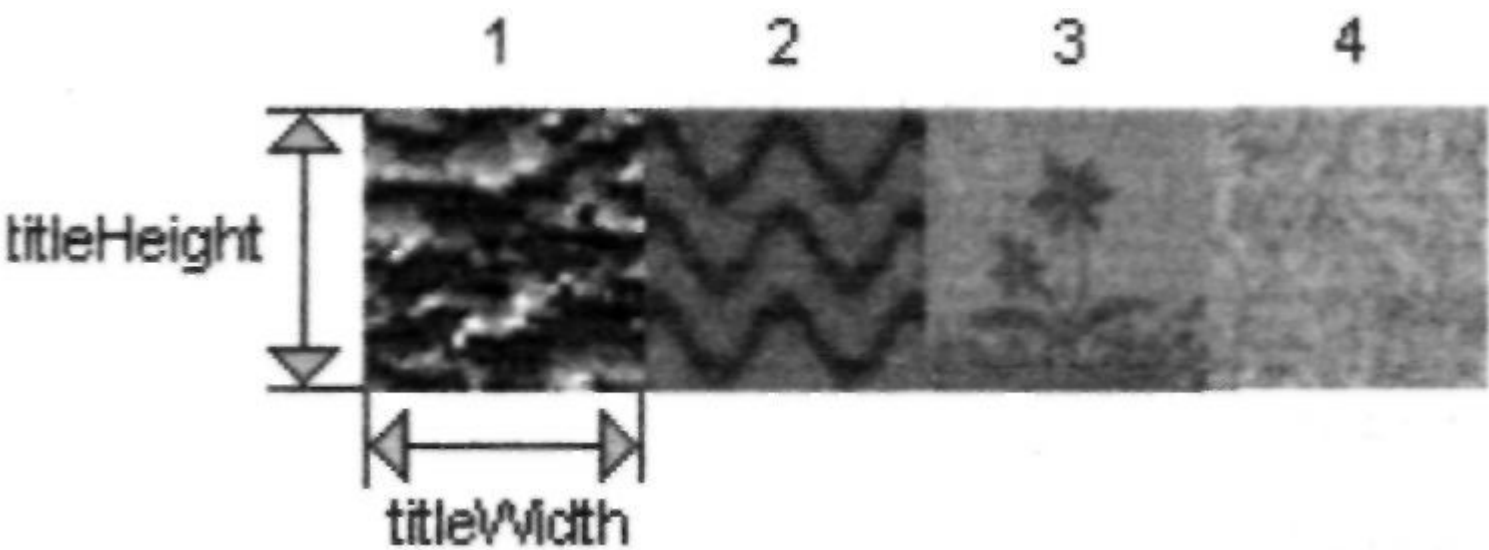


РИС. 8.1. Ячейки фонового изображения

Размеры одной ячейки по ширине и высоте могут быть разными, но все ячейки исходного изображения должны быть одинаковыми по размеру. В качестве примера возьмем за основу изображение на рис. 8.1 с шестью ячейками и предположим, что размер одной ячейки по ширине равен 10 пикселям, а по высоте - 15 пикселям, тогда загрузка изображения и создание объекта *TiledLayer* могут выглядеть следующим образом:

```
Image im = Image.createImage("/fon.png");
TiledLayer tl = new TiledLayer(3, 2, im, 10, 15);
```

Загрузив изображение и создав объект класса *TiledLayer*, вы можете приступить к разметке фона на игровом поле. Допустим, каждая из перечисленных по номеру ячеек обладает следующими характеристиками:

- 20. Камни;
- 21. Трава;
- 22. Вода;

- 23. Песок;
- 24. Воздух;
- 25. Заграждение.

А игровое поле разбито на 15 столбцов и 10 строк, тогда, создав массив данных, очень легко выполнить разметку всего поля, например таким образом:

```
int[] pole =
{
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    1, 1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    1, 1, 1, 1, 5, 5, 5, 1, 1, 5, 1, 1, 5, 5, 5,
    1, 1, 1, 1, 1, 5, 5, 1, 1, 1, 1, 1, 1, 5, 5,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 6, 6, 6, 6,
    1, 1, 1, 1, 1, 1, 4, 4, 4, 4, 6, 6, 6, 6, 6,
    4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3,
}
```

Затем весь имеющийся массив данных считывается с помощью цикла и рисуется методом `paint()` на экране телефона.

Познакомимся с методами класса `TiledLayer`:

- `int createAnimatedTile (int staticTileIndex)` - создает анимационный фон и возвращает следующий индекс ячейки;
- `int getCellHeight()` - получает высоту ячейки в пикселях;
- `int getCellWidth()` - получает ширину ячейки в пикселях;
- `int getColumns()` - получает количество колонок, на которое разбито изображение фона;
- `int getRows()` - получает количество строк, на которое разбито изображение фона;
- `void paint(Graphics g)` - рисует фон;
- `void setCell(int col, int row, int tileIndex)` - рисует заданную ячейку.

8.4. Класс LayerManager

Менеджер уровней представлен *классом* `LayerManager`. Это класс осуществляет представление любого количества *уровней* на игровом поле. Для создания объекта нужно воспользоваться конструктором класса `LayerManager`.

- `LayerManager()` - создает уровень.

А чтобы добавить уровни в игру, необходимо использовать следующие методы:

- `void append (Layer l)` - добавляет уровень в менеджер уровней;
- `Layer getLayerAt(int index)` - получает уровень с заданным индексом;
- `int getSize()` - получает общее количество уровней;

- `void insert (Layer l, int index)` - подключает новый уровень в заданный индекс;
- `void paint (Graphics g, int x, int y)` - представляет текущий менеджер уровней в заданных координатах;
- `void remove (Layer l)` - удаляет уровень из менеджера уровней.

Предположим, у вас имеются четыре уровня: фон, игрок, препятствия и артефакты. Для того чтобы связать все четыре уровня воедино, создается объект класса `LayerManager` и вызывается *метод* `append()`. Например:

```
LayerManager lm = new LayerManager();  
lm.append(fon);  
lm.append(igrok);  
lm.append(pre);  
lm.append(artf);
```

И все! Четыре перечисленных уровня отражаются на игровом поле.

8.5. Класс `Sprite`

Механизм работы с объектом *класса* `Sprite` идентичен модели работы с классом `TiledLayer`. Но если класс `TiledLayer` в основном отвечает за фоновое изображение, то с помощью класса `Sprite` рисуются на экране основные анимированные герои, космические корабли, машины, люди, звери, артефакты и т. д.

Изображение, загружаемое в игру, может быть выполнено в виде *анимационной последовательности*. Количество рисунков в анимационной последовательности не ограничено, а отсчет происходит от нуля. Располагаться рисунки могут как в виде столбца, так и в виде колонки, в зависимости от ваших предпочтений. Каждый рисунок анимационной последовательности называется *фреймом*. Фрейм может быть любого размера по ширине и высоте, но все фреймы должны быть одинаковых размеров. Размер фрейма должен быть известен, потому что он используется при создании объекта класса `Sprite`. Есть три конструктора класса `Sprite`, каждый из которых можно использовать при создании объекта класса `Sprite`:

- `Sprite(Image image)` - создает неанимированный спрайт;
- `Sprite(Image image, int frameWidth, int frameHeight)` - создает анимационный спрайт, взятый из заданного фрейма;
- `Sprite(Sprite s)` - создает спрайт из другого спрайта.

Для перехода по фреймам исходного изображения, а также для определения столкновения между объектами используются методы класса `Sprite`:

- `boolean collidesWith(Sprite s, boolean pixelLevel)` - определяет столкновение между спрайтами;
- `boolean collidesWith(TiledLayer t, boolean pixelLevel)` - определяет столкновение между спрайтом и препятствием, нарисованным при помощи класса `TiledLayer`;
- `int getFrame()` - получает текущий фрейм;

- `void nextFrame ()` - осуществляет переход на следующий фрейм;
- `void paint (Graphics g)` - рисует спрайт;
- `void prevFrame ()` - осуществляет переход на предыдущий фрейм;
- `void setFrame (int sequenceIndex)` - устанавливает заданный фрейм;
- `void setFrameSequence (int[] sequence)` - устанавливает определенную фреймовую последовательность;
- `void setImage (Image img, int frameWidth, int frameHeight)` - изменяет изображение спрайта на новое изображение;
- `void setTransform (int transform)` - производит трансформацию спрайта.
- `public void defineReferencePixel (int x, int y)` - изменяет опорную позицию спрайта, перенося ее в точку с координатами `x` и `y`.

Метод `defineReferencePixel()` изменяет опорную позицию спрайта, но для чего это нужно? Опорная позиция для спрайта задается левым верхним углом, но не всегда это удобно, поэтому опорную позицию можно перенести в центр спрайта. Например, если спрайт сделан в виде машины, то при вращении вокруг своей оси, если опорная позиция перенесена в центр, вращение будет правдоподобным. Но если не переопределить позицию, то вращение произойдет в точке левого верхнего угла спрайта, и выглядеть это будет не вполне естественно, как будто у машины пробито одно колесо. Для этого вызывается метод `defineReferencePixel()` с заданными координатами и переопределяет опорную позицию, например в центр спрайта:

```
defineReferencePixel (frameWidth/2,
frameHeight/2) ;
```

Метод `setTransform()` производит трансформацию спрайта, вращая или зеркально отображая спрайт вокруг своей оси с помощью следующих констант:

- `static int TRANS_MIRROR;`
- `static int TRANS_MIRROR_ROT180;`
- `static int TRANS_MIRROR_ROT270;`
- `static int TRANS_MIRROR_ROT90;`
- `static int TRANS_NONE;`
- `static int TRANS_ROT180;`
- `static int TRANS_ROT270;`
- `static int TRANS_ROT90.`

Посмотрите на рис. 8.2, где очень привлекательно показаны константы для различных состояний спрайта.

Рассмотрев классы `GameCanvas`, `Layer`, `Sprite`, `TiledLayer` и `LayerManager`, перейдем к практическим занятиям этой главы.

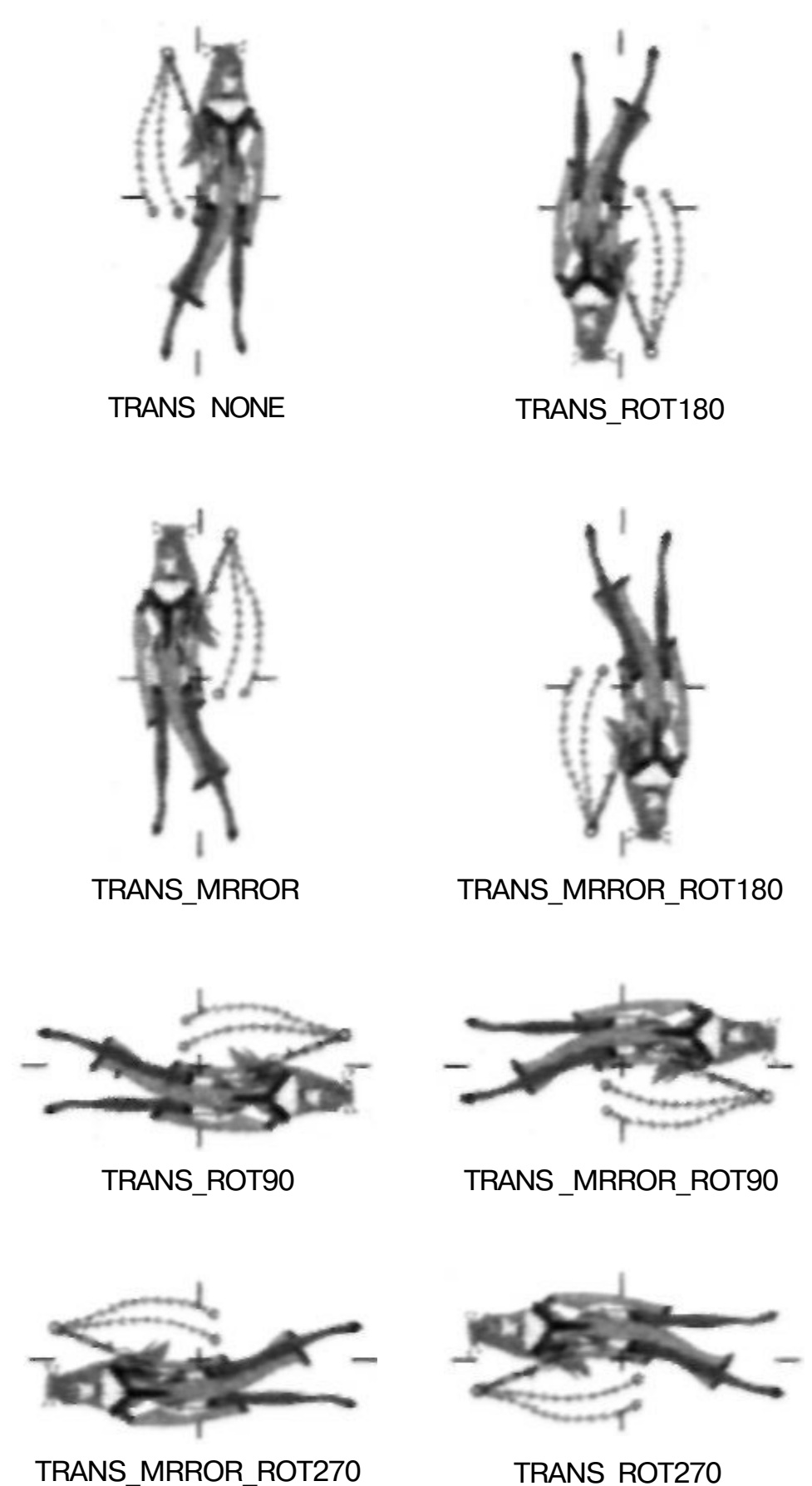


РИС. 8.2. Константы трансформации

8.6. Создание фонового изображения

С помощью класса `TiledLayer` можно создавать любое количество уровней, накладывая их друг на друга, а с помощью менеджера уровней, представленного классом `LayerManager`, - отслеживать все имеющиеся уровни. В качестве примера будет создан фон на основе элементов разметки игрового поля. *Фоновое изображение* загружается из файла `fon.png`. Само изображение выполнено в виде шести ячеек размером 15x15 пикселей.

В листинге 8.1 находится код примера, создающего фоновое изображение. Ознакомьтесь с листингом, а потом мы перейдем к анализу этого примера.

```
/**
Листинг 8.1
класс MainGame
*/

import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class MainGame extends MIDlet implements
CommandListener
{
    // команда выхода
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0);
    // объект класса MyGameCanvas
    private MyGameCanvas mr;

    public void startApp()
    {
        // обрабатываем исключительную ситуацию
        try{
            // инициализируем объект класса MyGameCanvas
            mr = new MyGameCanvas();
            // запускаем поток
            mr.start();
            // добавляем команду выхода
            mr.addCommand(exitMidlet);
            mr.setCommandListener(this);
            // отражаем текущий дисплей
            Display.getDisplay(this).setCurrent(mr);
        }catch (java.io.IOException zxz) {}
    }

    public void pauseApp() {}
}
```



```
public void destroyApp(boolean unconditional)
{
    // останавливаем поток
    if (mr != null) mr.stop();
}

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

/**
Файл MyGameCanvas.java
класс MyGameCanvas
*/

import java.io.IOException;
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class MyGameCanvas extends GameCanvas implements
Runnable
{
    // создаем объект класса TiledLayer
    private TiledLayer fonPole;
    // создаем объект класса LayerManager
    private LayerManager lm;
    // логическая переменная для выхода из цикла
    boolean z;

    public MyGameCanvas() throws IOException
    {
        // обращаемся к конструктору суперкласса Canvas
        super(true);
        // инициализируем fonPole
        fonPole = Fon();
        // создаем менеджер уровней
        lm = new LayerManager();
        // добавляем объект fonPole к уровню
```

```
        Im.append(fonPole);
    }

    public void start()
    {
        z = true;
        // создаем и запускаем поток
        Thread t = new Thread(this);
        t.start();
    }
    /* метод, создающий объект класса TiledLayer
    и загружающий фоновое изображение */
    public TiledLayer Fon()throws IOException
    {
        // загрузка изображения из файла ресурса
        Image im = Image.createImage("/fon.png");
        // создаем объект класса TiledLayer
        fonPole = new TiledLayer(/*столбцы*/10,/*строки*/10,
        /*изображение*/1т,/*ширина*/15,/*высота*/15);
        // разметка игрового поля
        int[] pole =
        {
            5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
            1, 5, 5, 5, 5, 5, 5, 5, 5, 5,
            1, 1, 5, 5, 5, 5, 5, 5, 5, 5,
            1, 1, 1, 1, 5, 5, 5, 1, 1, 5,
            1, 1, 1, 1, 1, 5, 5, 1, 1, 1,
            1, 1, 1, 1, 1, 1, 1, 1, 6, 6,
            1, 1, 1, 1, 1, 1, 6, 6, 6, 6,
            2, 4, 4, 4, 4, 4, 3, 3, 3, 3,
            2, 2, 2, 4, 4, 4, 3, 3, 3, 3,
            2, 2, 2, 4, 4, 4, 3, 3, 3, 3,
        };
        // цикл, считывающий разметку поля
        for (int i = 0; i < pole.length; i++)

            /* присваиваем каждому элементу игрового поля
            определенную ячейку изображения im*/
            fonPole.setCell(i % 10, i / 10, pole[i]);
        }
        return fonPole;
    }

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

```
public void run ()
{
    // получаем графический контекст
    Graphics g = getGraphics ();
    while (z)

        // рисуем графические элементы
        init(g);
        // останавливаем цикл на 20 миллисекунд
        try { Thread.sleep(20); }
        catch (Java.lang.InterruptedException zxz) {}
    }
}

private void init (Graphics g)
{
    // белый цвет фона для перерисовки экрана
    g.setColor(0xffffffff);
    // размер перерисовки экрана
    g.fillRect(0, 0, getWidth(), getHeight());
    // рисуем уровень с левого верхнего угла дисплея
    Im.paint(g, 0, 0);
    // двойная буферизация
    flushGraphics();
}
}
```

Листинг 8.1 состоит из двух классов `MainCanvas` и `MyGameCanvas`, находящихся в файлах `MainCanvas.java` и `MyGameCanvas.java`. Анализ листинга начнем с класса `MyGameCanvas`. В первых строках кода этого класса объявляются два объекта классов `TiledLayer` и `LayerManager`, а также логическая переменная `z`.

```
private TiledLayer fonPole;
private LayerManager Im;
boolean z;
```

Объект `fonPole` класса `TiledLayer` будет отвечать за фоновое изображение. Объект `Im` класса `LayerManager` является менеджером уровней. Логическая переменная `z` необходима для прерывания цикла в методе `run()` и для окончания системного потока, в котором происходит работа всего игрового цикла.

В конструкторе `MyGameCanvas` происходит инициализация объекта `fonPole` - класса `TiledLayer` и объекта `Im` класса `LayerManager`,. Инициализированный объект `fonPole` добавляется менеджером уровней к текущему уровню для представления на экране телефона. Обратите внимание, объект `fonPole` инициализируется с помощью метода `Fon()`.

```
Image im = Image.createImage("/fon.png");
fonPole = new TiledLayer(/*столб*/10, /*строки*/10, im, /
/*ширина*/15, /*высота*/15);
```

В этих двух строках происходят загрузка исходного изображения из файла ресурса и создание объекта `fonPole` с помощью конструктора класса `TiledLayer`.

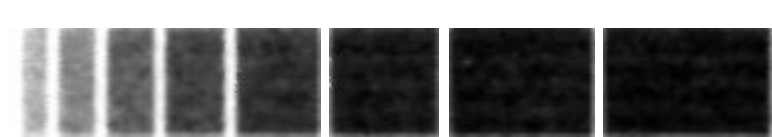
Вся разметка игрового поля выполнена в виде 10 строк и 10 столбцов. Первые два параметра конструктора класса `TiledLayer` как раз и отвечают за количество столбцов и строк. Третий параметр конструктора - это исходное изображение, выполненное в виде шести ячеек, каждая размером 15x15 пикселей. Два последних параметра конструктора класса `TiledLayer` определяют ширину и высоту ячейки. При создании объекта класса `TiledLayer` необходимо быть внимательным и указывать реальные размеры одной ячейки. Если размер одной ячейки будет, предположим, 20 x 20 пикселей, а вы обозначите как 15 x 15 пикселей, то в итоге ячейки изображения загружены не будут.

Дальше в методе `Fon()` происходит разметка игрового поля, выполненного в виде 10 столбцов и 10 строк.

```
int[] pole =
{
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    1, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    1, 1, 5, 5, 5, 5, 5, 5, 5, 5,
    1, 1, 1, 1, 5, 5, 5, 1, 1, 5,
    1, 1, 1, 1, 1, 5, 5, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 6, 6,
    1, 1, 1, 1, 1, 1, 6, 6, 6, 6,
    2, 4, 4, 4, 4, 4, 3, 3, 3, 3,
    2, 2, 2, 4, 4, 4, 3, 3, 3, 3,
    2, 2, 2, 4, 4, 4, 3, 3, 3, 3,
};
```

Все строки и столбцы состоят из элементов. Отсчет ячеек происходит от единицы и выше. Присвоение номера ячейки исходного изображения одному из элементов и организует разметку игрового поля, которое впоследствии будет рисоваться на экране. Единственное, о чем нельзя забывать, - это о размере дисплеев реальных телефонов. Если вы имеете 10 столбцов и размер каждой ячейки 15 пикселей по ширине, то в итоге ваше игровое поле в ширину будет $10 \times 15 = 150$ пикселей. Не каждый телефон может похвастаться таким разрешением, поэтому при создании игрового поля нужно учитывать эти нюансы. Вслед за разметкой игрового поля в методе `Fon()` происходит считывание всех элементов с помощью цикла `for`.

```
for(int i = 0; i < pole.length; i++)
{
    fonPole.setCell (i % 10, i / 10, pole[i]);
}
```

Присвоение номера ячейки определенному элементу происходит при помощи метода `setCell()`. В этом методе первый параметр - номер столбца, второй - номер строки и последний - номер ячейки изображения. Здесь главное - не запутаться, потому что номера столбцов и строк начинаются с нуля из-за того, что это обычный массив данных, а все массивы, как вы знаете, ведут свой отсчет с нуля, тогда как ячейка исходного изображения начинается с единицы. Сразу возникает вопрос, а почему не произвести отсчет тоже с нуля, чтобы не было путаницы? Дело в том, что отсчет и производится с нуля, но число ноль - это своего рода зарезервированное значение для ячеек изображения. Нулевое значение может использоваться, но оно не загружает ничего, поэтому отсчет ячеек ведется с единицы. С методом `Fon()` мы разобрались, перейдем к методу `init()`.

```
g.setColor(0xffffffff);  
g.fillRect(0, 0, getWidth(), getHeight());
```

В этих строках кода происходит постоянная перерисовка фона экрана. Эти действия вы производили в *главе 7*, когда разбирали механизм отсечения.

С помощью метода `paint()` рисуется уровень. Начало точки вывода уровня задано точкой 0,0, что соответствует началу системы координат.

И последний метод `flushGraphics()` осуществляет двойную буферизацию, копируя графические элементы из внеэкранного буфера на экран.

В методе `run()` происходит остановка игрового цикла. Перед тем как цикл создается с помощью оператора `while`, методом `getGraphics()` происходит получение графического контекста, что и является одним из достоинств механизма игрового цикла в профиле MIDP 2.0.

В файле `MainGame.java` создается основной класс мидлета `MainGame`. В методе `startApp()` производится создание объекта рассмотренного класса `MyGameCanvas`, добавляется команда выхода, запускается системный поток и отражается текущий дисплей. В методе `destroyApp()` происходит остановка потока методом `stop()` класса `MyGameCanvas`.

8.7. Обработка событий с клавиш телефона

В профиле MIDP 2.0 предусмотрена улучшенная обработка событий, получаемых с клавиш телефона. Используя метод `getKeyState()`, можно определять состояние клавиш телефона и действовать адекватным образом. В демонстрационном примере к этому разделу мы выведем на экран мячик, созданный при помощи класса `MySprite`, который является подклассом класса `Sprite`. В листинге 8.2 представлен код примера, в котором на экране рисуется синий мяч, а с помощью клавиш телефона **Up**, **Down**, **Left** и **Right** этот мяч можно передвигать по экрану. Листинг 8.2 состоит из трех классов: `MainGame`, `MyGameCanvas` и `MySprite`, - расположенных в трех разных файлах: `MainGame.java`, `MyGameCanvas.java` и `MySprite.java`.

```
/**
```

ЛИСТИНГ 8.2

```
класс MainGame
```

```
*/
```

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

```
import javax.microedition.midlet.*;
```

```
public class MainGame extends MIDlet implements  
CommandListener
```

```
{
```

```
// команда выхода
```

```
private Command exitMidlet = new Command("Выход",  
Command.EXIT, 0);
```

```
// объект класса MyGameCanvas
```

```
private MyGameCanvas mr;
```

```
public void startApp()
```

```
    // обрабатываем исключительную ситуацию
```

```
    try{
```

```
        // инициализируем объект класса MyGameCanvas
```

```
        mr = new MyGameCanvas();
```

```
        // запускаем поток
```

```
        mr.start();
```

```
        // добавляем команду выхода
```

```
        mr.addCommand(exitMidlet);
```

```
        mr.setCommandListener(this);
```

```
        // отражаем текущий дисплей
```

```
        Display.getDisplay(this).setCurrent(mr);
```

```
    }catch (java.io.IOException zxz) {};
```

```
}
```

```
public void pauseApp() {}
```

```
public void destroyApp(boolean unconditional)
```

```
    // останавливаем поток
```

```
    if (mr != null) mr.stop();
```

```
}
```

```
public void commandAction(Command c, Displayable d)
```

```
{
```

```
    if (c == exitMidlet)
```

```
    {
```

```
        destroyApp(false);
```

```
        notifyDestroyed();
    }
}

/**файл MyGameCanvas.java
класс MyGameCanvas
*/

import java.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class MyGameCanvas extends GameCanvas implements
Runnable
{
    // создаем объект класса MySprite
    private MySprite bol;
    // создаем объект класса LayerManager
    private LayerManager lm;
    // логическая переменная
    boolean z;

    public MyGameCanvas() throws IOException
    {
        // обращаемся к конструктору суперкласса Canvas
        super(true);
        // загружаем изображение
        Image im = Image.createImage("/bol.png");
        // инициализируем объект bol
        bol = new MySprite(im, 23, 23);
        // выбираем позицию в центре экрана
        bol.setPosition(getWidth()/2, getHeight()/2);
        // инициализируем менеджер уровней
        lm = new LayerManager();
        // добавляем объект bol к уровню
        lm.append(bol);
    }

    public void start()
    {
        z = true;
        // создаем и запускаем поток
```

```
Thread t = new Thread(this);
t.start();
}
// останавливаем поток
public void stop(){-z = false; }

public void run()
{
    // получаем графический контекст
    Graphics g = getGraphics ();
    while (z)
    {
        // обрабатываем события с клавиш телефона
        inputKey();
        // рисуем графические элементы
        init(g);
        // останавливаем цикл на 20 миллисекунд
        try { Thread.sleep(20); }
        catch (Java.lang.InterruptedException zxz) {}
    }
}

private void inputKey()
{
    // определяем нажатую клавишу
    int keyStates = getKeyStates();
    // код обработки для левой нажатой клавиши
    if ((keyStates & LEFT_PRESSED) != 0) bol.moveLeft();
    // код обработки для правой нажатой клавиши
    if ((keyStates & RIGHT_PRESSED) != 0) bol.moveRight();
    // код обработки для клавиши вверх
    if ((keyStates & UP_PRESSED) != 0) bol.moveUp();
    // код обработки для клавиши вниз
    if ((keyStates & DOWN_PRESSED) != 0) bol.moveDown();
}

private void init(Graphics g)
{
    // белый цвет фона
    g.setColor(0xffffffff);
    // перерисовываем экран
    g.fillRect(0, 0, getWidth(), getHeight());
    // рисуем уровень в точке 0,0
    Im.paint(g, 0, 0);
}
```



```
// двойная буферизация
flushGraphics();
};
}

/**
файл MySprite.java
класс MySprite
*/

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class MySprite extends Sprite
{
    // конструктор
    public MySprite(Image image, int fw, int fh)
    {
        // обращаемся к конструктору суперкласса
        super(image, fw, fh);
    }
    // метод для клавиши Left
    public void moveLeft()
    {
        // передвигаем спрайт
        move(-1,0);
    }

    // метод для клавиши Right
    public void moveRight()
    {
        // передвигаем спрайт
        move(1,0);
    }
    // метод для клавиши Up
    public void moveUp()
    {
        // передвигаем спрайт
        move(0,-1);
    }
    // метод для клавиши Down
    public void moveDown()
    {
        // передвигаем спрайт
        move(0,1);
    }
}
```

```
}
}
```

В файле `MySprite.java` находится класс `MySprite`, с которого и начнем рассмотрение листинга. Конструктор класса `MySprite` обращается к конструктору своего суперкласса `Sprite`.

```
super(image, fw, fh);
```

Этот конструктор имеет три параметра - это исходное изображение спрайта, ширина и высота фрейма. Спрайт, как вы помните, может иметь анимационную последовательность, поэтому необходимо точно знать ширину и высоту одного фрейма, при обязательном условии, что все фреймы спрайта должны быть одного размера. В этом примере мы не используем анимационной последовательности, и можно было обойтись и более простым конструктором суперкласса `Sprite`.

Для передвижения спрайта по экрану телефона созданы методы `moveLeft()`, `moveRight()`, `moveUp()` и `moveDown()`, в основу которых положены вызовы метода `move()`. Метод `move()` имеет два параметра - это координаты по осям *X* и *Y*. Задавая необходимое смещение на 1, 2, 3 и более пикселей по одной из осей, вы будете производить движение объекта по экрану.

Класс `MyGameCanvas` работает по схеме, использованной в *разделе 8.6*, с той лишь разницей, что вместо фонового изображения загружается спрайт в виде мячика. В конструкторе класса `MyGameCanvas` происходит загрузка исходного изображения `bol.png`, это и есть наш мячик, или спрайт. Спрайт размером 23 x 23 пикселя. При создании объекта `bol` класса `MySprite`

```
bol = new MySprite(im, 23, 23);
```

используется изображение `bol.png`, затем в конструкторе класса `MyGameCanvas` происходит выбор позиции для первоначальной отрисовки спрайта на экране с помощью метода `setPosition()`. Мячик рисуется в центре экрана и добавляется методом `append()` к уровню.

В методе `run()` в игровом цикле происходит вызов двух методов. Метод `init()` производит рисование всех графических элементов с помощью менеджера уровней, а метод `inputKey()` осуществляет обработку нажатий клавиш телефона.

```
private void inputKey()
{
    int keyStates = getKeyStates();
    if ((keyStates & LEFT_PRESSED) != 0) bol.moveLeft();
    if ((keyStates & RIGHT_PRESSED) != 0) bol.moveRight();
    if ((keyStates & UP_PRESSED) != 0) bol.moveUp();
    if ((keyStates & DOWN_PRESSED) != 0) bol.moveDown();
}
```

В методе `inputKey()` происходит определение нажатой клавиши посредством метода `getKeyState()`. Весь остальной код в методе `inputKey()` использует

оператор `if` для обработки нажатых клавиш, вызывая соответствующие методы `moveLeft()`, `moveRight()`, `moveUp()` или `moveDown()` для перемещения объекта по экрану.

В классе `MainGame` из файла `MainGame.java` создается объект класса `MyGameCanvas`, запускается системный поток и отражается текущий экран.

В этом примере использовался спрайт файла ресурса `bol.png`, состоящий из одного фрейма размером 23 x 23 пикселя. В следующем разделе мы рассмотрим технику анимации спрайтов в играх и будем использовать спрайт, состоящий уже из нескольких фреймов.

8.8. Анимация в игровом процессе

Анимация в игровом процессе строится на основе последовательной цепочки рисунков. Как вы уже знаете, отдельно взятый рисунок из анимационной последовательности в Java 2 ME называется фреймом. Для того чтобы осуществить плавную анимацию в игре, необходимо выполнить ряд сменяющих друг друга рисунков. Посмотрите на рис. 8.3, где изображен матрос с флажками.



РИС. 8.3. Анимационная последовательность

На рис. 8.3 все фреймы выполнены в виде горизонтальной цепочки, но это не обязательное условие, можно расположить фреймы любым удобным образом. Не забывайте о том, что отсчет начинается с нуля и идет слева направо и сверху вниз.

Анимация повсеместно используется в компьютерных и мобильных играх. Вы, наверное, замечали, что при перемещении в игре персонажа он производит ряд повторяющихся движений, создавая видимость игрового цикла. Такие элементарные движения - следствие перехода по имеющимся фреймам исходного изображения. Для этих целей в Java 2 ME имеются специальные методы класса `Sprite`.

Метод `nextFrame()` позволяет осуществить переход по всем имеющимся фреймам исходного изображения. Как только достигается последний фрейм, то автоматически происходит переход к первому фрейму с последующим переходом по всей имеющейся последовательности, что обеспечивает цикличность анимации.

Давайте рассмотрим пример, использующий анимационную последовательность, изображенную на рис. 8.3. В этом примере на экран выводится изображение матроса и осуществляется циклический переход по всем имеющимся фреймам, создавая эффект движения матроса, который с помощью семафорной азбуки передает слово «анимация».

В листинге 8.3, а также на компакт-диске в папке `\Code\Chapter8\Listing8_3\src` дается код примера, иллюстрирующего работу анимационной последовательности.

```
/*ЛИСТИНГ 8.3
```

```
класс MainGame
```

```
*/
```

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

```
import javax.microedition.midlet.*;
```

```
public class MainGame extends MIDlet implements  
CommandListener
```

```
{
```

```
// команда выхода
```

```
private Command exitMidlet = new Command("Выход",  
Command.EXIT, 0);
```

```
// объект класса MyGameCanvas
```

```
private MyGameCanvas mr;
```

```
public void startApp()
```

```
{
```

```
    // обрабатываем исключительную ситуацию
```

```
    try{
```

```
        // инициализируем объект класса MyGameCanvas
```

```
        mr = new MyGameCanvas();
```

```
        // запускаем поток
```

```
        mr.start();
```

```
        // добавляем команду выхода
```

```
        mr.addCommand(exitMidlet);
```

```
        mr.setCommandListener(this);
```

```
        // отражаем текущий дисплей
```

```
        Display.getDisplay(this).setCurrent(mr);
```

```
    }catch (java.io.IOException zzx) {};
```

```
}
```

```
public void pauseApp() {}
```

```
public void destroyApp(boolean unconditional)
```

```
{
```

```
    // останавливаем поток
```

```
    if(mr != null) mr.stop();
```

```
public void commandAction(Command c, Displayable d)
```

```
{
```

```
    if (c == exitMidlet)
```



```
        destroyApp(false) ;
        notifyDestroyed() ;
    }
}

/**
файл MyGameCanvas.java
класс MyGameCanvas
*/

import java.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class MyGameCanvas extends GameCanvas implements
Runnable
{
    // создаем объект класса MySprite
    private Matros matros;
    // создаем объект класса LayerManager
    private LayerManager lm;
    // логическая переменная
    boolean z;

    public MyGameCanvas() throws IOException
    {
        // обращаемся к конструктору суперкласса Canvas
        super(true);
        // загружаем изображение
        Image im = Image.createImage("/matros.png");
        // инициализируем объект matros
        matros = new Matros(im, 94, 100);
        // выбираем позицию
        matros.setPosition(30, 30);
        // инициализируем менеджер уровней
        lm = new LayerManager();
        // добавляем объект матроса к уровню
        lm.append(matros);
    }

    public void start()
    {
        z = true;
    }
}
```

```
// создаем и запускаем поток
Thread t = new Thread(this);
t.start();
}
// останавливаем поток
public void stop(){ z=false; }

public void run()
{
    // получаем графический контекст
    Graphics g = getGraphics();
    while (z)
    {
        // рисуем графические элементы
        init(g);
        // останавливаем цикл
        try { Thread.sleep(250); }
        catch (Java.lang.InterruptedException zxz) {}
    }
}

private void init(Graphics g)
{
    // белый цвет фона
    g.setColor(0xffffffff);
    // перерисовываем экран
    g.fillRect(0, 0, getWidth(), getHeight());
    // рисуем уровень в точке 0,0
    Im.paint(g, 0, 0);
    // рисуем анимацию
    matros.Animation();
    // двойная буферизация
    flushGraphics();
};
}

/**файл Matros.java
класс Matros
*/

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

```
public class Matros extends Sprite
{
    // конструктор
    public Matros(Image image, int fw, int fh)

        // обращаемся к конструктору суперкласса
        super(image, fw, fh);
}
// метод, осуществляющий анимацию
public void Animation()
{
    // вызываем следующий фрейм
    nextFrame();
}
}
```

В листинге 8.3 нам интересны несколько моментов. В классе `Matros`, являющемся подклассом класса `Sprite`, создается метод `Animation()`, который выглядит следующим образом:

```
public void Animation()
{
    nextFrame();
}
```

Метод `Animation()` осуществляет тот самый последовательный переход по имеющимся фреймам исходного изображения. В классе `MyGameCanvas` происходит создание объекта класса `Matros`:

```
private Matros matros;
```

Затем в конструкторе класса `MyGameCanvas` загружается изображение матроса и инициализируется объект `matros`.

```
Image im = Image.createImage("/matros.png");
matros = new Matros(im, 94, 100);
```

Размер одного фрейма с матросом равен 94 x 100 пикселей, поэтому указывается размер именно одного фрейма. По умолчанию загружается самый первый фрейм изображения, но можно использовать метод `setFrame()` для установки необходимого фрейма из анимационной последовательности. В методе `Graphics()` класса `MyGameCanvas` происходит вызов метода `Animation()`:

```
matros.Animation();
```

Это в итоге приводит к цикличному перебору всех имеющихся фреймов. Откомпилируйте код из листинга 8.3 и посмотрите работу этого примера. На экране телефона матрос с помощью семафорной азбуки передает слово «анимация».

8.9. Столкновение объектов

Практически во всех играх приходится обрабатывать события, связанные со *столкновением двух объектов* или с препятствием. В профиле MIDP 2.0 существуют три отличных метода, отслеживающих факт столкновения. Все три метода определены в классе `Sprite`.

- `collidesWith(Image image, int x, int y, Boolean pixelLevel)` - определяет факт столкновения со статическим изображением;
- `collidesWith(Sprite s, Boolean pixelLevel)` - определяет столкновение между двумя объектами класса `Sprite`;
- `collidesWith(TiledLayer t, Boolean pixelLevel)` - отслеживает столкновение между объектом класса `Sprite` и объектом класса `TiledLayer`.

Используя эти методы, можно обрабатывать всевозможные ситуации, связанные со столкновением. Как правило, при столкновении должны произойти какие-то события, чаще всего связанные с изменением первоначального состояния объекта. Это может быть уничтожение объекта, его перемещение, уменьшение или увеличение. В основе изменения состояния объекта положен перебор имеющихся фреймов анимационной последовательности или перерисовка изображения на новом месте. Для этих операций в классе `Sprite` имеются несколько методов. С одним из методов - `nextFrame()` - вы уже познакомились в *разделе 8.8*, рассмотрим оставшиеся методы:

- `prevFrame()` - с помощью этого метода происходит переход к предыдущим фреймам изображения. Этот метод подобен методу `nextFrame()`, только переход осуществляется в обратном порядке;
- `setFrame()` - производит установку при помощи индекса заданного фрейма из всей имеющейся последовательности фреймов;
- `setFrameSequence()` - устанавливает определенную фреймовую последовательность при помощи массива индексов в анимационной последовательности;
- `getFrame()` - узнает, какой из фреймов исходного изображения используется в текущий момент;
- `setImage()` - заменяет исходное изображение на новое. Можно использовать этот метод, например, в том случае, если объект уничтожен и его необходимо перерисовать заново.

Набора этих методов вполне достаточно для обработки различных ситуаций, возникающих в игровом процессе. Однако необходимо очень тщательно разобраться в действии всех вышеперечисленных методов. Для этого был написан исходный код примера, иллюстрирующего работу методов `nextFrame()`, `prevFrame()`, `setFrame()` и `setFrameSequence()`. В этой программе на экран выводятся четыре бомбы и синий шарик, перемещая который в разные стороны, можно произвести столкновение с четырьмя бомбами. Все бомбы являются объектами класса `MySprite`, являющимся подклассом класса `Sprite`. Метод,

обрабатывающий столкновение мяча и бомбы, использует один из четырех методов `nextFrame()`, `prevFrame()`, `setFrame()` и `setFrameSequence()` для каждой из бомб, что красочно иллюстрирует работу каждого метода. Исходные изображения бомбы и мяча выполнены в виде последовательности четырех фреймов размером 23 x 23 пикселя. Первый фрейм мяча и бомбы *является* исходным изображением, а три последующих фрейма реализованы в виде взрыва. Переход по этим фреймам создает иллюзию взрыва мячика или бомбы. Но от того, что для каждой бомбы, как мы договорились, используются различные методы, результат может оказаться неожиданным. Поэтому внимательно посмотрите на код примера в листинге 8.4 и запустите программу с компакт-диска `\Code\ Chapter8\Listing8_4\bin\Listing8_4.jad` с помощью любого эмулятора, поддерживающего профиль MIDP 2.0. Либо откомпилируйте исходный код из листинга 8.4 (он также находится на компакт-диске в папке `\Code\Listing8_4\src`) и внимательно ознакомьтесь с работой этой программы. Я уверен, что вы без труда разберетесь, что именно нужно сделать для логического завершения взрывов бомб и мяча. Алгоритм действий очень прост и может быть следующим после столкновения мяча с одной из мин. Необходимо произвести взрыв, последовательно переходя по всем фреймам, после чего, например, нарисовать бомбу и мячик заново. В листинге 8.4 предложен исходный код примера.

```
/**
Листинг 8.4
класс MainGame
*/

import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class MainGame extends MIDlet implements
CommandListener
{
    // команда выхода
    private Command exitMidlet = new Command("Выход",
Command.EXIT, 0);
    // объект класса MyGameCanvas
    private MyGameCanvas mr;

    public void startApp()
    {
        // обрабатываем исключительную ситуацию
        try{
            // инициализируем объект класса MyGameCanvas
            mr = new MyGameCanvas();
            // запускаем поток
```

```
mr.start() ;
// добавляем команду выхода
mr.addCommand(exitMidlet);
mr.setCommandListener(this);
// отражаем текущий дисплей
Display.getDisplay(this).setCurrent(mr);
} catch (java.io.IOException zxz) {};
}

public void pauseApp() {}
public void destroyApp(boolean unconditional)
{
    // останавливаем поток
    if (mr != null) mr.stop();
}

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

/**
файл MyGameCanvas.java
класс MyGameCanvas
*/
import java.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
public class MyGameCanvas extends GameCanvas implements
Runnable
{
    // создаем объект класса MySprite
    private MySprite bol;
    // создаем объект класса LayerManager
    private LayerManager lm;
    // создаем бомбы
    private MySprite bomba1, bomba2, bomba3, bomba4;
```

```
// логическая переменная
boolean z;

public MyGameCanvas() throws IOException
{
    // обращаемся к конструктору суперкласса Canvas
    super(true);
    // загружаем изображение мяча
    Image bolImage = Image.createImage("/bol.png");
    // инициализируем объект bol
    bol = new MySprite(bolImage, 23, 23);
    // выбираем позицию в центре экрана
    bol.setPosition(getWidth()/2, getHeight()/2);
    // загрузка изображения бомбы
    Image bombaImage = Image.createImage("/bomba.png");
    // инициализируем первую бомбу
    bomba1 = new MySprite(bombaImage, 23, 23);
    // выбираем позицию для первой бомбы
    bomba1.setPosition(10, 10);
    // инициализируем вторую бомбу
    bomba2 = new MySprite(bombaImage, 23, 23);
    // выбираем позицию для второй бомбы
    bomba2.setPosition( getWidth()-30, 10);
    // инициализируем третью бомбу
    bomba3 = new MySprite(bombaImage, 23, 23);
    // выбираем позицию для третьей бомбы
    bomba3.setPosition(10, getHeight()-40);
    // инициализируем четвертую бомбу
    bomba4 = new MySprite(bombaImage, 23, 23);
    // выбираем позицию для четвертой бомбы
    bomba4.setPosition(getWidth()-30, getHeight()-40);
    // инициализируем менеджер уровней
    Im = new LayerManager();
    // добавляем мяч к уровню
    Im.append(bol);
    // добавляем бомбы к уровню
    Im.append(bomba1);
    Im.append(bomba2);
    Im.append(bomba3);
    Im.append(bomba4);
}
// обрабатываем столкновение
public void stolknovenie()
```

```
// столкновение с первой бомбой
if(bol.collidesWith(bombal, true))
{
    bol.nextFrame();
    bombal.nextFrame();
}
// столкновение со второй бомбой
if(bol.collidesWith(bomba2, true))
{
    bol.prevFrame();
    bomba2.prevFrame();
}
// столкновение с третьей бомбой
if (bol.collidesWith(bomba3, true))
{
    bol.setFrame(2);
    bomba3.setFrame(0);
}
// столкновение с четвертой бомбой
if (bol.collidesWith(bomba4, true))
{
    int[] i = {2,3};
    bol.setFrame(0);
    bomba4.setFrameSequence(i);
}
}

public void start()
{
    z = true;
    // создаем и запускаем поток
    Thread t = new Thread(this);
    t.start();
}
// останавливаем поток
public void stop(){ z = false; }

public void run()
{
    // получаем графический контекст
    Graphics g = getGraphics();
    while (z)
    {
        // столкновение с препятствием
```



```
        stolknovenie();
        // обрабатываем события с клавиш телефона
        inputKey();
        // рисуем графические элементы
        init(g);
        // останавливаем цикл на 20 миллисекунд
        try { Thread.sleep(20); }
        catch (Java.lang.InterruptedException zxz) {}
    }
}

private void inputKey()
{
    // определяем нажатую клавишу
    int keyStates = getKeyStates();
    // код обработки для левой нажатой клавиши
    if ((keyStates & LEFT_PRESSED) != 0) bol.moveLeft();
    // код обработки для правой нажатой клавиши
    if ((keyStates & RIGHT_PRESSED) != 0) bol.moveRight();
    // код обработки для клавиши вверх
    if ((keyStates & UP_PRESSED) != 0) bol.moveUp();
    // код обработки для клавиши вниз
    if ((keyStates & DOWN_PRESSED) != 0) bol.moveDown();
}

private void init(Graphics g)
{
    // желтый цвет фона
    g.setColor(0xffff00);
    // перерисовываем экран
    g.fillRect(0, 0, getWidth(), getHeight());
    // рисуем уровень в точке 0,0
    Im.paint(g, 0, 0);
    // двойная буферизация
    flushGraphics();
}
}
```

файл MySprite.java

класс MySprite

*/

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

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

public class MySprite extends Sprite
{
    // конструктор
    public MySprite(Image image, int fw, int fh)
    {
        // обращаемся к конструктору суперкласса
        super(image, fw, fh);
    }
    // метод для левой нажатой клавиши
    public void moveLeft()
    {
        // передвигаем спрайт
        move(-1,0);
    }
    // метод для правой нажатой клавиши
    public void moveRight()
    {
        // передвигаем спрайт
        move(1,0);
    }
    // метод для клавиши вверх
    public void moveUp()
    {
        // передвигаем спрайт
        move(0,-1);
    }
    // метод для клавиши вниз
    public void moveDown()
    {
        // передвигаем спрайт
        move(0,1);
    }
}
```

В листинге 8.4 содержатся три класса: `MainGame`, `MyGameCanvas` и `MySprite`. Основной код обработки столкновений бомб и мяча находится в классе `MyGameCanvas`, этому классу мы и уделим особое внимание при разборе листинга.

В конструкторе класса `MyGameCanvas` происходят загрузка изображения мяча из файла ресурса `bol.png`, инициализация объекта `bol`, класса `MySprite` и устанавливается позиция прорисовки на экране объекта `bol`.

```
Image bolImage = Image.createImage("/bol.png");
bol = new MySprite(bolImage, 23, 23);
bol.setPosition(getWidth()/2, getHeight()/2);
```

Заметьте, что позиция вывода мяча на экран установлена в центре экрана, но эта позиция определена для левой верхней точки мяча, поэтому именно левый верхний угол изображения спрайта мяча будет находиться в центре экрана. Для того чтобы нарисовать сам спрайт в центре экрана, нужно переопределить опорную позицию мяча с помощью метода `defineReferencePixel()`.

Затем в конструкторе класса `MyGameCanvas` загружается изображение бомбы.

```
Image bombaImage = Image.createImage( "/bomba.png" );
```

После этого происходит инициализация четырех объектов `bomba1`, `bomba2`, `bomba3` и `bomba4` класса `MySprite` и устанавливается позиция для вывода всех четырех бомб на экран телефона.

```
bomba1 = new MySprite(bombaImage, 23, 23);
bomba1.setPosition(10, 10);
bomba2 = new MySprite(bombaImage, 23, 23);
bomba2.setPosition(getWidth()-30, 10);
bomba3 = new MySprite(bombaImage, 23, 23);
bomba3.setPosition(10, getHeight()-40);
bomba4 = new MySprite(bombaImage, 23, 23);
bomba4.setPosition(getWidth()-30, getHeight()-40);
```

Все четыре бомбы рисуются в разных углах экрана слева направо и сверху вниз.

Произведя загрузку необходимых изображений и инициализируя все объекты класса `MySprite`, можно добавить их к уровню с помощью менеджера уровней.

```
Im.append(bol) ;
Im.append(bomba1) ;
Im.append(bomba2) ;
Im.append(bomba3) ;
Im.append(bomba4) ;
```

Наша задача в этом примере - это определение столкновения бомб и мячика, для этого создан метод `stolknovenie()`, где при помощи конструкции `if/else` происходит обработка столкновения объектов `bol` и `bomba1 - bomba4`. Сейчас было бы очень хорошо, если бы вы могли запустить с компакт-диска программу из листинга 8.4, она находится в папке `\Code\Chapter8\Listing8_4\bin\Listing8_4.jad`, и попробовали осуществить столкновение мяча с четырьмя бомбами. Как мы договорились, при столкновении будет обсуждаться работа четырех разных методов.

В столкновении с первой бомбой, находящейся в левом верхнем углу экрана, используется *метод* `nextFrame()`. Если вы переместите мячик по экрану, то при

наезде на первую бомбу произойдет взрыв бомбы и мяча. То есть начнется перебор всех имеющихся фреймов обоих объектов в циклическом порядке. Как только вы уберете мячик с бомбы, взрыв обоих объектов прекратится, потому что закончится перебор фреймов изображений. А состояние бомбы и мяча будет соответствовать одному из фреймов всей анимационной последовательности, при этом возможности повлиять на остановку перехода по фреймам в методе `nextFrame()` нет.

Вторая бомба, находящаяся в правом верхнем углу экрана, при обработке столкновения использует практически идентичный предыдущему *метод* `prevFrame()`, отличающийся лишь тем, что переход по всем существующим фреймам бомбы и мяча происходит в обратном порядке. Остановить работу метода на нужном фрейме также невозможно.

Третья бомба рисуется в нижнем углу экрана, и для обработки столкновения мяча с бомбой используется *метод* `setFrame()`. Этот метод производит переход по заданным фреймам всей анимационной последовательности по номеру или индексу фрейма. В этом примере используется следующий код при столкновении мяча и третьей бомбы.

```
bol.setFrame(2);  
bomba3.setFrame(0);
```

Когда вы передвинете мячик на третью бомбу, то увидите, что изображение мячика изменится и будет соответствовать третьему фрейму всей анимационной последовательности мячика. Состояние бомбы останется неизменным, потому что используется индекс 0, а это первый фрейм бомбы. И последняя бомба при столкновении задействует *метод* `setFrameSequence()`, благодаря которому можно использовать фреймовые последовательности в виде массива индексов.

8.10. Игра «Метеоритный дождь»

Приступая к реализации любой программы, необходимо иметь готовую идею этой самой программы, а также потратить достаточно много времени на разработку программных классов, которые будут использоваться в приложении. Ни в коем случае нельзя приступать к работе над исходным кодом, не зная, сколько и какие классы будут у вас в игре. В противном случае вы потратите очень много времени на разработку игры, потому что все время будете что-то переделывать, улучшать и т. д.

В нашем случае мы всего этого в некоторой степени лишены по одной простой причине. Вы не умеете или не совсем понимаете, как делаются мобильные игры. В связи с этим мы несколько изменим подход в создании демонстрационной игры, а именно, вместо того чтобы сразу продумать методы и способы реализации всей игры, мы будем двигаться шаг за шагом от простого к сложному.

Сначала, и уже в следующей главе, будет создан каркас игровых классов, затем мы добавим в игру меню, далее начнем работать с графикой, создадим игровую карту, нарисуем главного героя игры и метеориты. Заставим все объекты двигаться, летать, сталкиваться, взрываться и т. д. Таким образом, мы шаг за шагом в конечном счете придем к созданию полноценной игры. Но это не значит, что для создаваемой игры отсутствовал этап проектирования программы. Он как раз таки

присутствовал, да еще и в каком объеме, поскольку необходимо было придумать методику создания игры на базе, которую можно было бы понятно и, главное, доступно объяснить и показать создание мобильной игры. Но этот процесс мы несколько упростим и растянем его как бы на все оставшиеся главы книги. По мере изучения нового материала будут изучаться методы реализации тех или иных особенностей программирования мобильных игр на Java 2 ME.

8.10.1. Идея игры

В основу идеи игры легла космическая стрелялка с видом сверху. Главный герой - это биомеханический корабль, являющийся симбиозом уникальных сплавов с живой материей. Обшивка корабля выращена из разумного растения Forozium с далекой планеты Rogitan галактической системы Senock. Такая обшивка позволяет развивать кораблю огромную скорость и повышенное маневрирование. Корабль оснащен новейшими видами вооружения и генетической системой регенерации патронов и снарядов ко всем известным видам оружия в пределах 8000 галактик на основе различного сырья. Химический двигатель корабля дает возможность кораблю обходиться без дозаправки и смены комплектующих на протяжении более ста лет, пока жива биосоставляющая корабля...

Нафантазировать, как вы понимаете, можно многое, вопрос только: как все это реализовать? Но на самом деле в нашей игре пользователь будет сражаться за главного героя, который в свою очередь летит на выполнение своей очередной миссии. В какой-то промежуток времени на его пути встречается большое метеоритное облако, которое в течение нескольких уровней он должен успешно преодолеть. Получается, что эта игра - один из эпизодов прославленного биомеханического корабля, летящего навстречу новым приключениям. Организовав эту вроде бы несложную игровую атмосферу, мы, таким образом, поучимся создавать мобильные игры, а уже на базе полученных знаний вы сможете отправить корабль в более опасное путешествие...

Поскольку в игре рассматривается эпизод с метеоритным облаком, то и название игры «Метеоритный дождь» подходит как нельзя кстати. Но, как вы понимаете, название игры нужно выбирать очень тщательно и обдуманно. Как сказал капитан Врунгель: как корабль назовешь - так он и поплывет. В последнее время очень часто в названиях игр используются названия нашумевших кинофильмов, но это, конечно, удел больших игровых студий, которые могут позволить себе приобрести лицензию на создание такой игры. С другой стороны, ничто вам не мешает назвать свою игру, например, «Сын джедая. Властелин тайной комнаты» или еще что-нибудь в этом духе.

8.10.2. Графика

Графическая часть любой игры является одной из важнейших составляющих. От того, как нарисована графика, будет зависеть общее восприятие всей игры в целом. Чем хуже графика в игре, тем хуже эта игра будет продаваться. Времена простых закрашенных квадратиков уже давно в прошлом. Подходить к созданию игровой

графики нужно профессионально, и лучше, если этим будет заниматься человек, действительно владеющий своим предметом отлично.

В книге игра создавалась в виде демонстрационного примера, поэтому особых требований к качеству игровой графики не предъявлялось. Были созданы ряд экранов с заставками, главный герой, метеорит, пули, взрывы и другие графические элементы. Кстати, очень важен конечный размер графического файла. В некоторых более дешевых телефонах существует лимит на общий объем устанавливаемого в устройство приложения. И если ваша игра будет больше этого максимального размера, то программа не сможет установиться на телефон. В качестве основного телефона при разработке игры был выбран стандартный эмулятор инструментария Wireless Toolkit, который имеет разрешение 240 x 320 пикселей. Сейчас большая часть телефонов на рынке имеет именно такое разрешение экрана.

8.10.3. Исходные коды

Исходные коды игры вы найдете на компакт-диске в папке **\Code**. В каждой последующей главе (начиная с главы 10) мы будем модернизировать исходный код игры «Метеоритный дождь». В главе 9 будет создан игровой каркас классов, который в дальнейшем с каждой новой главой будет изменяться. В связи с этим к названию игры для каждой новой главы я буду добавлять дополнительную цифру, совпадающую с числовым обозначением текущей главы. Например, для следующей главы название игры будет таким: **Meteoric_rain_9**. Дополнительно каждая глава на компакт-диске имеет свое обозначение, например **\Code\Chapter9\Meteoric_rain_9**.

<http://palata-x.narod.ru>



<http://palata-x.narod.ru>

Глава 9. Формируем каркас игры

В большинстве компаний, специализирующихся на создании мобильных программ, для разработки игр применяются определенные шаблоны классов. Каждый такой шаблон содержит набор классов и определяет общую структуру всей игры. Для любого типа игр можно создать свой шаблон необходимых классов и впоследствии при реализации того или иного проекта успешно использовать подходящий шаблон. Такая универсализация процесса создания игр значительно экономит время, силы и деньги, затрачиваемые на проект.

Лично я в своем активе имею восемь различных шаблонов, которые создал на основе многолетнего опыта работы в мобильной индустрии. На базе одного из шаблонов в течение одного месяца можно сделать практически любую мобильную игру! В основном сложность формирования очередной игры заключается в создании новой графики и адаптации имеющегося набора игровой логики под конкретную игру. На рисование графики уходит много времени, и это составляет порядка 70-90% времени работы над всей игрой. Адаптация, или написание новой игровой логики для игры, - также весьма трудная задача.

Игры в отличие от простых программ имеют очень сложное строение, поэтому, создавая структуру классов мобильной игры, необходимо обязательно учитывать множество различных игровых состояний, таких как инициализация игровых объектов, вход в игровой цикл, пауза в процессе игры, обработка пользовательского ввода и игровой логики, выход из игры и многое другое.

В этой главе мы сформируем структуру классов игры «Метеоритный дождь» на основе шаблона (правда, в несколько упрощенном варианте), который я сделал достаточно давно и успешно применяю при создании мобильных игр. Этот шаблон классов был создан специально для типа игр, где все события происходят с видом сверху и основаны на движении объектов параллельно или навстречу друг другу. В этот тип игр входят всевозможные космические, исторические или современные военные летные сражения, гонки на машинах, мотоциклах или других транспортных средствах с видом сверху. Но прежде чем перейти к анализу классов игры «Метеоритный дождь», давайте познакомимся с общим механизмом работы всех мобильных игр.

9.1. Механизм работы мобильных игр

Любая игра имеет множество различных состояний, или рабочих фаз, посмотрите на рис. 9.1, где графически представлена общая схема работы игры. Как видно из этого рисунка, мобильные игры могут иметь несколько рабочих фаз, о которых мы сейчас поговорим более подробно.

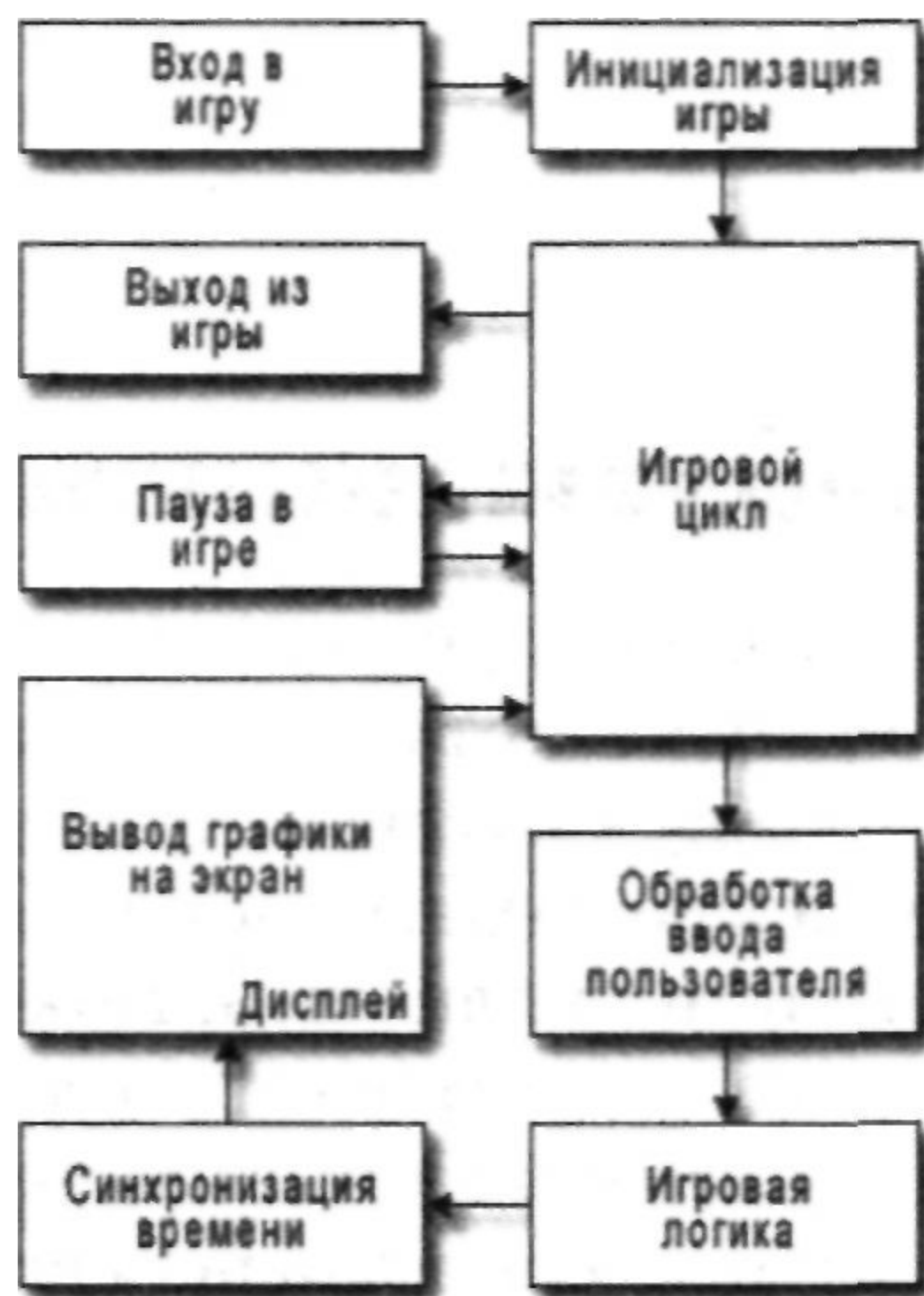


РИС. 9.1. Схема работы мобильных игр

9.1.1. Вход в игру

Вход в игру означает фактический запуск приложения на мобильном телефоне. После установки игры на устройство на рабочем столе телефона появляется графическая иконка. Выбрав джойстиком иконку игры, вы тем самым запускаете работу установленного приложения, то есть осуществляете вход в игру. После запуска игры на телефоне программа переходит к этапу инициализации.

9.1.2. Инициализация игры

Этап инициализации игры включает в себя множество различных действий, направленных на загрузку и инициализацию всей программы в целом. На этом этапе происходят выделение, или распределение, системной памяти телефона, инициализация, создание и загрузка игровых объектов, изображений, звуковых файлов, ресурсов и других игровых компонентов. И только после того как все компоненты созданы и загружены в память телефона, управление над работой программы передается в игровой цикл.

9.1.3. Игровой цикл

Все мобильные и компьютерные игры основаны на непрерывном цикле, в котором происходит постоянное обновление игрового процесса. Этот непрерывный цикл называется *игровым циклом*. Известный термин FPS, или частота смены кадров в одну секунду, характеризует игровой цикл как цикл, обеспечивающий непрерывную смену кадров в игре с заданной периодичностью. Как правило, период обновления игры может составлять порядка 30 раз в одну секунду (FPS 30).

Это время обновления игрового процесса позволяет добиться плавной анимации и сопоставимо по качеству с просмотром кинофильма.

В игровом цикле происходит постоянное обновление игры, включая обработку пользовательского ввода, игровой логики и вывода графики на экран телефона. Игровой цикл - это ключевой компонент любой игры! Как вы построите игровой цикл, так и будет работать ваша игра.

9.1.4. Обработка ввода пользователя

Обработка пользовательского ввода - это получение событий или команд от пользователя с телефонной клавиатуры или джойстика. Все полученные команды от пользователя в игре обрабатываются в соответствии с набором игровой логики. На некоторых моделях телефонов игрой удобнее управлять джойстиком, а на других моделях лучше всего использовать клавиши.

Примечание. К сожалению, производители телефонов меньше всего думают о том, как сделать удобный телефон для игр., за исключением, пожалуй, компании Nokia. На мой взгляд, самая лучшая система управления игрой выполнена в смартфонах Nokia N-Gage и Nokia N-Gage QD. Очень удобно держать смартфон N-Gage горизонтально, управляя джойстиком левой рукой, а правой рукой с помощью клавиатуры осуществлять различные дополнительные действия.

В игре управление джойстиком происходит традиционным способом. Доступны команды **Вверх**, **Вниз**, **Влево**, **Вправо** и **Огонь** (нажатие на центр джойстика). На клавиатуре телефона для всех перечисленных команд также зарезервированы клавиши со следующими номерами:

- **Вверх** - клавиша под номером 2;
- **Вниз** - клавиша под номером 8;
- **Влево** - клавиша под номером 4;
- **Вправо** - клавиша под номером 9;
- **Огонь** - клавиша под номером 5.

Дополнительно в игре можно использовать и другие клавиши телефона для различных игровых опций, например для выбора оружия, смены героя, просмотра карты и т. д.

9.1.5. Игровая логика

Игровая логика - это искусственный интеллект, который программист создает специально для реализации логической составляющей игры. На основе полученных команд от пользователя и игровой логики происходят обновление состояния игры и вывод графики на экран телефона. Подход в создании искусственного интеллекта может быть различным. В этой книге вы познакомитесь с основными приемами создания игрового интеллекта.

9.1.6. Синхронизация времени

Игровой цикл, а вместе с ним и игровая графика должны обновляться с определенной периодичностью. В мобильных играх обновление игры, или количество смены кадров в одну секунду, может достигать порядка 15-30 кадров. Промежуток обновления игровой сцены, или время, необходимое на перерисовку графики, определяется с помощью этапа синхронизации времени.

9.1.7. Вывод графики на экран

Вывод графики на экран - это построение игровой сцены, или рисование графики на экране телефона. Построение игровой сцены происходит на системном уровне, и вам как программисту достаточно для этого написать всего пару строк исходного кода, о чем мы поговорим позже в этой главе.

9.1.8. Пауза в игре

Во время игры иногда нужно сделать секундный перерыв или просто отвлечься на небольшой промежуток времени. Для того чтобы не терять все свои завоеванные трудом игровые позиции, в играх существует режим паузы. Паузу в играх обычно привязывают к одной из телефонных клавиш или назначают на одну из клавиш выбора (soft key, или подэкранные клавиши телефона). Назначая команду паузы на клавишу выбора, программист создает информационный диалог, который отражается на экране телефона. В этом диалоге, как правило, присутствует пара вопросов с предложением продолжить игру, воспользоваться небольшой паузой или выйти из игры.

9.1.9. ВЫХОДИЗ ИГРЫ

Выход из программы - это автоматическое закрытие работающей программы с освобождением выделенной системной памяти, или, как принято говорить, освобождением захваченных ресурсов, и удалением всех созданных игровых объектов. Для этих целей в Java 2 ME имеется ряд методов, помогающих программисту создать корректное завершение работы программы. Выход из программы обычно происходит из главного меню игры, но можно создать диалог, по которому игрок мог бы выходить непосредственно из игрового процесса, как это было описано в предыдущем разделе.

9.2. Как работают шаблоны

Шаблоны, которые имеются в моем активе, содержат набор различных классов. Механизм работы игр и часть классов во всех шаблонах остаются неизменными, формируя тем самым некий игровой каркас классов, на основе которого и создаются игры. Посмотрите на рис. 9.2, где представлен общий механизм работы всех шаблонов.

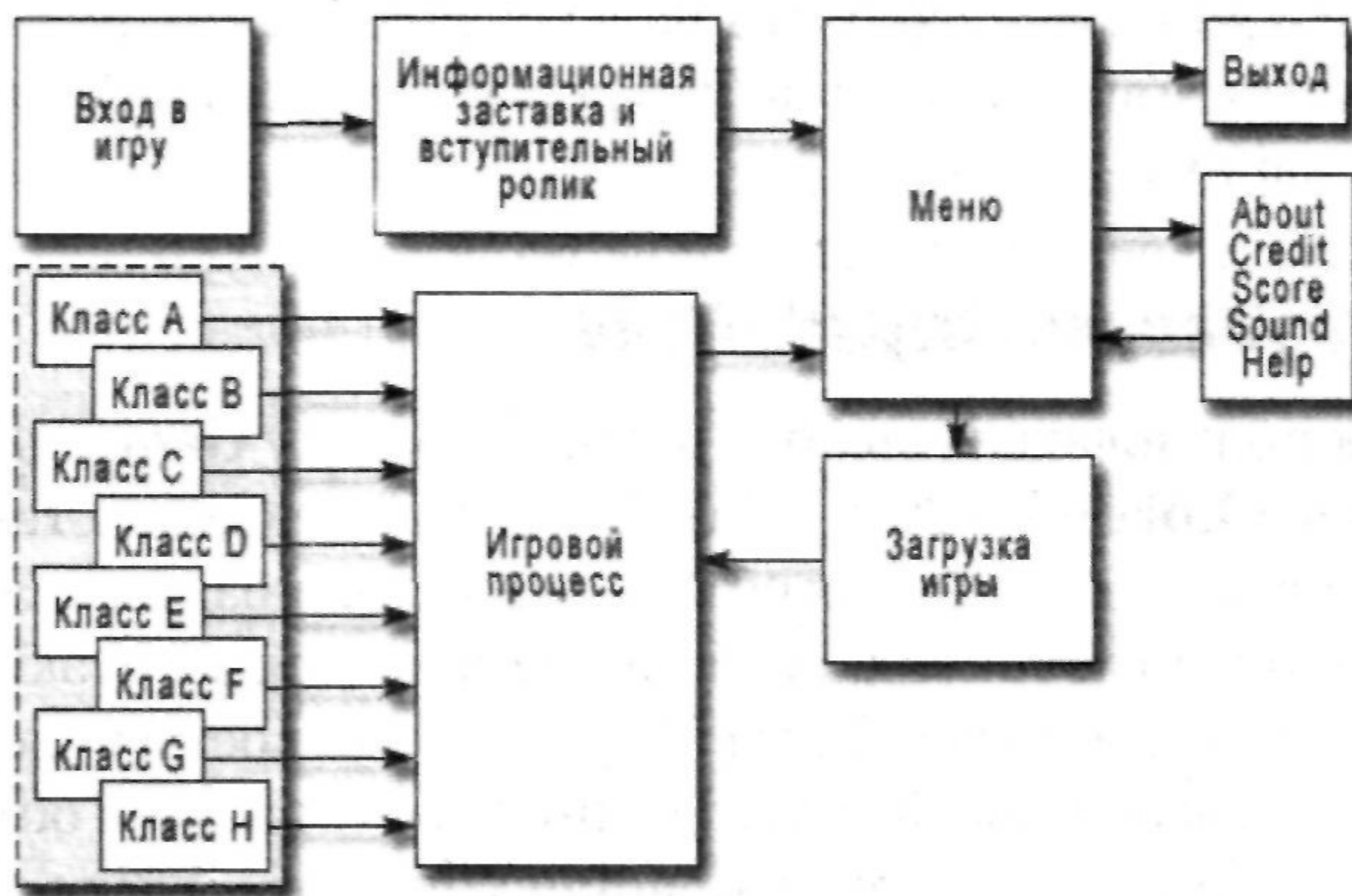


Рис. 9.2. Механизм работы шаблонов

9.2.1. Запуск игры и информационная заставка

При запуске игры на телефоне первым делом появляется информационная заставка с названием компании или любой другой сопутствующей информацией. В момент показа заставки на экране телефона происходят выделение памяти, создание игровых классов и загрузка их в память телефона. Заставка будет показана на экране телефона до тех пор, пока все компоненты игры не будут созданы и загружены в память. После этого игра автоматически переходит к вступлению.

9.2.2. Вступительный ролик

Вступление перед игрой обычно рассказывает об основных событиях, происходящих в игре, или о предыстории будущих событий. Вступление может быть создано в виде набора сменяющих друг друга изображений, либо можно создать полноценный видеоролик. Если пользователь не хочет смотреть вступительный ролик, он вправе пропустить его показ, нажав для этого соответствующую клавишу телефона. По окончании просмотра вступительного ролика пользователь попадает в основное меню игры.

9.2.3. Меню

Меню игры - это своеобразный пульт управления с набором таких команд, как **Game, Sound, Help, About, Credit, Exit** в английской версии или **Играть, Звук, Помощь, Об игре, Об авторе, Выход** в русской версии игры. С помощью перечисленных команд пользователь может начать игру, включить или отключить звук в игре, просмотреть помощь и информацию о создателе и издателе игры, а также выйти из программы.

Меню игры обычно делают в виде красивого рисунка с элементами анимации, поскольку меню - это своеобразная визитная карточка всей игры. Также можно сделать меню, применяя для этого стандартные классы API Java 2 ME. В этом

случае меню будет одинаково работать на всех телефонах, но, правда, при этом игра потеряет свою индивидуальность. В игре «Метеоритный дождь» мы сделаем меню, применяя графику.

9.2.4. Загрузка и запуск игры

После того как пользователь выбрал в меню команду **Game**, на экране телефона рисуется заставка **Loading** (Загрузка). В момент показа заставки происходят загрузка уровня и установка игровых объектов на свои позиции. Заставка рисуется на экране в течение двух секунд, но если времени для загрузки уровня и установки объектов на позиции понадобится больше, то это время будет выделено системой в автоматическом режиме. Как только процесс установки объектов закончен и программа готова к работе, заставка **Loading** убирается с экрана и запускается игровой процесс.

Игровой процесс содержит в себе игровой цикл, обработку пользовательского ввода, синхронизацию времени обновления игры и вывод графики на экран телефона. Дополнительно имеется ряд классов, с помощью которых создаются различные игровые объекты, взрывы, выстрелы, спецэффекты и многое другое. Набор и количество дополнительных классов определяются в зависимости от конкретной создаваемой игры. Думается, что основной механизм работы игр вам понятен, поэтому давайте перейдем к разбору общей структуры классов нашей будущей игры.

9.3. Структура классов игры «Метеоритный дождь»

Шаблон классов определяет как общую структуру классов игры, так и тип самой игры. Например, для гонок на транспортных средствах с видом сверху может использоваться один набор классов, тогда как для логических игр - совсем другой. В нашей с вами игре я использовал один из своих шаблонов. На этом этапе мы только сориентируемся, для каких целей служит тот или иной класс, а уже в следующих разделах рассмотрим составляющую каждого класса по отдельности, но без подробного разбора исходного кода.

Итак, какие классы нам понадобятся для создания игры.

- `GameMidlet` - основной класс мидлета является производным от библиотечного класса `MIDlet`. Это, как вы знаете, стартовая площадка для любой программы на Java 2 ME;
- `Splash` - наша первая заставка, которая появляется на экране телефона при запуске игры. В момент показа заставки происходят выделение памяти и создание всех игровых объектов;
- `Menu` - графическое меню игры с набором команд, обеспечивающих начало игры, информацию об игре, помощь, выход из игры и т. д.;
- `Loading` - загрузочная заставка игры, в момент ее показа происходят загрузка текущего уровня и установка игровых объектов по своим позициям;

- Background - игровой фон, или игровая карта, созданная специально для каждого уровня игры;
- Ship - это наш главный герой, космический корабль-робот, бороздящий пространства галактик и устанавливающий справедливость во всем мире;
- Meteorite - метеориты, летящие навстречу кораблю, столкновения с ними чреваты потерей жизненной энергии корабля;
- Shot - этот класс представляет выстрелы в игре для главного персонажа сказки;
- Explosion - игровые взрывы, возникающие при столкновении пули корабля с метеоритами и метеоритов с кораблем;
- MainGameCanvas - в этом классе сосредоточен основной игровой процесс. Здесь создаются игровые объекты, реализован игровой цикл, происходят обработка пользовательского ввода, обновление состояния игры, вывод графики на экран телефона, включение режима паузы, выход в меню, окончание игры и т. д.

Как видите, все перечисленные классы характеризуют тот механизм работы и состояния игры, который мы рассматривали ранее в этой главе. Теперь давайте перейдем к подробному анализу классов игры «Метеоритный дождь», но без рассмотрения исходного кода. На этой стадии нам необходимо лишь продумать общую структуру имеющихся у нас классов, а уже на этапе работы над исходным кодом классов мы будем думать, как нам лучше реализовать тот или иной класс.

9.4. Как устроен каркас игровых классов?

В предыдущих разделах мы обстоятельно проанализировали механизм работы мобильных игр и с багажом полученных знаний вполне можем создать свой каркас игры. В коммерческих проектах, как вы уже знаете, создание игры почти всегда строится на основе шаблонов. В нашем случае мы не имеем такого шаблона, и поэтому нам предстоит создавать шаблон классов самим от начала и до конца. Конечно, в сущности мы будем рассматривать один из шаблонов, созданный мною ранее, но подход в работе над исходным кодом будет строиться так, как будто мы начинаем работу над проектом с нуля.

Прежде всего необходимо подумать о механизме работы и о классах, которые будут составлять костяк игры. Что касается механизма работы каркаса классов игры, то схема функционирования здесь следующая. Первоначально происходит загрузка первой игровой заставки, под прикрытием которой создаются и инициализируются игровые объекты, далее идет загрузочная заставка с загрузкой текущего уровня и выставлением объектов по позициям, после чего в работу включается игровой цикл.

Что касается игровых классов, которые понадобятся на этом этапе, то нам обязательно нужно создать класс `GameMidlet`, производный от библиотечного класса `MIDlet`, а значит, наследующий все его возможности. Без этого класса, как вы понимаете, не может существовать ни одна телефонная программа, и именно с этого класса начинается работа любого приложения. Еще нам понадобятся классы `Splash`, `Loading` и `MainGameCanvas`.

Класс `Splash` - это первая игровая заставка, и во время ее показа на экране телефона происходят создание и инициализация всех игровых объектов. После этого управление программой переходит к классу `Loading`. На экране телефона рисуется загрузочная заставка `Loading`, за кулисами которой совершаются загрузка текущего уровня и установка игровых объектов на свои позиции. Далее в работу включается класс `MainGameCanvas`, представляющий игровой цикл.

Итак, для формирования каркаса игры нам необходимо создать четыре новых класса:

- `GameMidlet`;
- `Splash`;
- `Loading`;
- `MainGameCanvas`.

9.5. Класс GameMidlet

Класс `GameMidlet` - это основной класс мидлета, наследующий возможности класса `MIDlet` из пакета `javax.microedition.midlet`. Через класс `GameMidlet` будет происходить управление всей игрой. Для этих целей в классе `GameMidlet` создается ряд методов и объектов других классов. Посмотрите на листинг 9.1 с исходным кодом этого класса, а также на рис. 9.3, где представлена UML-диаграмма класса `GameMidlet`.

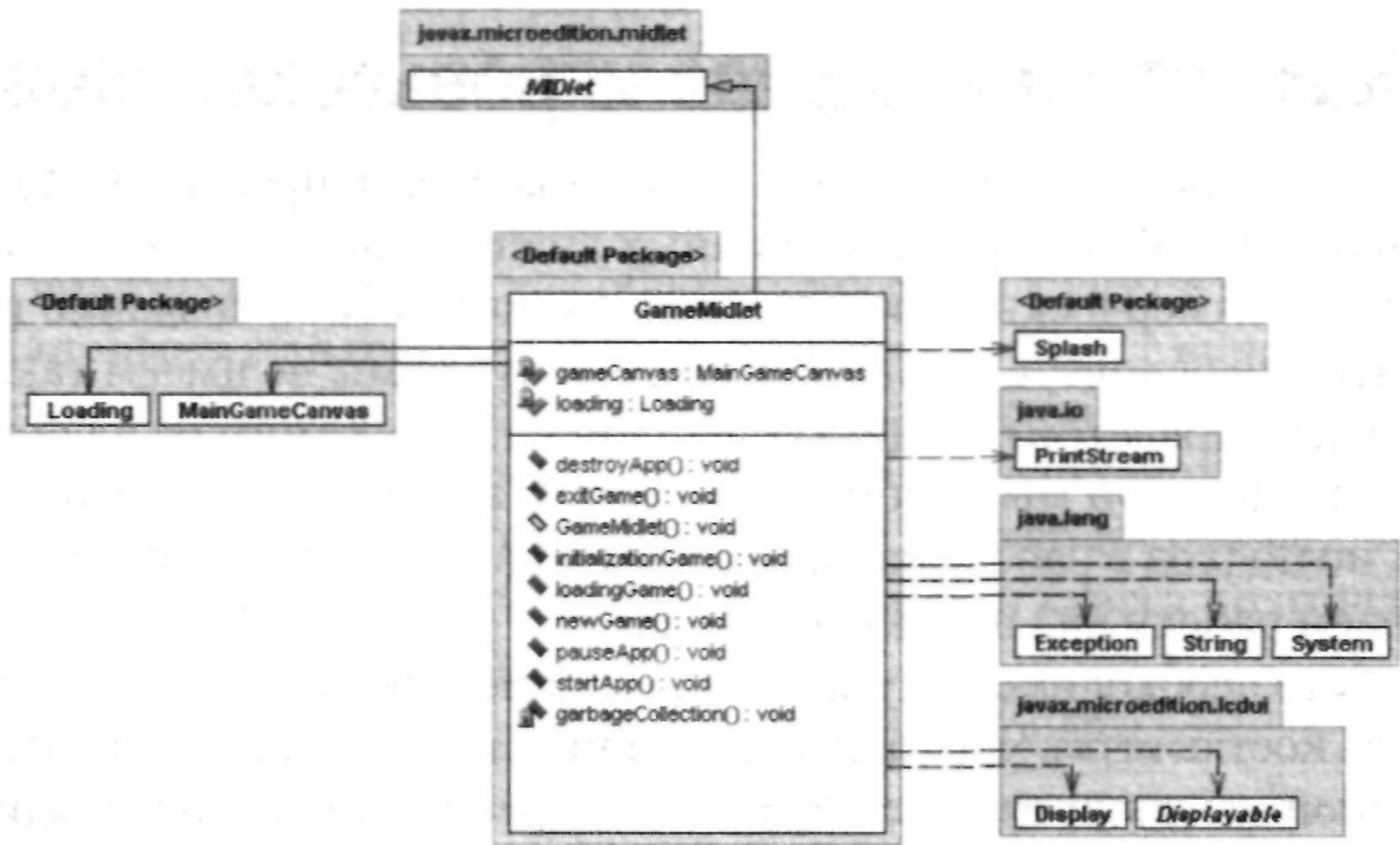


Рис. 9.3. UML-диаграмма класса `GameMidlet`

```
/*
 * GameMidlet.java
 * Основной класс мидлета
 * ЛИСТИНГ 9.1
 */

// библиотеки импорта
```

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

/**
 * @author Stanislav Gornakov
 * Aversion 1.0
 */

public class GameMidlet extends MIDlet{
    // объект класса MainGameCanvas
    private MainGameCanvas gameCanvas = null;
    // объект класса Loading
    private Loading loading = null;

    // конструктор
    public GameMidlet(){
    }

    // стартовая, или входная, точка в работе всей программы
    public void startApp(){
        try{
            // создаем и запускаем класс Splash
            Display.getDisplay(this).setCurrent(new Splash(this));
        }catch (Exception ex) {
            System.err.println("Class GameMidlet method startApp()");
        }
    }

    public void pauseApp(){
    }

    // освобождение всех выделенных системой ресурсов
    public void destroyApp(boolean unconditional){
        garbageCollection();
    }

    // инициализация игры
    public synchronized void initializationGame(){
        try{
            // создаем объект класса MainGameCanvas
            gameCanvas = new MainGameCanvas(this);
            // создаем объект класса Loading
            loading = new Loading(this);
            // запускаем работу метода
        }
    }
}
```

```
        loadingGame();
    }catch (Exception ex) {
        System.err.println("Method initializationGame()");
    }
}

// загрузка заставки Loading
public void loadingGame() {
    try{
        // запускаем системный поток и работу класса Loading
        loading.start();
        // показываем экран с работой класса Loading
        Display.getDisplay(this).setCurrent(loading);
    }catch (Exception ex) {
        System.err.println("Class GameMidlet method LoadingGame()");
    }
}

// запускаем работу игры
public void newGame() {
    try{
        // загружаем текущий уровень и ставим объекты по позициям
        gameCanvas.setGame();
        // запускаем работу игрового цикла
        gameCanvas.start();
        // показываем экран с работой класса MainGameCanvas
        Display.getDisplay(this).setCurrent(gameCanvas);
        // останавливаем работу класса Loading
        loading.stop();
    }catch (Exception ex) {
        System.err.println("Class GameMidlet method newGame()");
    }
}

// обнуляем все объекты
private void garbageCollection() {
    loading = null;
    gameCanvas = null;
    System.gc();
}

// выход из игры -
public void exitGame() {
    destroyApp(true);
}
```



```
        notifyDestroyed();  
    }  
}
```

В начале исходного кода `GameMidlet.java` происходит глобальное объявление двух объектов `gameCanvas` и `loading`. Первый объект `gameCanvas` является объектом класса `MainGameCanvas`. С помощью этого объекта мы будем запускать игровой цикл. Объект `loading` представляет загрузочную заставку для текущего уровня игры. При объявлении объектов в исходном коде всегда лучше явно присваивать им значения `null` во избежание всевозможных проблем.

После запуска приложения на телефоне первым в работу вступает класс `GameMidlet`. В теле этого класса следуют создание глобальных переменных, выполнение исходного кода конструктора класса и запуск метода `startApp()`. Именно с поиска метода `startApp()` в исходном коде класса `GameMidlet` начинается работа всей программы. В методе `startApp()` создается объект класса `Splash` с его моментальным выполнением и как следствие - показ на экране телефона результатов работы класса методом `setCurrent()`.

```
try{  
    Display.getDisplay(this).setCurrent(new Splash(this));  
}catch(Exception ex){  
    System.err.println("Class GameMidlet method startApp()");  
}
```

Обратите внимание, что при создании класса используется конструкция кода `try/catch`, необходимая для обработки исключительных ситуаций, которые могут возникнуть во время работы программы. Если в коде есть ошибки, то на экране в среде программирования, с которой вы работаете, появится соответствующая информация. Чтобы выводимая средой программирования информация об ошибках была понятна, лучше всего использовать надписи, характеризующие этот вид ошибки, например:

```
System.err.println("Class GameMidlet method startApp()");
```

или, допустим, при загрузке изображения, карты игры, фоновой картинке:

```
System.err.println("Изображение Ship.png не найдено");
```

В этом случае вам сразу будет понятна проблема, возникающая в исходном коде игры. Обработку исключительных ситуаций необходимо производить для загружаемых изображений, звуковых эффектов, при работе с игровым циклом и т. д. Почти все среды программирования обладают встроенными функциями контроля над обработкой исключительных ситуаций, и если вы забыли это сделать явно, то вам немедленно будет выведено соответствующее сообщение. Как правило, в таких случаях компиляция проекта блокируется до тех пор, пока в исходном коде не будет прописана обработка исключительной ситуации.

С содействием класса `Splash` на экран телефона выводится первая игровая заставка и совершается запуск метода `initializationGame()`. В методе

`initializationGame()` происходит создание двух объектов `gameCanvas` и `loading`, а также запуск метода `loadingGame()`.

```
gameCanvas = new MainGameCanvas(this);  
loading = new Loading(this);  
loadingGame();
```

При создании двух объектов `gameCanvas` и `loading` в качестве параметра передается объект `midlet`. Это необходимо для того, чтобы в этих классах имелась возможность доступа к методам класса `GameMidlet`, с помощью которых можно запустить игру, выйти из игры, включить паузу в игре и т. д. После создания двух объектов в работу включается метод `loadingGame()`, тело которого выглядит следующим образом:

```
loading.start();  
Display.getDisplay(this).setCurrent(loading);
```

Здесь происходит автоматическая смена экранов или смена графических заставок, рисуемых на экране, поскольку совершается переход уже к работе класса `Loading` и показу его на дисплее. В свою очередь, класс `Splash` заканчивает работу. С помощью деструктора этого класса и сборщика мусора Java 2 ME он удаляется из памяти, поскольку нужен нам всего лишь один раз во время первой загрузки игры.

Далее на экране телефона появляется заставка с надписью **Loading**, под прикрытием которой происходят загрузка текущего уровня и установка объектов на свои позиции вызовом метода `newGame()`.

```
public void newGame() {  
    try{  
        // загружаем текущий уровень и ставим объекты по позициям  
        gameCanvas.setGame();  
        // запускаем работу игрового цикла  
        gameCanvas.start();  
        // показываем экран с работой класса MainGameCanvas  
        Display.getDisplay(this).setCurrent(gameCanvas);  
        // останавливаем работу класса Loading  
        loading.stop();  
    }catch (Exception ex) {  
        System.err.println("Class GameMidlet method newGame()");  
    }  
}
```

Как только текущий уровень загружен, а все объекты установлены на позиции, совершаются запуск игрового цикла, представленного работой класса `MainGameCanvas`, и остановка работы класса `Loading` посредством вызова метода `loading.stop()`.

Для выхода из игры в классе `GameMidlet` предусмотрен метод `exitGame()`.

```
public void exitGame() {  
    destroyApp(true);
```

```
notifyDestroyed();
}
```

```
/*
 * Splash.java
 * Первая игровая заставка
 * ЛИСТИНГ 9.2
 */

// библиотеки импорта
import javax.microedition.lcdui.*;
import Java.io.*;

/**
 *
 * @author Stanislav Gornakov
 * Aversion 1.0
 */

class Splash extends Canvas implements Runnable{
// мидлет
private GameMidlet midlet = null;
// изображение
private Image imageSplash = null;
// конструктор класса
Splash(GameMidlet midlet){

this.midlet = midlet;
try {
// загружаем изображение
imageSplash = Image.createImage("/res/Splash.png");
}catch (Exception ex) {
System.err.println("Splash.png it is not loaded");
}
// включаем полноэкранный режим
setFullScreenMode(true);
// создаем новый поток
new Thread(this).start();

//запускаем цикл
public void run(){
synchronized(this){
try{
// останавливаем работу цикла на 2 секунды
wait(2000L);
}catch (InterruptedException e) {
```



```
}
try {
    // создаем игровые объекты
    midlet.initializationGame();
} catch (Exception ex) {
    System.err.println("Class Splash method run()");
}

// рисуем на экране телефона
public void paint(Graphics graphics){
    // определяем прямоугольник, закрашиваемый цветом
    graphics.fillRect(0, 0, getWidth(), getHeight());
    // цвет фона белый
    graphics.setColor(250, 250, 250);
    // выводим на экран загруженное изображение
    graphics.drawImage(imageSplash, getWidth() / 2,
        getHeight() / 2,
        Graphics.VCENTER | Graphics.HCENTER);
}
}
```

В области глобальных переменных исходного кода `Splash.java` объявляются два объекта: объект `midlet` класса `GameMidlet`, необходимый для доступа к методам этого класса, и объект `imageSplash` класса `Image`. Объект `imageSplash` представляет изображение заставки, которая загружается из ресурсов программы и рисуется на экране телефона.

В конструкторе класса `Splash` происходит инициализация объекта `midlet` значением `midlet`, которое передается в класс `Splash` при создании объекта в качестве параметра. То есть фактически совершается инициализация объекта `midlet` класса `Splash`, или присвоение этому объекту значения объекта `midlet` класса `GameMidlet`.

Объект `imageSplash` в конструкторе класса инициализируется изображением загрузочной заставки. С помощью метода `createImage()` из ресурсов программы загружается изображение `Splash.png`. В нашем случае при загрузке изображения `Splash.png` его поиск происходит в папке `/res` того же рабочего каталога проекта. Если вы создаете для хранения изображений специальные папки, то в этом случае при загрузке изображений указывается полный путь к ресурсу, например:

```
imageSplash = Image.createImage("/res/icon/Splash.png");
```

В этой строке исходного кода загрузка изображения `Splash.png` происходит из папки `\res\icon` корневого каталога рабочего проекта.

Далее в конструкторе класса `Splash` вызовом метода `setFullScreenMode()` включается полноэкранный режим отображения информации на экране телефона.

Метод `setFullscreenMode()` - это библиотечный метод, вызов которого включает или выключает в приложении полноэкранный режим. Если вызвать этот метод с параметром `true`, то задействуется полноэкранный режим отображения информации на экране, если `false` - то включается оконный режим. В оконном режиме на экране телефона будут рисоваться две панели сверху и снизу, содержащие различные элементы управления программой. Как правило, в мобильных и компьютерных играх задействуется полноэкранный режим, что значительно улучшает общее восприятие игры. Метод `setFullscreenMode()` доступен только в профиле MIDP 2.0, в первой редакции профиля этого метода не было.

Для показа заставки на экране телефона, а также одновременного вызова метода `initializationGame()` нам необходимо создать цикл, которому, в свою очередь, для работы понадобится системный поток. В связи с этим в конструкторе класса `Splash` мы создаем и сразу же запускаем системный поток.

```
new Thread(this).start();
```

По запуску системного потока соответственно запускается работа цикла, представленная методом `run()`, а также происходит выполнение метода `paint()`.

Метод `run()` принадлежит интерфейсу `Runnable` и представляет собой постоянно работающий мотор, на основе которого можно реализовать непрерывный цикл для работы с анимацией. Метод `paint()` доступен от класса `Canvas` и отвечает за прорисовку графики на экране телефона.

В методе `run()` мы останавливаем работу цикла методом `wait()` на две секунды для показа заставки и далее вызываем метод `initializationGame()` для инициализации игровых объектов. Вызов метода `wait()` с параметром `2000L` означает задержку или остановку работы цикла на 2000 миллисекунд, что в эквиваленте равно двум секундам. По прошествии двух секунд паузы происходит вызов метода `initializationGame()`, но вы спросите, а зачем нужно было останавливать работу цикла методом `wait()`, когда можно было бы обойтись прямым вызовом метода `initializationGame()`?

Если обходиться без метода `wait()`, то графическая заставка не будет выведена на экран телефона, а точнее система не успеет эту заставку нарисовать, поскольку последует немедленный вызов метода `initializationGame()`. В этом методе, как вы знаете, происходит инициализация игровых объектов, а в частности создание объектов `gameCanvas` и `loading`.

На данном этапе у нас в программе мало исходного кода, особенно в классе `gameCanvas`, и создание этого объекта и переход к его выполнению произойдут мгновенно, то есть мы не увидим графическую заставку на экране. И даже когда игра будет создана полностью и на инициализацию всех игровых объектов понадобится несколько секунд, то на экране телефона заставка будет рисоваться с большой задержкой, а может, не выводиться и вовсе (здесь все зависит от марки телефона). Поэтому вызов метода `wait()` необходим для своевременного вывода заставки на дисплей.

С помощью метода `paint()` происходит отображение на экране имеющейся в программе графики. Первоначально в этом методе мы определяем прямоугольник,

который будем закрашивать определенным цветом, а потом устанавливаем цвет фона и рисуем изображение на экране.

```
public void paint(Graphics graphics){  
    // определяем прямоугольник, закрашиваемый цветом  
    graphics.fillRect(0, 0, getWidth(),getHeight());  
    // цвет фона белый  
    graphics.setColor(250, 250, 250);  
    // выводим на экран загруженное изображение  
    graphics.drawImage(imageSplash,    getWidth()/2,  
        getHeight()/2,  
        Graphics.VCENTER | Graphics.HCENTER);  
}
```

Изображение заставки рисуется точно в центре экрана, что позволяет впоследствии без проблем адаптировать или портировать исходный код на телефоны с любым разрешением экрана. Например, если сделать заставку по своему размеру 128 x 128 пикселей, то на телефоне с любым размером экрана заставка всегда будет находиться точно в центре, а фон можно закрашивать любым цветом, хорошо сочетающимся с вашей заставкой. Тем самым вы достигнете гибкой реализации исходного кода, но при желании можно рисовать заставку на весь экран и адаптировать ее для каждой модели телефона, здесь все зависит от вашего желания и фантазии. В конце работы метода `initializationGame()` происходит вызов метода `loadingGame()`, что приводит к запуску работы класса `Loading`.

9.7. Класс Loading

Класс `Loading` необходим нам для сокрытия загрузки текущего уровня игры и установки всех игровых объектов на свои позиции. Класс `Loading`, так же как и класс `Splash`, наследует возможности класса `Canvas` и реализует интерфейс `Runnable`. Давайте посмотрим на исходный код этого класса, содержащийся в листинге 9.3, а также проанализируем UML-диаграмму этого класса, изображенную на рис. 9.5.

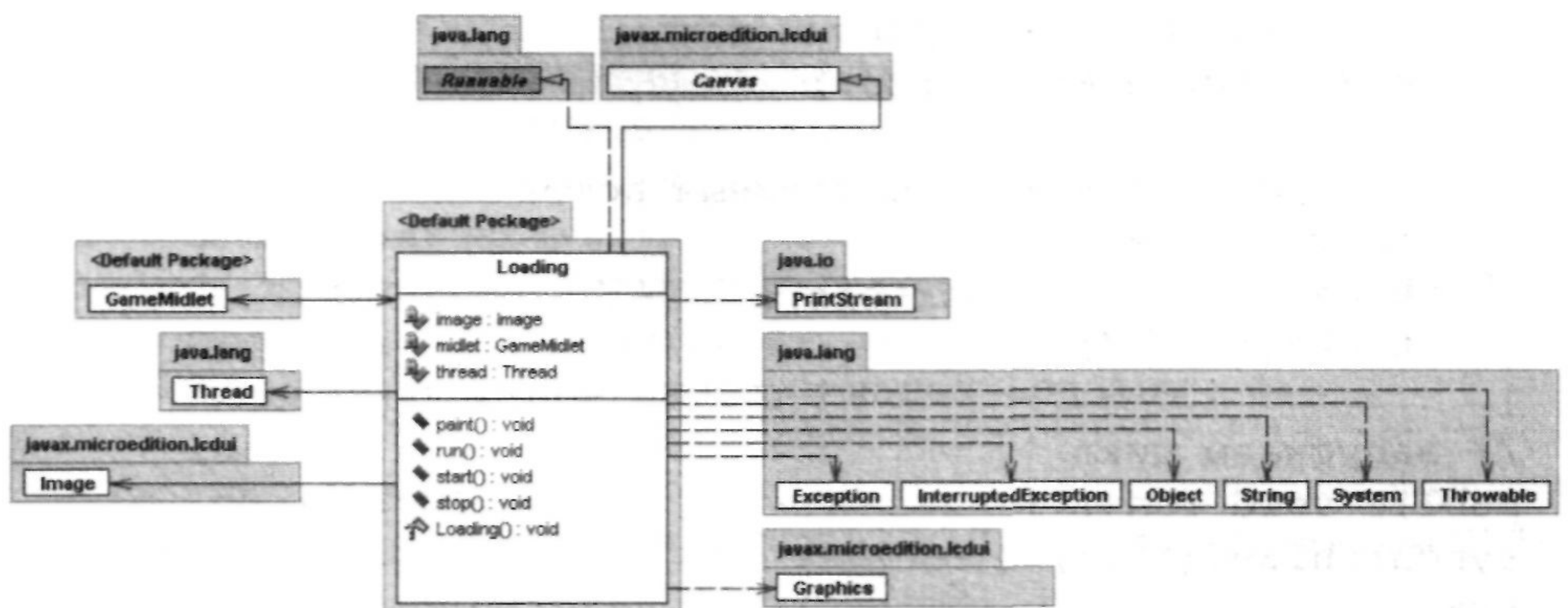


Рис. 9.5. UML-диаграмма класса Loading

```
/*
 * Loading.java
 * Загрузочная заставка
 * ЛИСТИНГ 9.3
 */

//
import javax.microedition.lcdui.*;

/**
 * @author Stanislav Gornakov
 * (Aversion 1.0
 */

class Loading extends Canvas implements Runnable {
// мидлет
private GameMidlet midlet = null;
// объявляем объект для создания системного потока
private volatile Thread thread = null;
// изображение
private Image image = null;

Loading(GameMidlet midlet){
this.midlet = midlet;
try {
// загружаем заставку
image = Image.createImage("/Loading.png");
}catch (Exception ex) {
System.err.println("Loading.png it is not loaded");
}
// включаем полноэкранный режим
setFullScreenMode(true);
}
// создаем и запускаем системный поток
public void start () {
thread = new Thread(this);
thread.start();

// запускаем цикл
public void run() {
synchronized(this) {
try {
wait(2000L);
```



```
}catch (InterruptedException e) {  
}  
try {  
    midlet.newGame();  
}catch (Exception ex) {  
System.out.println(ex);  
}  
}  
}  
// останавливаем системный поток  
public void stop() {  
    thread = null;  
    System.gc();  
}  
// Рисуем на экране заставку  
public void paint(Graphics graphics) {  
    graphics.setColor(250, 250, 250);  
    graphics.fillRect(0, 0, getWidth(), getHeight());  
    graphics.drawImage(image, getWidth()/2, getHeight()/2,  
        Graphics.VCENTER | Graphics.HCENTER);  
}  
}
```

Реализация класса Loading чем-то напоминает реализацию класса Splash. Это вполне очевидно, поскольку оба этих класса направлены на выполнение примерно одних и тех же действий, а именно показа заставки во время создания и загрузки уровня и игровых объектов.

В момент запуска работы класса Loading на экране рисуется заставка, а под ее прикрытием запускается метод newGame() класса GameMidlet, исходный код которого выглядит следующим образом:

```
public void newGame() {  
try{  
    // загружаем текущий уровень и ставим объекты по позициям  
    gameCanvas.setGame();  
    // запускаем работу игрового цикла  
    gameCanvas.start();  
    // показываем экран с работой класса MainGameCanvas  
    Display.getDisplay(this).setCurrent(gameCanvas);  
    // останавливаем работу класса Loading  
    loading.stop();  
}catch (Exception ex) {  
System.err.println("Class GameMidlet method newGame()");  
}  
}
```

В этом методе мы загружаем текущий уровень, расставляем объекты по позициям, запускаем игровой цикл посредством класса `MainGameCanvas`, отражая результат его работы на экране, а затем останавливаем работу класса `Loading`. В отличие от класса `Splash`, класс `Loading` более универсален, и запуск системного потока, а соответственно, и цикла для вывода графики реализован несколько иначе.

В конструкторе класса `Splash` создавался и запускался системный поток. В игре этот класс нам нужен всего один раз - тогда, когда мы только-только запускаем игру, далее по всей игре класс `Splash` не используется. Поэтому, создав и запустив в конструкторе класса `Splash` системный поток, все обязанности по его закрытию, а также удалению из памяти объекта этого класса мы возлагаем на системные ресурсы телефона. А именно на деструктор класса `Splash`, который остановит работу запущенного системного потока, и сборщик мусора `Java 2 ME`, который, в свою очередь, позаботится о корректном удалении объекта.

С классом `Loading` ситуация несколько иная. Этот класс используется в игре часто и необходим при каждом запуске очередного уровня, поэтому запуск и остановка системного потока в классе `Loading` вынесены в два отдельных метода - `start()` и `stop()`. Механизм работы методов следующий.

В классе `GameMidlet` создается объект `loading` класса `Loading`, а при детальном рассмотрении получается, что этот объект, если так можно выразиться, создан глобально для всего исходного кода игры. То есть через методы `initializationGame()` и `loadingGame()` происходят создание и запуск работы класса `Loading` посредством объекта `midlet` класса `GameMidlet`. В методе `loadingGame()` класса `GameMidlet` совершается запуск системного потока класса `Loading`.

```
loading.start();
```

Вместе с ним соответственно начинается выполнение метода `run()` класса `Loading`, который представляет собой обычный цикл. Без запуска и работы системного потока выполнение цикла `run()` будет иметь лишь единичный вызов этого метода и всего цикла в частности, поэтому для полноценной работы цикла и создается новый системный поток. Запустив системный поток класса `Loading`, мы в любое время с помощью метода `stop()` можем остановить поток на необходимый промежуток времени. То есть, единожды создав объект `loading` класса `Loading`, далее посредством методов `start()` и `stop()` можно останавливать и запускать системный поток и цикл класса `Loading` любое необходимое нам количество раз. Самое главное достоинство этого механизма заключается в том, что объект `loading` во всем исходном коде создается всего один раз, а значит, и системной памяти для содержания объекта выделяется минимальное количество, а точнее один раз на протяжении всей работы игры.

В программах `Java 2 ME` выделение памяти для созданного объекта происходит из ресурсов телефона. Каждый созданный объект отбирает свой объем системной памяти мобильного устройства. Самое интересное заключается в том, что даже после удаления объекта и вызова для этого системных методов `System.gc()` или

`Runtime.gc()` мы не можем гарантировать (особенно в сравнительно старых моделях телефонов) полное освобождение системной памяти телефона от удаляемых объектов!

Получается, что чем больше вы будете создавать объектов в игре, тем меньше памяти у телефона будет оставаться на корректное функционирование самой игры, и может наступить момент, когда для продолжения работы программы у телефона не останется памяти и программа просто-напросто зависнет! Вследствие этого всегда лучше создавать объекты единожды и пользоваться ими на протяжении всей игры, нежели постоянно создавать, удалять и вновь создавать объект одного и того же класса. В этом случае нет гарантии, что в один прекрасный момент ваша игра не зависнет от нехватки системной памяти, выделяемой для работы всей игры.

9.8. Класс MainGameCanvas

Класс `MainGameCanvas` - это один из главных классов всей игры. В этом классе сосредоточена работа игрового цикла программы. Класс `MainGameCanvas` наследует возможности класса `GameCanvas` и реализует интерфейс `Runnable`. Абстрактный класс `GameCanvas` из пакета `javax.microedition.lcdui.game` предоставляет программисту ряд отработанных механизмов для работы с графикой, анимацией, игровым циклом, обработкой пользовательского ввода и многого другого. Класс `GameCanvas` появился во второй версии профиля MIDP 2.0. На рис. 9.6 представлена UML-диаграмма нашего класса `MainGameCanvas`, а в листинге 9.4 находится его исходный код.

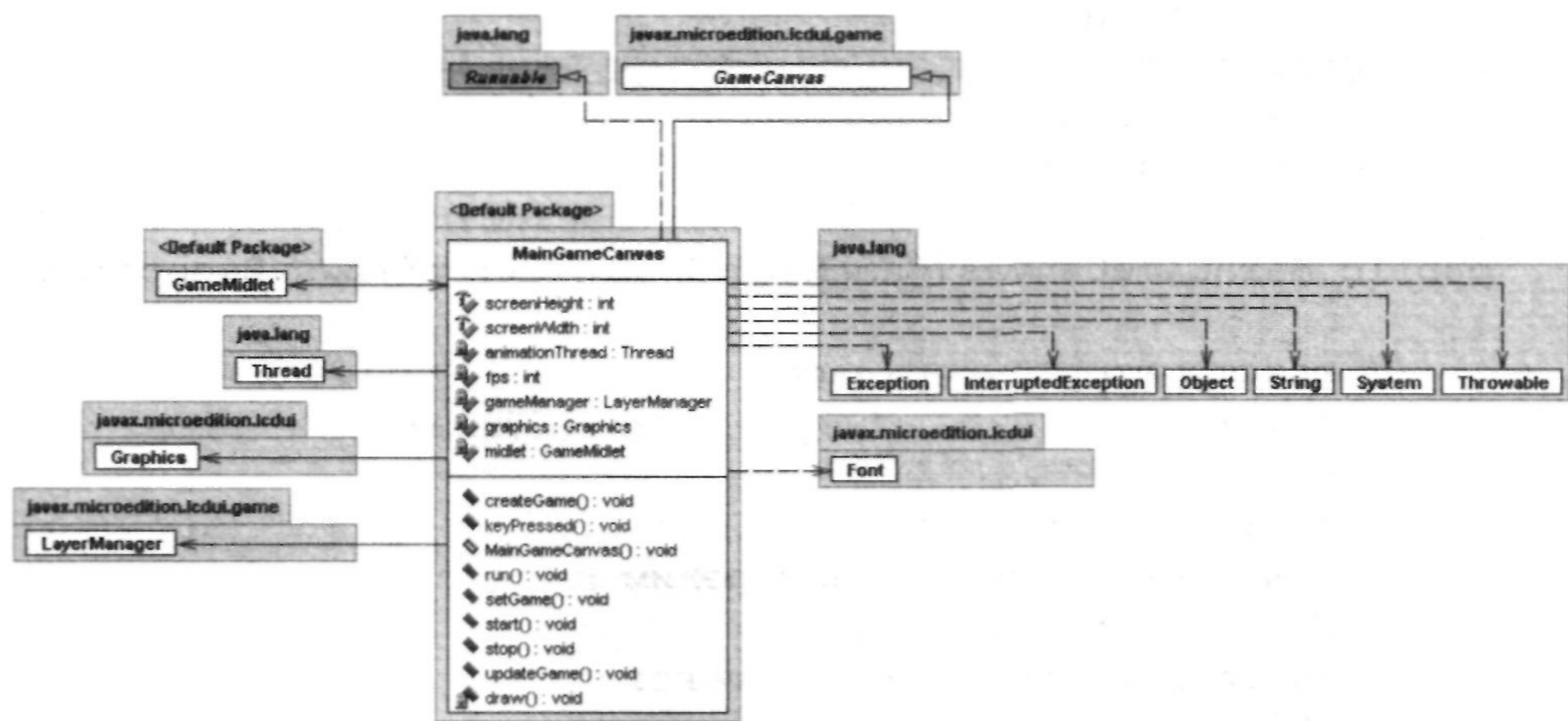


Рис. 9.6. Диаграмма класса MainGameCanvas

```
/*
 * MainGameCanvas.java
 * Игровой цикл
 * ЛИСТИНГ 9.4
```

```
*/

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**
 * @author Stanislav Gornakov
 * (Aversion 1.0
 */

public class MainGameCanvas extends GameCanvas implements
Runnable {
    // мидлет
    private GameMidlet midlet = null;
    // графический контекст
    private Graphics graphics = null;
    // новый системный поток
    private volatile Thread animationThread = null;
    // менеджер слоев
    private LayerManager gameManager = null;
    // переменная для расчета частоты смены кадров
    private int fps = 50;
    // ширина дисплея
    int screenWidth = 0;
    // высота дисплея
    int screenHeight = 0;

    public MainGameCanvas(GameMidlet midlet) throws Exception
    {
        // конструктор суперкласса
        super(true);
        // мидлет
        this.midlet = midlet;
        // включаем полноэкранный режим
        setFullScreenMode(true);
        // создаем графический контекст
        graphics = getGraphics();
        // узнаем ширину экрана телефона
        screenWidth = this.getWidth();
        // узнаем высоту экрана телефона
        screenHeight = this.getHeight();
        // создаем менеджер слоев
        gameManager = new LayerManager();
    }
}
```



```
// создаем игровые объекты
createGame();
}
// создаем и запускаем системный поток
public void start() {
    // создаем новый поток
    animationThread = new Thread(this);
    // запускаем поток
    animationThread.start();
}
// игровой цикл
public void run() {
    Thread currentThread = Thread.currentThread();
    try {
        while (currentThread == animationThread) {
            long startTime = System.currentTimeMillis();
            if (isShown()) {
                // обновляем состояние игры
                updateGame();
                // рисуем на экране
                draw();
                // двойная буферизация
                flushGraphics();
            }
            // синхронизация времени обновления игрового цикла
            long endTime = System.currentTimeMillis() - startTime;
            if (endTime < fps) {
                synchronized (this) {
                    wait(fps - endTime);
                }
            } else {
                currentThread.yield();
            }
        }
    } catch (InterruptedException ie) {}
}
// останавливаем системный поток
public void stop() {
    animationThread = null;
}

// обрабатываем нажатие подэкранных клавиш телефона
public void keyPressed(int keyCode) {
```

```
        if(keyCode == -6 || keyCode == -7){
            // останавливаем поток
            this.stop ();
            // выходим из игры
            midlet.exitGame();
        }
    }
    // выводим на экран графику
    private void draw() {
        graphics.setColor(250, 250, 250);
        graphics.fillRect(0, 0, screenWidth, screenHeight);
        // рисуем на экране телефона
        gameManager.paint(graphics, 0, 0);
        // для красоты нарисуем название игры на экране
        graphics.setColor(0, 0, 0);
        graphics.setFont(Font.getFont(Font.FACE_SYSTEM,
            Font.STYLE_BOLD,
            Font.SIZE_SMALL));
        graphics.drawString("МЕТЕОРИТНЫЙ ДОЖДЬ", screenWidth/2,
            screenHeight/2,
            Graphics.TOP | Graphics.HCENTER);
    }
    // создаем игровые объекты
    public void createGame() throws Exception {
    }

    // загружаем текущий уровень и устанавливаем объекты на
    позиции
    public void setGame(){
    }

    // обновляем состояние игры
    public void updateGame() {
    }
}
```

9.8.1. Глобальные переменные

В области глобальных переменных необходимо объявить ряд объектов, которые впоследствии мы будем глобально использовать в исходном коде класса `MainGameCanvas`. На этом этапе объектов будет значительно меньше, чем в окончательной версии игры, но по мере работы над проектом в каждой последующей главе будут объявляться новые дополнительные объекты.

Сейчас для создания каркаса игры нам понадобятся следующие объекты.

```
private GameMidlet midlet = null;
private Graphics graphics = null;
private volatile Thread animationThread = null;
private LayerManager gameManager = null;
private int fps = 50;
int screenWidth = 0;
int screenHeight = 0;
```

Первый объект `midlet` класса `GameMidlet` дает возможность обращаться к методам класса `GameMidlet`, с помощью которых можно останавливать и запускать игровой процесс, - это мы уже выяснили с вами ранее. Объект `graphics` класса `Graphics` представляет графический контекст устройства и необходим для рисования графики на экране телефона. Следующий объект `animationThread` класса `Thread` представляет системный поток, необходимый для реализации анимации в игровом процессе.

Объект `gameManager` библиотечного класса `LayerManager` представляет собой так называемый менеджер слоев, который дает возможность программисту регулировать количество слоев, рисуемых на экране телефона. То есть вы можете создать слой, который на экране телефона представляет главного героя игры, отдельно создать слой врагов, слой, представляющий фоновый рисунок, артефакты и т. д. и с помощью объекта `gameManager` класса `LayerManager` выводить их на экран в определенной последовательности.

В конце блока глобальных переменных следует объявление трех переменных. Переменная `fps`, исходя из своего названия, необходима для отслеживания количества показанных кадров на экране телефона в одну секунду. С этой переменной мы столкнемся во время изучения игрового цикла. Две следующие переменные `screenWidth` и `screenHeight` впоследствии будут содержать значения соответственно ширины и высоты дисплея телефона, но на данном этапе переменным присваивается значение нуль. Как уже говорилось, при объявлении глобальных объектов и переменных не ленитесь обнулять их, если они объявляются без каких-либо первоначальных значений.

9.8.2. Конструктор класса *MainGameCanvas*

Конструктор класса `MainGameCanvas` содержит исходный код, инициализирующий часть глобальных объектов и переменных. Обратите внимание на строку кода `super(true)`, в которой мы декларируем возможность использования потенциала класса `GameCanvas`.

```
super(true);
this.midlet = midlet;
setFullScreenMode(true);
graphics = getGraphics();
```

```
screenWidth = this.getWidth();  
screenHeight = this.getHeight();  
gameManager = new LayerManager();  
createGame();
```

Во второй строке кода совершается инициализация объекта `midlet`, а также включается полноэкранный режим отображения информации на экране телефона вызовом метода `setFullScreenMode(true)`. Не забывайте вызывать этот метод в каждом своем классе, который отвечает за прорисовку графики на экране, например для класса, представляющего меню, заставку, игровые опции и т. д.

Объект `graphics` создается системным методом `getGraphics()` класса `Graphics`, и это стандартный механизм во всех программах Java 2 ME. Вам не нужно беспокоиться о том, как это происходит на системном уровне, просто вызовите метод `getGraphics()` при инициализации объекта `graphics`.

Некоторые программисты объявляют, создают и инициализируют объект `graphics` непосредственно в игровом цикле или в методе, который постоянно вызывается в этом игровом цикле (что, в принципе, одно и то же). Тогда на каждой итерации цикла происходят повторное объявление, создание и инициализация объекта `graphics`, и кто знает, как поведет себя системная память конкретно взятой модели телефона в этом случае. Более того, если вам потребуется рисовать что-либо на экране телефона за пределами игрового цикла, то нужно будет либо создавать объект `graphics` заново, либо передавать его в качестве параметра в один из методов для рисования графики. Поэтому значительно лучше и проще объявлять и создавать объект `graphics` глобально и один раз для всего исходного кода класса `MainGameCanvas`.

Далее в конструкторе класса `MainGameCanvas` идут две переменные `screenWidth` и `screenHeight`. С помощью библиотечных методов `getWidth()` и `getHeight()` эти две переменные получают текущие значения ширины и высоты экрана телефона. В этих переменных можно указывать и заданные размеры экранов, например для смартфона Nokia N72 ширина составит 179 пикселей, а высота - 208 пикселей. В таком случае под каждый телефон вам придется переписывать указанные значения заново, поэтому значительно проще применять методы `getWidth()` и `getHeight()` для автоматического определения размеров дисплея. Впоследствии при создании игры мы очень часто будем пользоваться переменными `screenWidth` и `screenHeight` для установки объектов на свои позиции, определения различного рода столкновений и многого другого.

Далее в конструкторе класса `MainGameCanvas` происходит создание менеджера слоев `gameManager`, который, как мы уже выяснили, отвечает за количество и порядок вывода слоев на экран телефона. И в конце конструктора происходит вызов метода `createGame()`, к рассмотрению которого мы переходим.

9.8.3. Метод *createGame()*

Исходный код метода `createGame()` выглядит следующим образом, и, как видно из кода, метод пока определен без реализации.


```
// создаем игровые объекты
public void createGame() throws Exception {
    // пока без реализации
}
```

В дальнейшем, работая над проектом, в методе `createGame()` будут создаваться, или инициализироваться, дополнительные объекты других игровых классов. Например, в следующих главах мы начнем работу над игровой картой и создадим для этих целей класс `Background`. Чтобы этот класс, или игровую карту, добавить в игру, необходимо в области глобальных переменных класса `MainGameCanvas` создать объект класса `Background` (дабы объект был виден всему исходному коду класса), а затем в конструкторе класса `MainGameCanvas` создать, или инициализировать, этот объект.

В игре «Метеоритный дождь» будет несколько игровых классов, объекты которых необходимо создавать в конструкторе класса `MainGameCanvas`, и чтобы не сваливать в кучу весь исходный код в конструктор класса `MainGameCanvas`, специально был создан метод `createGame()`. После объявления и создания всех объектов игровых классов необходимо установить их на свои позиции уже непосредственно в игровом процессе и следить за их состоянием. Для этих целей сформированы методы `setGame()` и `updateGame()`.

9.8.4. Установка объектов и обновление состояния игры

После объявления и создания объектов игровых классов необходимо установить объекты на свои позиции. Что это значит? Для всей игры единожды создается определенное количество объектов. На каждом новом уровне мы будем задействовать те или иные объекты, а может и все сразу, но в каждом новом уровне игровая позиция, занимаемая объектом (как, впрочем, и его игровая логика), будет изменяться. То есть на новом уровне объект будет устанавливаться в новом месте и получать какое-то определенное задание. Для этого и создан метод `setGame()`, тело которого пока находится без реализации.

```
// загружаем текущий уровень и устанавливаем объекты на
// позиции
public void setGame(){
}
```

Метод `setGame()` вызывается в методе `newGame()` класса `GameMidlet`, то есть каждый раз при загрузке нового уровня, что позволяет устанавливать объекты на новые позиции, обнуляя позиции в предыдущих уровнях. Получается такая многоступенчатая передача, когда в глобальной области класса `MainGameCanvas` объявляются объекты, затем в его конструкторе они создаются и уже в методе `setGame()` расставляются по своим местам.

Но расставить объекты по местам - это полдела, еще необходимо заставить эти объекты двигаться, думать, стрелять.., то есть жить полноценной жизнью. Для этих целей образован метод `updateGame()`, тело которого пока также находится

без реализации, но где впоследствии будут отдаваться различного рода команды всем созданным объектам.

```
// обновляем состояние игры
public void updateGame() {
}
```

Метод `updateGame()` вызывается на каждой итерации игрового цикла в методе `run()`, что позволяет нам с помощью этого метода производить постоянное обновление состояния игры и пристально следить за всеми фазами игрового процесса. Например, мы объявили, создали и установили на позицию один из метеоритов. Чтобы он двигался, необходимо вызывать соответствующие методы класса `Meteorite` (которые нам предстоит еще создать) в методе `updateGame()`, что позволяет обновлять состояние объекта на каждой итерации цикла, а это порядка 30 кадров, или 30 раз за одну секунду (в идеале). Обновляя состояние объекта 30 раз за одну секунду, мы добиваемся прежде всего плавной анимации и адекватных контрдействий на действие самого игрока. Подробно об игровом цикле и методе `run()` мы поговорим в конце этой главы.

9.8.5. Выводим графику на экран телефона

Мало объявить, создать, установить объекты на позиции и следить за их состоянием, необходим дополнительный механизм представления имеющейся в игре графики на экране. В Java 2 ME эта функция реализована на системном уровне, и нам необходимо только аккуратно вызвать пару-тройку методов, и дело, как говорится, в шляпе. Для этих целей в игре образован метод `draw()`, который вызывается в игровом цикле, представленном методом `run()`.

```
private void draw() {
    // ограничивающий прямоугольник
    graphics.fillRect(0, 0, screenWidth, screenHeight);
    // цвет перерисовки
    graphics.setColor(250, 250, 250);
    // выводим всю графику на экран телефона
    gameManager.paint(graphics, 0, 0);
    // это дополнительный код, рисующий на экране название
    // игры
    graphics.setColor(0, 0, 0);
    // определяем шрифт
    graphics.setFont(Font.getFont(Font.FACE_SYSTEM,
        Font.STYLE_BOLD,
        Font.SIZE_SMALL));
    // выводим на экран название игры
    graphics.drawString("МЕТЕОРИТНЫЙ ДОЖДЬ", screenWidth/2,
        screenHeight/2,
        Graphics.TOP | Graphics.HCENTER);
}
```

Итак, метод `draw()`, который при ближайшем рассмотрении сильно похож на библиотечный метод `paint()`, используемый нами при создании заставок. Все это ягоды одного поля, и они призваны для рисования графики на дисплее, но если метод `paint()` библиотечный, то метод `draw()` мы уже формируем сами.

В первых двух строках исходного кода метода `draw()` создается ограничивающий прямоугольник и устанавливается цвет для перерисовки экрана. Что нам это дает? За одну секунду происходит смена порядка 30 кадров, а значит, происходит 30 перерисовываний всей картинке игры на экране. Если не стирать каждое предыдущее рисование графики, то за одну секунду мы увидим 30 накладываются друг на друга картинок, то есть предыдущая картинка не будет удаляться с экрана.

Чтобы этого не происходило, создается прямоугольник, в нашем случае это вся поверхность дисплея, и 30 раз за одну секунду, то есть после каждой новой картинки, экран отчищается заданным цветом (стирается с экрана), а новая картинка рисуется уже по этому цвету. В качестве цвета перерисовки в программировании игр чаще всего используется черный цвет, реже белый, но можно использовать, например, цвет, близкий к основному цвету игровой карты, или использовать определенный цвет для создания всей фоновой заставки.

Третья и самая главная строка исходного кода вызывает метод `paint()` для объекта менеджера слоев `gameManager`, с тем чтобы вывести всю рисуемую графику на экран. В качестве параметров в метод `paint()` передается объект `graphics`, отвечающий в Java 2 ME за графический контекст устройства, и два целочисленных параметра, устанавливающих по осям X и Y точку вывода графики на экран. По традиции это левый верхний угол дисплея, но в некоторых случаях эти значения могут изменяться. Пример такого подхода будет представлен в следующих главах, когда будут изучаться вывод и движение на экране главного героя игры.

Следующие три строки исходного кода на данном этапе являют собой простую информационную надпись для вывода названия игры, для того чтобы вы хоть что-то увидели на экране устройства, поскольку никаких игровых действий мы еще не реализовывали. Впоследствии эти строки в игре будут удалены.

Теперь давайте рассмотрим метод `keyPressed()`, представляющий обработку нажатий подэкранных клавиш, а затем перейдем к самому сложному - изучению механизма работы игрового цикла.

9.8.6. Обработка клавиш выбора

Получать события с клавиш телефона в Java 2 ME можно различными способами. Метод `keyPressed()` основан на работе с так называемыми *ключевыми кодами*, определенными в виде числовых констант.

Каждая клавиша телефона имеет свой ключевой код, и чтобы узнать, какая из клавиш была нажата, достаточно проверить по номеру клавиши событие: клавиша нажата или нет (конструкция кода `if/else`). Если клавиша нажата, то необходимо выполнить одно действие, а если нет - то другое. Смотрим на исходный код метода `keyPressed()`.


```
public void keyPressed(int keyCode) {  
    if(keyCode == -6 || keyCode == -7){  
        // останавливаем поток  
        this.stop();  
        // выходим из игры  
        midlet.exitGame();  
    }  
}
```

В большинстве моделей телефонов цифры -6 и -7 назначены на две подэкранные клавиши, соответственно на левую и правую. Поэтому и проверка на нажатие этих двух клавиш происходит с помощью цифр -6 и -7. В исходном коде метода `keyPressed()` по нажатии одной из клавиш выбора происходит вызов методов `stop()` и `exitGame()`. Метод `stop()` приостанавливает работу системного потока и игрового цикла в частности на определенный промежуток времени, работу метода мы рассмотрим в следующем разделе, когда коснемся формирования игрового цикла.

Метод `exitGame()` класса `GameMidlet` запускает процесс по обнулению созданных объектов, освобождению захваченных ресурсов и выходу из запущенной программы. То есть по нажатии одной из подэкранной клавиш происходит элементарное закрытие программы - выход. На этом этапе у нас нет игрового меню, и выход из запущенной программы осуществляется напрямую с двух клавиш выбора. Впоследствии, после создания меню, на левую подэкранную клавишу будет назначена команда выхода в меню игры, а на правую клавишу - команда паузы в игре. По повторном нажатии правой клавиши в режиме паузы игра будет запускаться вновь с того места, на котором она была остановлена.

К сожалению, у некоторых производителей телефонов ключевые коды для клавиш отличаются от основной массы кодов других производителей. Крупные ведущие производители стараются придерживаться некоего стандарта, и это правильно, но в некоторых моделях телефонов представленный механизм работать не будет. Информацию о ключевых кодах по производителям можно найти на сайтах производителей телефонов.

Альтернатива использованию ключевых кодов состоит в применении класса `CommandListener`, который мы изучали ранее в этой книге. Но не всегда для обработки нажатия клавиш выбора в играх хорошо использовать класс `CommandListener`. Дело в том, что при обработке событий класс `CommandListener` на подэкранных клавишах создает свое всплывающее меню, выполненное в виде шаблона на базе классов пользовательского интерфейса. В простых программах без особого наличия красивой графики этот механизм приемлем, а вот в играх, где вы рисуете множество разносторонней красивой графики, два всплывающих шаблона меню, причем на разных телефонах, будут выглядеть абсолютно не одинаково, смотреться некрасиво и ни к месту.

Зато на любом телефоне этот механизм будет функционировать одинаково, а вот ключевые коды придется изменять для каждого производителя. Впрочем,

с ключевыми кодами можно и схитрить, создав, например, определенные константы для каждой клавиши и дополнительно отдельный класс, который будет определять производителя телефона и считывать для ваших констант заданную последовательность ключевых кодов.

9.8.7. Игровой цикл

Для полноценной работы игрового цикла необходимо создать системный поток. Системный поток в Java 2 ME позволяет формировать бесконечную работу системного процесса, в котором можно постоянно обновлять экран телефона, а точнее состояние игры и графику, рисуемую на дисплее.

Это как если бы вы плыли на лодке без весел по реке с быстрым-быстрым течением и другой вариант - закрытая заводь, но уже без течения и по-прежнему без весел. То есть когда системный поток создан и запущен в игре, то появляется возможность пересовывать на экране графику или обновлять состояние всей игры. Как только поток остановлен, на экране отобразится последний кадр игры, и он будет статическим без дальнейшей возможности анимации и обновления состояния игры. Если поток запустить с этого места вновь, то выполнение игры продолжится. Механизм запуска и остановки системного потока используется в игре с помощью вызовов двух методов - `start()` и `stop()`.

Создаем и запускаем поток

Метод `start()` создает новый поток и запускает в системе его работу. Вызов этого метода происходит в методе `newGame()` класса `GameMidlet`. Получается так, что при загрузке каждого нового уровня происходят инициализация нового потока и его запуск.

```
public void start() {  
    // создаем новый поток  
    animationThread = new Thread(this);  
    // запускаем поток  
    animationThread.start();  
}
```

Для остановки системного потока используется метод `stop()`.

```
public void stop() {  
    animationThread = null;  
}
```

В этом методе системному потоку, который представлен объектом `animationThread`, присваивается значение, равное нулю, а соответственно поток обнуляется и останавливается. С помощью метода `stop()` легко создать режим паузы в игре, а чтобы продолжить игру вновь, необходимо вызвать метод `start()`.

Каждый выход в меню игры или выход из игры необходимо сопровождать методом `stop()`, останавливая тем самым работу системного потока. Суть запуска

и остановки потока, думается, вам понятна, и теперь можно переходить к созданию игрового цикла.

Создаем игровой цикл

Игровой цикл в программах формируется на базе метода `run()` интерфейса `Runnable`. Для постоянного функционирования этого метода требуется работающий системный поток (метод `start()`). Специфика метода `run()` заключается в том, что при запуске системного потока этот метод становится основной игровой площадкой (как сцена в театре) или мотором, который постоянно работает. Он заводит игру и постоянно выполняет все то, что реализовано внутри самого метода. Поэтому это идеальное место для создания непрерывного игрового цикла, а соответственно, и постоянного обновления состояния игры.

Для создания игрового цикла в методе `run()` используется обычный цикл `while()`. Выполнение этого цикла и создает непрерывную цепь повторяющихся событий, но! Работа цикла `while()` сводится к непрерывному и зацикленному выполнению исходного кода внутри цикла. Это выполнение цикла происходит с громадной скоростью, которая, естественно, может уменьшаться или увеличиваться в зависимости от количества исходного кода внутри цикла или нагрузки на систему (процессор и память телефона). Скорость работы цикла необычайно велика, поэтому необходимо создать механизм, который будет регулировать проход цикла за одну секунду порядка 30 раз (в идеале), что даст нам нужные 30 кадров в одну секунду. Если продолжить сравнение с быстрым течением реки, то нам необходимо построить плотину на ее пути и регулировать скорость течения реки.

Плотины бывают разными, точно так же вариантов для регулирования механизма работы цикла огромное количество! Мы познакомимся с простым и сложным механизмами создания игрового цикла. Простой механизм очень часто используют в простейших играх или программах. В частности, простой цикл мы будем использовать в игре при реализации меню и игровых опций. Сложный цикл построен уже на более профессиональном уровне и позволяет делать точную синхронизацию времени исполнения исходного кода внутри цикла.

Простой игровой цикл

Использование *простого* игрового цикла обусловлено малым наличием исходного кода в программе или внутри самого цикла. Простые циклы построены на основе остановки работы цикла на определенный промежуток времени, по истечении которого работа цикла возобновляется вновь, и так до бесконечности или выхода из игрового цикла. Посмотрите на исходный код простого игрового цикла.

```
// создаем и запускаем системный поток
public void start () {
    loop = true;
    thread = new Thread(this);
    thread.start();
}
```

```
// игровой цикл
public void run() {
    while(loop) {
        // обновляем состояние игры
        updateGame();
        // выводим на экран графику
        draw();
        // метод для перерисовки графики
        repaint();
        // задерживаем работу цикла
    }
    try {
        Thread.sleep (30);
    } catch (Exception ex) {
        System.err.println("Ошибка в цикле");
    }
}

// останавливаем поток
public void stop() {
    loop = false;
    thread = null;
}
```

В этом исходном коде подразумевается, что вы уже объявили и создали необходимый набор объектов и переменных для создания полноценной программы по типу блока классов, рассматриваемых нами в этой главе.

Два метода `start()` и `stop()` соответственно запускают и останавливают работу системного потока. Дополнительная переменная `loop` следит непосредственно за состоянием цикла `while()`. В методе `run()` создается и начинается выполнение цикла `while()`. Методы `updateGame()` и `draw()` служат для обновления состояния игры и вывода графики на экран телефона. Здесь принцип работы программы соответствует принципу, рассмотренному нами ранее в предыдущих разделах этой главы. Метод `repaint()` - это библиотечный метод Java 2 ME. Его функции заключаются в создании механизма постоянного обновления графики на дисплее, поэтому просто вызывайте этот системный метод `repaint()` в игровом цикле для обновления экрана. Следующая конструкция исходного кода цикла обеспечивает задержку работы цикла на определенное время.

```
try{
    Thread.sleep (30);
} catch (Exception ex) {
    System.err.println("Ошибка в цикле");
}
```

Как мы уже говорили, это своего рода плотина, регулирующая скорость течения реки, а в нашем случае - механизм, регулирующий скорость работы системного потока. В этом коде поток останавливается с помощью библиотечного метода `sleep()`. Параметр этого метода задает значение времени, на которое останавливается работа системного потока. Для этих целей используется целочисленное значение 30, почему?

Нам необходимо добиться смены 30 кадров за одну секунду. Время в методе `sleep()` и в программах на Java 2 ME измеряется в миллисекундах, а значит, одна секунда - это 1000 миллисекунд. Делим одну секунду на 30 кадров.

1000 миллисекунд / 30 кадров = 33,3 миллисекунды для одного кадра

В итоге получаем искомое значение времени, на которое необходимо остановить работу всего цикла. Чтобы облегчить нагрузку, распределяемую на процессор телефона, округляем это значение до 30 миллисекунд и подставляем его в метод `sleep()`, добиваясь тем самым 30-кадровой смены графики за одну секунду.

Работу простого цикла хорошо использовать в элементарных программах, где нет большого количества исходного кода, а соответственно, и большой нагрузки на процессор и память. И представьте себе мощную игру с графикой на пару мегабайт, а к этому все и идет, увеличение объема конечного файла игр просто неизбежно. Если вспомнить времена операционной системы DOS, то когда-то десяток-другой игр могли спокойно уместиться на одну дискету, а сейчас пара DVD-дисков для одной игры - это нормальное положение вещей. Большое количество графики и исходного кода приводит к более длительному проходу всех методов внутри игрового цикла, но все в этом вопросе зависит от мощности системных ресурсов телефона (процессор и память).

Сложный игровой цикл

Создать *сложный игровой цикл* можно на основе множества различных схем и вариантов. В организации игровых циклов для своих программ я давно придерживаюсь методики, рекомендованной программистами компании Nokia, где большинство демонстрационных примеров используют один и тот же подход в реализации игрового цикла. Различные демонстрационные примеры можно найти на сайте компании по адресу в Интернете <http://www.forum.nokia.com>.

Это проверенная и надежная система цикла, функционирующая как швейцарские часы. Работа этого игрового цикла построена так же, как и в простом цикле на остановке работы цикла на заданное количество времени. Если в простом цикле значение времени задается жестко, то в сложном игровом цикле это значение так же задается жестко, но при этом дополнительно происходит постоянная синхронизация времени с выполнением цикла. Если графика на экране телефона обновилась быстрее чем 30 миллисекунд, то система производит дополнительную синхронизацию, стараясь выдержать примерно один и тот же временной промежуток для смены всех кадров игры.

На самом деле для мобильных телефонов частота в 15-25 кадров в одну секунду является одной из оптимальных величин. Сравнивать мощность процессора

телефона с компьютерными системами, где этот показатель в десять раз больше, не имеет смысла. Поэтому если вы планируете создавать игру с большим объемом исходного кода и график, то лучше изначально заложить частоту смены кадров, равную 15-25 кадрам. Иначе впоследствии, например на одном из участков игры, где в определенный промежуток времени скопится много игровых процессов, игра может подтормаживать, то есть система начнет выделять больше времени на перерисовку графики или обновление состояния игры. Конечно, здесь все зависит от мощности процессора и объема системной памяти. Например, в последних моделях телефонов, особенно в смартфонах и коммуникаторах, можно смело ставить 30 кадров и более в одну секунду, поскольку и процессор мощный, и системной памяти в достатке.

Теперь перейдем непосредственно к рассмотрению игрового цикла, который используется в нашей игре (класс MainGameCanvas). Смотрим на исходный код цикла и разбираемся с ним.

```
// создаем и запускаем поток
public void start() {
    // создаем новый поток
    animationThread = new Thread(this);
    // запускаем поток
    animationThread.start();
}
// игровой цикл
public void run() {
    Thread currentThread = Thread.currentThread();
    try {
        while(currentThread == animationThread) {
            long startTime = System.currentTimeMillis();
            if(isShown()) {
                // обновляем состояние игры
                updateGame();
                // выводим на экран графику
                draw();
                // двойная буферизация
                flushGraphics();
            }
            // синхронизация времени обновления игрового цикла
            long endTime = System.currentTimeMillis() - startTime;
            if (endTime < fps) {
                synchronized(this) {
                    wait(fps - endTime);
                }
            } else {
                currentThread.yield();
            }
        }
    }
}
```

```
    }  
    } catch (InterruptedException ie) {  
    }  
}  
// останавливаем поток  
public void stop(){  
    animationThread = null;  
}
```

Методы `start()` и `stop()` мы уже рассмотрели ранее, поэтому переходим к работе цикла. В методе `run()` создается и объявляется новый объект `currentThread`, представляющий системный поток. Это такой временный объект или копия системного потока, которому присваивается текущее значение основного потока. Этот механизм позволяет нам сравнивать, а впоследствии и синхронизировать время перерисовки графики. На входе в цикл `while()` происходит сравнение двух значений потоков - `currentThread` и `animationThread`, если эти значения одинаковы, то происходит вход в цикл `while()` и его выполнение. Если нет, то цикл `while()` не выполняется и, соответственно, программа не функционирует. Такая ситуация может возникнуть, когда в методе `stop()` системному потоку присваивается значение нуль и поток останавливается. Этот механизм с остановкой цикла аналогичен рассмотренному простому циклу, где в качестве рычага для остановки цикла использовалась переменная `loop`, которую также можно использовать в сложном игровом цикле взамен сравнения двух значений потоков - `currentThread` и `animationThread`.

После входа в цикл `while()` в самом его начале создается переменная `startTime`, которой присваивается текущее системное значение времени, для чего используется метод `System.currentTimeMillis()`. Впоследствии с помощью этой переменной мы будем синхронизировать затрачиваемое время на перерисовку графики.

Далее в цикле используются конструкция кода `if/else` и метод `isShown()` библиотечного класса `Displayable`. Это стандартный механизм и системный метод для прорисовки графики в сложных мидлетах. Работа этого метода заключается в правильном отображении, или визуализации, всей рисуемой графики на экране. Поэтому просто вызывайте этот метод с параметром `true`, а система сама будет отслеживать правильность представления графики.

Далее в цикле идет вызов трех методов.

```
// обновляем состояние игры  
updateGame();  
// выводим на экран графику  
draw();  
// двойная буферизация  
flushGraphics();
```

Первые два метода мы уже подробно изучили и теперь знаем, что именно в этом месте цикла происходит непрерывное обновление состояния всего игрового

процесса. Третий библиотечный метод `flushGraphics()` - один из важнейших винтиков в работе механизма перерисовки графики. Если вы вспомните простой цикл, то там использовался метод `repaint()`, следящий за обновлением графики. Этот метод применяется при работе с классом `GameCanvas` и характерен для несложных программ под первый профиль MIDP 1.0. Во второй версии профиля MIDP 2.0 был основан новый метод `flushGraphics()`, который по своей структуре более сложный и обеспечивает сравнительно высокую степень и скорость перерисовки графики на экране.

Метод `flushGraphics()` обеспечивает в программах создание вторичного буфера для хранения очередного кадра графики, производя тем самым полноценную двойную буферизацию. Смысл работы двойной буферизации очень прост. Для вывода графики на экран используется дополнительный буфер, в который помещается очередной кадр игры. Этот буфер по своим параметрам идентичен первичной поверхности экрана, и за счет постоянной смены буферов или переключения поверхностей экрана достигается плавная анимация в графике.

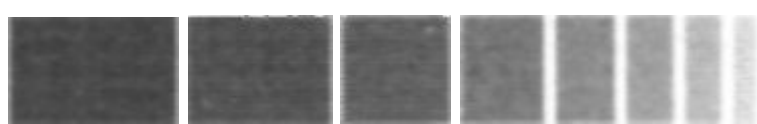
Например, вы вывели на экран главного героя игры и желаете переместить его в другой конец экрана. Для этого необходимо стереть рисунок в этом месте и нарисовать его заново в другом. Если производить эти действия непосредственно на экране, то анимация будет происходить с заметными глазу рывками. Другое дело, когда вы рисуете главного героя в дополнительном буфере, то есть рисуете каждый последующий кадр игры первоначально в дополнительном буфере. В этом случае система на огромной скорости переключает между собой оба буфера (те самые 30 кадров в секунду), добиваясь в результате плавной анимации игрового процесса.

Внутренняя структура системного метода `flushGraphics()` скрыта от программиста, поэтому знать, как именно внутренние сервисы телефона делают эту операцию, необязательно, просто вызывайте этот метод для корректного обновления графики на экране телефона в игровом цикле, а метод `flushGraphics()` сделает за вас все сам.

Последний блок исходного кода в цикле `while()` обеспечивает механизм синхронизации времени, необходимого на смену очередного кадра. Эта система стандартна и достаточно проста. Создается переменная `endTime`, которой присваивается значение текущего системного времени, но уже по окончании единичного прохода цикла. Одновременно с присвоением этого значения происходит вычитание первоначального значения времени до начала прохода цикла, хранящееся в переменной `startTime`. Тем самым мы получаем время, затраченное на один проход цикла. Затем с помощью конструкции исходного кода `if/else` происходит сравнение значения переменной `endTime` и переменной `fps`. В переменной `fps` задано значение времени, необходимое на определение частоты смены кадров в одну секунду. У нас в игре это 20 кадров в одну секунду, которые мы вычисляем по следующей формуле:

$$1000 \text{ миллисекунд} / 50 \text{ кадров} = 20 \text{ миллисекунд для одного кадра игры}$$

При синхронизации времени, если значение переменной `endTime` меньше, чем значение переменной `fps`, или единичный проход цикла осуществился меньше



чем за 20 миллисекунд, то задействуем конструкцию кода для синхронизации времени, необходимого на обновление состояния игры.

```
synchronized(this) {  
    wait(fps - endTime);  
}
```

Если все нормально, то продолжаем работу системного потока вызовом метода `yield()`. Это библиотечный метод Java 2 ME, который следит или продолжает корректную работу системного потока. Вот таким вот образом работает сложный игровой цикл. На этом можно поставить жирную точку. Мы полностью разобрались со всем исходным кодом каркаса классов игры и можем приступить к дальнейшей разработке игры. Исходный код рассмотренного проекта вы найдете на компакт-диске в папке **Code\Chapter9**.

<http://palata-x.narod.ru>

Глава 10. Добавляем в игру меню

Ни одна хорошо сделанная мобильная, компьютерная или консольная игра не обходится без меню. *Меню* - это не просто стартовая страница всего приложения, это хорошо отлаженный механизм, позволяющий пользователю управлять работой программы. В связи с этим необходимо очень тщательно продумывать и планировать работу меню. Старайтесь избегать множественных вложений, непонятных команд и неоправданных лишних диалоговых окон плана: а вы действительно хотите выйти, а вы точно не передумали и т. д. Все должно быть очень просто и в то же время красиво оформлено, и не забывайте о том, что меню - это еще и одна из первых интерактивных страниц, на которую попадет игрок. Если ваше меню будет вызывать у человека полное уныние или он будет «блукать» по нему в поисках кнопки для старта игры, то ваша игра точно надолго не задержится на телефоне пользователя.

В этой главе мы продолжим работу над игрой «Метеоритный дождь» и добавим в игру интерактивное меню. Наше меню будет представлено новым классом под названием *Menu*, в котором мы опишем все действия, связанные с запуском игры, показом дополнительных экранов, создадим механизм перемещения джойстика по имеющимся командам и многое другое. Сначала давайте создадим новый класс *Menu*, отвечающий за реализацию меню в игре, а затем разберем сам механизм вывода на экран телефона игрового меню.

10.1. Идея реализации игрового меню

Что касается способов создания и реализации меню, то тут все зависит только от вашей фантазии. Можно придумать что угодно и сколько угодно, главное - знать и понимать, как работает в целом вся система меню. Я предлагаю один из способов реализации меню, основанный на смене состояния целочисленной переменной. Идея следующая.

Происходит запуск игры, и пользователь попадает в меню. Наше меню представлено одним большим графическим изображением и тремя небольшими по размеру табличками с надписями **Игра**, **Об игре** и **Выход** (рис. 10.1). Каждое изображение поставляется в отдельном файле, а три таблички фактически создают три команды для запуска игры, показа экрана с информацией об игре и выхода из игры. Переход по всем командам будет осуществляться командами джойстика **Вверх** и **Вниз**.

В момент перехода по имеющимся командам, чтобы показать пользователю, какая из команд активна в данный момент, мы будем сдвигать дощечку с надписью выбранной команды влево на 25 пикселей, а неактивные команды будут оставаться на своем месте или возвращаться на свое место. Таким образом, пользователь сразу

заметит, какая из команд активна в данный момент, а какая - нет. Посмотрите на рис. 10.2, где представлен в графическом режиме механизм сдвига табличек. После чего мы перейдем к рассмотрению исходного кода класса Menu.



Рис. 10.1. Игровое меню



Рис. 10.2. Механизм сдвига табличек влево

Примечание. На эмуляторе инструментария Wireless Toolkit в полноэкранном режиме верхняя панель не убирается с экрана, и в связи с этим любое изображение сдвигается на несколько пикселей вниз. Поэтому при запуске примера с меню на этом эмуляторе графика отображается не совсем корректно. Для тестов игрового меню лучше выберите эмулятор одного из производителей телефонов.

10.2. Класс Menu

Очевидно, что для описания игрового меню лучше всего создать отдельный класс, чем помещать исходный код в игровой процесс. Мы так и поступим, создав класс Menu, которому и отведем роль представления меню. В листинге 10.1 показан полный исходный код класса Menu. Давайте сначала внимательно посмотрим на этот код, а затем перейдем к его детальному анализу.

```
/*
 * Menu.Java
 * ЛИСТИНГ 10.1
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import javax.microedition.rms.*;
import javax.microedition.media.*;
```



```
import Java.io.*;

/**
 * @author Stanislav Gornakov
 * @version 1.0
 */

public class Menu extends Canvas implements Runnable{
private GameMidlet midlet = null;
private volatile Thread thread = null;
private Image imageMenu = null;
private Image imageAbout = null;
private Image imageCursorGame = null;
private Image imageCreditAbout = null;
private Image imageCursorExit = null;
private boolean loop = false;
private boolean about= false;
int x45 = 0;
int x70 = 0;
int yCursorGame = 0;
int yCursorAbout = 0;
int yCursorExit = 0;
int state = 1;

public Menu(GameMidlet midlet) {
    this.midlet = midlet;
    setFullScreenMode(true);
    try {
        imageMenu = Image.createImage("/Menu.png");
        imageAbout = Image.createImage("/Splash.png");
        imageCursorGame = Image.createImage("/Game.png");
        imageCreditAbout = Image.createImage("/About.png");
        imageCursorExit = Image.createImage("/Exit.png");
    }catch (Exception ex) {
        System.err.println("In the block =menu= images are
            loaded");
    }
    // Курсоры
    x45 = 45;
    x70 = 70;
    yCursorGame = 180;
    yCursorAbout = 220;
    yCursorExit = 260;
```

```
        state = 1;
    }

    public void start() {
        loop = true;
        thread = new Thread(this);
        thread.start();
    }

    public void run() {
        while (loop) {
            repaint();
            try {
                Thread.sleep(20);
            } catch (Exception ex) {
                System.err.println("Class Menu method run()");
            }
        }
    }

    public void stop() {
        loop = false;
        thread = null;
        System.gc();
    }

    public void paint(Graphics graphics) {
        graphics.setColor(250, 250, 250);
        graphics.fillRect(0, 0, this.getWidth(),
            this.getHeight());
        if (about) {
            graphics.drawImage(imageAbout, 0, 0, 0);
        } else {
            graphics.drawImage(imageMenu, 0, 0, 0);
            if (state == 1) {
                graphics.drawImage(imageCursorGame, x45,
                    yCursorGame, 0);
                graphics.drawImage(imageCreditAbout, x70,
                    yCursorAbout, 0);
                graphics.drawImage(imageCursorExit, x70,
                    yCursorExit, 0);
            }
            if (state == 2) {
                graphics.drawImage(imageCursorGame, x70,
```



```
        yCursorGame, 0 );
        graphics.drawImage(imageCreditAbout, x45,
            yCursorAbout, 0 );
        graphics.drawImage(imageCursorExit, x70,
            yCursorExit, 0 );
    }
    if (state == 3) {
        graphics.drawImage(imageCursorGame, x70,
            yCursorGame, 0 );
        graphics.drawImage(imageCreditAbout, x70,
            yCursorAbout, 0 );
        graphics.drawImage(imageCursorExit, x45,
            yCursorExit, 0 );
    }
}

protected void keyPressed(int keyCode) {
    int action = getGameAction(keyCode);

    if (about) {
        if (keyCode == -7)    about = false;
    }

    switch (action) {

        // нажать Down
        case Canvas.DOWN:
            state +=1;
            if (state > 3) state =1;
            break;

        // нажать Up
        case Canvas.UP:
            state -=1;
            if (state < 1) state =3;
            break;

        // нажать Fire
        case Canvas.FIRE:

            // игра
            if (state == 1) {
                this.stop ();
            }
        }
    }
}
```

```
        midlet.loadingGame();
    }
    // об игре
    if(state == 2){
        about = true;
    }
    // ВЫХОД
    if(state == 3){
        this.stop();
        midlet.exitGame();
    }
    break;
}
}
```

Класс Menu использует изученный нами в предыдущей главе простой механизм игрового цикла на основе создания нового потока. Сформировать цикл и новый поток нам необходимо для того, чтобы получить возможность передвигать таблички по экрану. Без нового потока нельзя организовать циклическое перерисовывание графики на экране телефона, поскольку как в тихой заводи без течения все предметы на воде будут оставаться без движения.

В области глобальных переменных класса Menu мы объявляем ряд переменных и объектов. С некоторыми объектами вы уже знакомы по предыдущей главе, поэтому уделим внимание новым объектам и переменным.

```
private Image imageMenu = null;
private Image imageAbout = null;
private Image imageCursorGame = null;
private Image imageCreditAbout = null;
private Image imageCursorExit = null;
private boolean about = false;
int x45 = 0;
int x70 = 0;
int yCursorGame = 0;
int yCursorAbout = 0;
int yCursorExit = 0;
int state = 1;
```

Первые пять объектов класса Image необходимы нам для загрузки в игру графических изображений, трех табличек с командами, основной подложки меню и изображения для экрана «Об игре».

Затем в этом блоке кода идет объявление булевой переменной about с состоянием false. В дальнейшем по состоянию этой переменной мы будем определять, что именно нам показывать - меню или экран «Об игре». Далее в коде следует объявление сразу шести следующих целочисленных переменных.

- `int x45` - эта переменная содержит координату по оси X, равную 45 пикселям. Когда мы будем сдвигать одну из табличек, то именно это значение по ширине и будет соответствовать сдвигу данной таблички.
- `int x70` - эта переменная у нас задает базовое местонахождение по оси X для неактивных в данный момент табличек.
- `int yCursorGame`, `int yCursorAbout`, `int yCursorExit` - следующие три переменные задают координаты по оси Y, на которые устанавливаются три таблички: **Игра**, **Об игре** и **Выход**.
- `int state` - это главная переменная, которая будет определять состояние активной в данный момент таблички, а вот как именно - вы узнаете позже в этой главе.

Далее в конструкторе класса `Menu` происходят загрузка в игру всех графических изображений и инициализация переменных, отвечающих за точки вывода табличек на экран. Для заставки «Об игре» была использована заставка титульного экрана игры, но вы можете нарисовать свою заставку.

Затем в исходном коде класса `Menu` следуют один за другим три метода `start()`, `run()` и `stop()`, организующие запуск системного потока, его работу и остановку. В предыдущей главе мы подробно останавливались на этих трех методах. Принцип функционирования этих методов остается прежним, поэтому пропускаем их рассмотрение и переходим к двум очень важным методам - `paint()` и `keyPressed()`.

Первый метод `paint()` предназначен для рисования графики на экране телефона. В нашей редакции метод `paint()` функционирует следующим образом. Если состояние переменной `about` равно `false`, то мы рисуем на экране телефона все то, что находится за оператором `else`, то есть игровое меню. Если оператор `about` равен `true`, то вместо игрового меню на экране рисуется заставка «Об игре».

```
public void paint(Graphics graphics) {
    graphics.setColor(250, 250, 250);
    graphics.fillRect(0, 0, this.getWidth(),
        this.getHeight());
    if (about) {
        graphics.drawImage(imageAbout, 0, 0, 0);
    } else {
        graphics.drawImage(imageMenu, 0, 0, 0);
        if (state == 1) {
            graphics.drawImage(imageCursorGame, x45,
                yCursorGame, 0);
            graphics.drawImage(imageCreditAbout, x70,
                yCursorAbout, 0);
            graphics.drawImage(imageCursorExit, x70,
                yCursorExit, 0);
        }
        if (state == 2) {
```

```
        graphics.drawImage(imageCursorGame, x70,
                           yCursorGame, 0);
        graphics.drawImage(imageCreditAbout, x45,
                           yCursorAbout, 0);
        graphics.drawImage(imageCursorExit, x70,
                           yCursorExit, 0);
    }
    if (state == 3) {
        graphics.drawImage(imageCursorGame, x70,
                           yCursorGame, 0);
        graphics.drawImage(imageCreditAbout, x70,
                           yCursorAbout, 0);
        graphics.drawImage(imageCursorExit, x45,
                           yCursorExit, 0);
    }
}
}
```

В итоге для показа экрана «Об игре» нам необходимо просто сменить состояние переменной `about` с `false` на `true`, но об этом чуть позже. Теперь о том, как мы будем определять, какая из команд в данный момент активна.

Для этих целей в исходном коде класса `Menu` создается дополнительная целочисленная переменная `state`. При инициализации эта переменная равна единице.

```
int state = 1;
```

То есть как только на экране телефона отображается игровое меню, то в методе `paint()` срабатывает конструкция кода, где переменная `state` равна единице.

```
if (state == 1) {
    graphics.drawImage(imageCursorGame, x45, yCursorGame, 0);
    graphics.drawImage(imageCreditAbout, x70, yCursorAbout, 0);
    graphics.drawImage(imageCursorExit, x70, yCursorExit, 0);
}
```

В этом случае у нас активна команда **Играть**, а табличка с этой командой сдвигается на 25 пикселей влево. Получается, что нам достаточно просто сменить значение переменной `state` на 2 или 3 - й мы имеем уже сдвинутую влево табличку соответственно с командами **Об игре** или **Выход**. Таким образом, при изменении значения переменной `state` мы добиваемся активизации той или иной команды меню. Но это касается только визуализации графики на экране, теперь нам необходимо создать сам механизм изменения переменной `state` и обработать выполнение команд по нажатию джойстика. Все эти действия происходят в методе `keyPressed()`.

```
protected void keyPressed(int keyCode) {
    int action = getGameAction(keyCode);
```



```
if (about){
    if(keyCode == -7)    about = false;
}

switch(action){

    // нажать Down
    case Canvas.DOWN:
        state +=1;
        if (state > 3) state = 1;
    break;

    // нажать Up
    case Canvas.UP:
        state -=1;
        if (state < 1) state =3;
    break;

    // нажать Fire
    case Canvas.FIRE:

        // игра
        if(state == 1) {
            this.stop ();
            midlet.loadingGame();
        }
        // об игре
        if(state == 2){
            about = true;
        }
        // выход
        if (state == 3){
            this.stop();
            midlet.exitGame();
        }
    break;
}
}
```

Работа этого метода предельно проста. Если переменная `about` равна значению `true`, то на нажатие правой клавиши выбора телефона мы назначаем изменение состояний этой переменной. То есть когда экран с заставкой «Об игре» отображается на экране, при нажатии левой клавиши выбора (можно также добавить и правую клавишу выбора) состояние переменной `about` изменяется

на `false`, а значит, в методе `paint()` на экране телефона рисуется уже игровое меню.

Для игрового меню в методе `keyPressed()` формируется конструкция кода на базе оператора `switch`, который в соответствии с нажатием джойстика определяет, какой из блоков `case` должен функционировать в текущий момент. Выполняя команду джойстиком **Вверх** или **Вниз**, мы увеличиваем или уменьшаем значение переменной `state` на единицу. При этом создаем условие, в котором происходит постоянная проверка значения переменной на больше, чем 3, и меньше, чем 1:

```
if(state > 3) state = 1;
```

или

```
if(state < 1) state = 3;
```

Это условие не дает переменной `state` выйти за пределы значений от 1 до 3 и создает своего рода циклическую прокрутку сверху вниз и наоборот.

А уже на базе состояния переменной `state` в блоке кода `case Canvas.FIRE` (нажатие джойстика по центру) происходит выбор определенных действий. В частности, для запуска игры происходит остановка потока в классе `Menu` методом `this.stop()`. Затем следует запуск работы класса `Loading` посредством выполнения метода `midlet.loadingGame()`. Для показа заставки «Об игре» состояние переменной `about` изменяется на `true`, а для выхода из игры исполняется блок кода за условием `if (state == 3)`.

```
// нажать Fire
case Canvas.FIRE:

    // игра
    if(state == 1){
        this.stop();
        midlet.loadingGame();
    }
    // об игре
    if (state == 2){
        about = true;
    }
    // ВЫХОД
    if (state == 3){
        this.stop();
        midlet.exitGame();
    }
break;
}
```

Все легко и достаточно просто, и, главное, таких методик работы меню можно придумать сколько угодно.

10.3. Планируем запуск меню

При запуске игры на мобильном телефоне пользователь через титульную заставку игры и заставку загрузки попадает прямым в игровой процесс. Нам же необходимо сначала вывести на экран меню и только потом запускать течение игрового процесса, а меню должно стать механизмом управления игрой. Чтобы осуществить этот замысел, нам необходимо обратиться к исходному коду класса `GameMidlet`.

В листинге 10.2 представлен обновленный код этого класса по сравнению с предыдущим проектом. Все нововведения в исходном классе `GameMidlet` я выделил жирным шрифтом, и в дальнейшем все новые строки исходного кода или все добавления будут выделяться жирным шрифтом. Так вам будет значительно удобнее читать и, главное, понимать код программы.

```
/*
 * GameMidlet.java
 * ЛИСТИНГ 10.2
 *
 */

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
/**
 *
 * @author Stanislav Gornakov
 * Aversion 1.0
 */

public class GameMidlet extends MIDlet {
    private MainGameCanvas gameCanvas = null;
    private Loading loading = null;
private Menu menu = null;

    public GameMidlet() {
    }

    public void startApp() {
        try{
            Display.getDisplay(this).setCurrent(new Splash(this));
        }catch (Exception ex) {
            System.err.println("Class GameMidlet метод startApp ()");
        }
    }

    public void pauseApp() {
```

```
}

public void destroyApp(boolean unconditional) {
    garbageCollection();
}

public synchronized void initializationGame() {
    try {
        menu = new Menu(this);
        loading = new Loading(this);
        gameCanvas = new MainGameCanvas(this);
        gameMenu();
    } catch (Exception ex) {
        System.err.println("Class GameMidlet метод
            initializationGame()");
    }
}

public void gameMenu() {
    try {
        menu.start();
        Display.getDisplay(this).setCurrent(menu);
    } catch (Exception ex) {
        System.err.println("Class GameMidlet метод gameMenu()");
    }
}

public void loadingGame() {
    try {
        loading.start();
        Display.getDisplay(this).setCurrent(loading);
    } catch (Exception ex) {
        System.err.println("Class GameMidlet метод
            LoadingGame()");
    }
}

public void newGame() {
    try {
        gameCanvas.setGame();
        gameCanvas.start();
        Display.getDisplay(this).setCurrent(gameCanvas);
        loading.stop();
    } catch (Exception ex) {
```



```
        System.err.println("Class GameMidlet  metod newGame()");
    }
}

private void garbageCollection() {
    menu = null;
    loading = null;
    gameCanvas = null;
    System.gc();
}

public void exitGame() {
    destroyApp(true);
    notifyDestroyed();
}
}
```

В глобальных переменных класса `GameMidlet` происходит объявление нового объекта `menu` класса `Menu`, созданного нами ранее в этой главе.

```
private Menu menu = null;
```

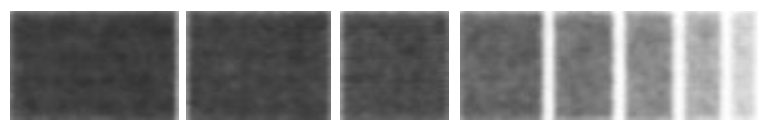
Затем в методе `initializationGame()` происходят непосредственное создание объекта `menu` и вызов нового метода `gameMenu()`. Заметьте, что этот метод заменил ранее вызывавшийся в этом месте метод `loading()`. Таким образом, мы после титульной заставки показываем на экране игровое меню.

```
public void gameMenu() {
    try{
        menu.start();
        Display.getDisplay(this).setCurrent(menu);
    }catch (Exception ex) {
        System.err.println("Class GameMidlet  metod gameMenu()");
    }
}
```

С помощью этого метода происходят запуск системного потока класса `Menu` и представление на экране результатов работы этого класса. Фактически этими действиями мы запускаем работу меню, а далее в самом меню в зависимости от выбранной команды происходит запуск игры, показ экрана «Об игре» или выход из программы.

Последнее нововведение касается исходного кода класса `MainGameCanvas`. В этом классе на нажатие клавиш выбора добавляется вызов метода `gameMenu()`, что автоматически приводит пользователя в меню игры.

```
public void keyPressed(int keyCode) {
    if(keyCode == -6 || keyCode == -7) {
```



```
        this.stop();  
        midlet.gameMenu();  
    }  
}
```

В дальнейшем на обе клавиши выбора мы назначим несколько другой механизм работы перехода к новым уровням, но пока, как только пользователь нажимает одну из клавиш выбора, игровой процесс останавливается и на экране отображается меню игры. Новый вход в игру инициализирует новый игровой процесс.

На этом разбор полетов по созданию игрового меню считаю законченным, и мы переходим к следующей главе, где займемся созданием игровой карты.

<http://palata-x.narod.ru>



<http://palata-x.narod.ru>

Глава 1 1 . Создание игровых карт

В мобильных играх, так же как и в компьютерных играх, существует термин игровая карта. *Игровая карта* - это определенный рисунок или картинка, расположенная на заднем фоне игры и экрана телефона, создающая полноценный антураж законченной игровой сцены. В качестве игровой карты может выступать, например, рисунок неба с тучами или ландшафт какой-то местности. Карта может быть как статической - находиться без движения, так и двигающейся в пространстве в определенном направлении. Обычно в этом случае в игре создается механизм под названием скроллинг.

Скроллинг - это перемещение фоновой картинки или игровой карты в заданном направлении. Скроллинг помогает создать иллюзию движения в игровой сцене, например все того же неба или ландшафта местности. В игре «Метеоритный дождь» мы создадим игровую карту со скроллингом, которая будет изображать большое звездное пространство и флотилию различных кораблей. Движение карты будет происходить сверху вниз, и в этом случае будет казаться, что корабль летит вперед.

1 1 . 1 . Игровая карта

Фоновая картинка и игровая карта - по своей сути это одно и то же. И то, и другое формируют полноценную игровую сцену. Другое дело, что можно использовать несколько различных рисунков и накладывать их друг на друга. В этом случае, например, один рисунок будет представлять небо, которое будет находиться без движения, а другой рисунок - ландшафт местности, который будет двигаться в одном из направлений. Тогда получается, что ландшафт - это действительно игровая карта, а небо - обычный фоновый рисунок. Но на самом деле оба этих элемента игровой сцены близки по своему смыслу и содержанию. Тем более что в мобильных играх оба этих компонента обычно представляются одним и тем же классом под названием `TiledLayer`. Поэтому не стоит делать особых разграничений между фоновым рисунком и игровой картой, все это детали одной игровой сцены, которые позволяют создавать насыщенный игровой процесс.

1 1 . 1 . 1 . Техника компоновки карт

Способов создания карт превеликое множество. Давайте представим для себя один из возможных вариантов создания карты. Например, вы решили создать карту и нарисовали для этого, допустим, большой пейзаж звездного неба. Очевидно, что нарисованный пейзаж звездного неба представляет собой большую картинку, а соответственно, и размер графического файла в килобайтах будет весьма

значительным. В этом случае нагрузка, распределяемая на ресурсы телефона, будет огромной. Конечно, если у вас небо будет представлено всего одним рисунком на весь экран, тогда это одно дело, а если это большая карта, скажем в десять размеров дисплея с элементом скроллинга, то размер графического рисунка в килобайтах этой карты будет очень большим.

Не забывайте и о том, что кроме карты у вас в игре будет дополнительно еще много других графических элементов, которые в сумме составят достаточно большой объем ресурсов, отбираемых у телефона. Хорошо, если это мощный смартфон или коммуникатор с огромным объемом системной памяти и мощным процессором, а если это простенький телефон со средними характеристиками, что тогда? Тогда игра просто не загрузится и не будет работать на этом телефоне.

Самое интересное заключается в том, что при создании все того же звездного неба используется один и тот же набор элементов, в частности синее небо и различные звезды. Если мыслить логически, то достаточно создать некий набор звезд и небольшой по размеру кусочек синего неба и далее все эти компоненты размножить на экране! То есть можно создать несколько различных кусочков звездного неба и сложить из этих кусочков большой пейзаж, как это было сделано в главе 8.

Например, мы имеем три вида звезд, выполненных на синем фоне неба под номерами 1,2,3, и под номером 4 синий кусочек, представляющий весь фон неба. Теперь нам достаточно размножить по всему экрану телефона в определенном или хаотичном порядке все эти кусочки неба - и мы получим полноценный пейзаж звездного неба. Подобный подход в создании и компоновке игровых карт значительно экономит системную память мобильного устройства.

Единственным нюансом в таком подходе компоновки игровой карты является размер этих самых кусочков звездного неба. Ширина и высота одного кусочка могут быть любыми, но все кусочки между собой должны быть одинаковы! Это правило, которое нужно выполнять! Дополнительно необходимо учитывать особенность размеров экрана телефона. Допустим, что мы имеем экран телефона с размером в 176 пикселей по ширине и 220 пикселей по высоте, тогда, чтобы все кусочки вместились в один экран, нужно вычислить примерную ширину и высоту каждого куса относительно экрана, и будет она следующей:

$$16 \text{ пикселей ширина куса} * \text{на } 11 \text{ штук} = 176 \text{ пикселей}$$

$$20 \text{ пикселей высота куса} * \text{на } 11 \text{ штук} = 220 \text{ пикселей}$$

То есть размер одного куса звездного неба, для того чтобы поместиться в один экран, должен составлять 16 пикселей по ширине и 20 пикселей по высоте, а все куски должны быть одинаковых размеров. Естественно, этот подход в подсчете размеров актуален только тогда, когда происходит создание карты под определенный размер экрана телефона. Но обычно игровые карты делаются больших размеров с возможностью скроллинга, например ландшафт, по которому перемещаются персонажи игры, поэтому этот фактор не так критичен. Даже если вы делаете карту ровно под конкретный экран телефона, а кусочки того же звездного неба не подкорректировали, то ничего страшного не произойдет. Все то, что попадет за границы дисплея, отсечется и не прорисуется на экране. Здесь, скорее, уже стоит

вопрос в корректности отображения карты и той действительности, которой вы задумали по представлению пейзажа в игровой сцене.

11.2. Многослойные карты

В Java 2 ME для представления и загрузки в игру карты или фоновых картинок имеется класс `TiledLayer` из пакета `javax.microedition.lcdui.game`, который мы подробно изучили в *главе 8*. С помощью этого библиотечного класса вы можете загружать в игру те самые кусочки рисунка, из которых строится уже полноценная игровая карта. В английском языке и трактовке Java 2 ME такие кусочки рисунков носят название `Tile`. В дословном переводе с английского языка на русский это означает плитка или ячейка. Иногда на профессиональном жаргоне можно услышать слово тайлы - это своего рода англоязычная транскрипция слова `Tile`. У нас все больше используется слово ячейка, поэтому будем придерживаться этого обозначения.

Кстати, в одной из книг по схожей тематике (перевод с английского языка на русский) для этих целей использовался термин - замощенный слой. Когда мне один из читателей задал вопрос на эту тему и спросил, что это такое (он, собственно, и сообщил мне о выходе этого издания), то, честно говоря, сразу даже не было понятно, о чем идет речь. Конечно, это всего лишь маленькие огрехи перевода, а для читателей того издания поясняю, что `Tile Layer` в переводе на русский язык обозначает слой, состоящий из определенного количества ячеек или плиток. То есть, формируя из ячеек (наших кусочков) игровую карту, вы фактически в трактовке Java 2 ME и получаете слой ячеек или плиток.

Здесь главное - понимать, что слой ячеек - это и есть ваша игровая карта. Почему тогда используется название слой? Дело в том, что игровую карту можно создать из нескольких *слоев*, накладывающихся друг на друга. Такой механизм позволяет создавать многослойные проекции игровой сцены, где, допустим, один слой изображает тучи,двигающиеся в одну сторону, другой слой будет представлять землю, нодвигающуюся уже в другую сторону, и т. д. В итоге окончательная игровая карта может состоять из различного количества слоев.

На компакт-диске в папке с названием **Chapter11** я положил два примера, которые сделал специально для вас. Один пример имеет название `Tile`, а другой - `Animated`. Первый пример `Tile` показывает, как можно использовать большое количество различных слоев в игровых картах и перемещать их в пространстве в разных направлениях. А второй пример `Animated` иллюстрирует пример создания анимированной карты. Исходный код обоих примеров несложен, я уверен, что вы сможете разобраться с примерами самостоятельно. Оба проекта делались под разрешение экрана 176 x 208 пикселей, но это не столь важно, и запустить их можно на любом эмуляторе.

11.3. Инструменты создания игровых карт

Создавать карту для любой игры значительно проще и быстрее с помощью редактора карты или редактора уровней (в случае с трехмерными играми). Как правило, редактор карт представляет собой визуальную среду, в которой с комфортом

можно сформировать карту. Для мобильных Java-игр в мире имеется немало подобных средств (JavaMap, Tiled, MapiJavaTiled, Tile Studio...), но лично я уже очень давно и достаточно плодотворно использую бесплатный редактор карт MappyWin32. В этом разделе на примере работы с этим редактором мы создадим карту для игры «Метеоритный дождь».

К большому сожалению, по лицензионному соглашению этот продукт распространять на компакт-диске с книгой нельзя, поэтому вам придется скачать редактор из Интернета самостоятельно. Найти редактор MappyWin32 можно по адресу <http://www.tilemap.com.uk>, благо весит программа всего 1,5 Мбайта. Редактор не требует инсталляции, и вы можете разместить приложение в любом удобном для вас каталоге дискового пространства.

11.3.1. Создаем карту

Перед началом *создания карты* вы, естественно, должны позаботиться о создании и компоновке рисунка с элементами сцены, на базе которых и будет создаваться карта (см. главу 8). Для игры «Метеоритный дождь» используется графический файл Background.png с изображениями кораблей, звезд и туманностей (рис. 11.1).

Размер изображения составляет 240 x 288 пикселей. В игре мы будем использовать ячейки с размером 24 x 32 пикселя. Эти значения могут быть любыми, но, исходя из разрешения экрана, проще всего придерживаться именно этих размеров. Игровая карта по своей ширине будет размером в 240 пикселей (акkurat в размер дисплея), а по высоте - в 1600 пикселей. Получается, что если мерить карту ячейками, то по ширине нам понадобится 10 ячеек, а по высоте - 50 ячеек. Эти значения необходимо определить заранее, поскольку они понадобятся нам на этапе формирования карты в MappyWin32. Теперь давайте перейдем к редактору и создадим карту для игры «Метеоритный дождь».



Рис. 11.1. Графический файл Background.png

Откройте редактор карт MappyWin32 и в рабочем окне выполните команды **File => New Map**. Откроется небольшое по размеру диалоговое окно **Mappy: New Map (easy)**, изображенное на рис. 11.2. В этом окне необходимо указать размеры создаваемой карты. В поле **pixels width** нужно указать размер одной ячейки по ширине (у нас это 24 пикселя), а в поле **pixels high** - высоту одной ячейки (32 пикселя). Затем в поле **tile width** указывается количество используемых ячеек в карте по ширине. Как мы договорились, карта будет равна размеру экрана в 240 пикселей, а значит, 10 ячеек по ширине, нам будет в самый раз. Далее в поле **tile high** необходимо указать высоту создаваемой карты в ячейках, и это значение равно 50 ячейкам. Чтобы сформировать карту на базе указанных значений, нажмите кнопку ОК.

После этих действий в левой части редактора карт появится пустая карта заданных размеров, окрашенная в черный цвет. В свою очередь, с левой стороны у вас будет доступен один черный блок размером 24 x 32 пикселя. Это своего рода

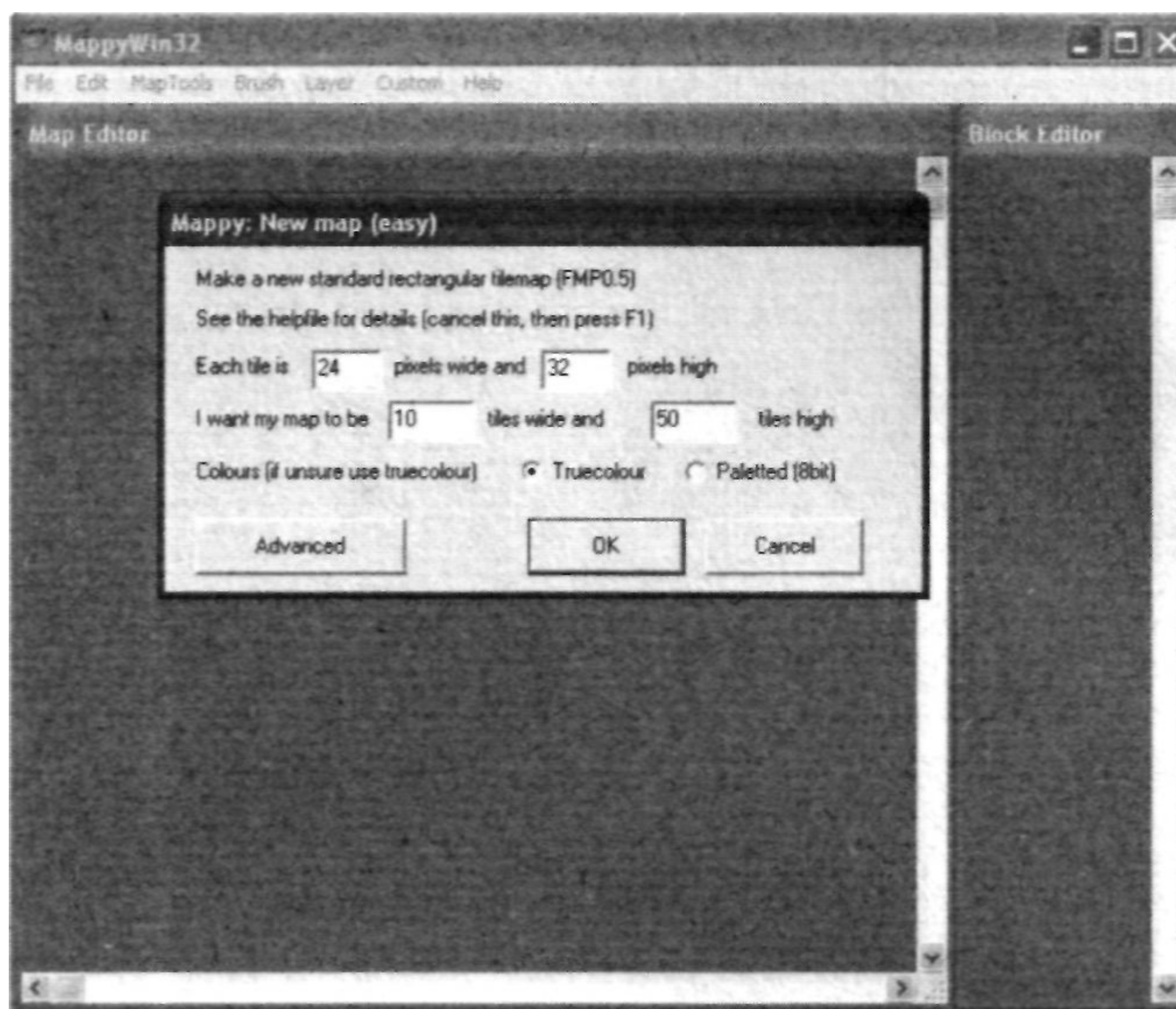


Рис. 11.2. Диалоговое окно Mappy: New Map (easy)

старательная резинка. Далее необходимо загрузить в редактор графический файл `Background.png` и уже на базе этого файла формировать карту для игры.

Для загрузки изображения в редактор выполните в рабочем окне MappyWin32 команды **File => Import**. Откроется стандартное диалоговое окно обзора, с помощью которого вам необходимо проследовать в каталог, где располагается файл `Background.png`, и выбрать его курсором мыши. Моментально в правой части рабочего окна редактора отобразится загруженный рисунок, поделенный на ячейки, с размером 24 x 32 пикселя (рис. 11.3). Теперь самое время сохранить проект на компьютере в любой папке с любым названием.

Механизм создания карты сводится к следующему. Щелчок левой кнопкой мыши на одной из ячеек загруженного в редактор изображения как бы автоматически привязывает выбранную ячейку к курсору мыши. Соответственно, щелчок той же кнопкой мыши, но уже на поверхности самой карты, приведет к вставке этой ячейки в карту. Таким вот незатейливым способом вам необходимо сформировать всю карту, а затем обязательно сохранить весь проект командами **File => Save/Save As**. Для игры «Метеоритный дождь» была сформирована карта, представленная на рис. 11.4.

После создания карты ее необходимо экспортировать в игру, а точнее получить от редактора номера и последовательность используемых ячеек, которые подставляются в массив данных при формировании фона (см. главу 8). Для этого нужно в окне редактора и текущем проекте выполнить команды **File => Export**. Откроется диалоговое окно **Export**, представленное на рис. 11.5. Во всех имеющихся элементах управления этого окна нам необходимо выбрать только один флажок с названием **Map array as comma values only (CSV)**, что позволит сохранить по-

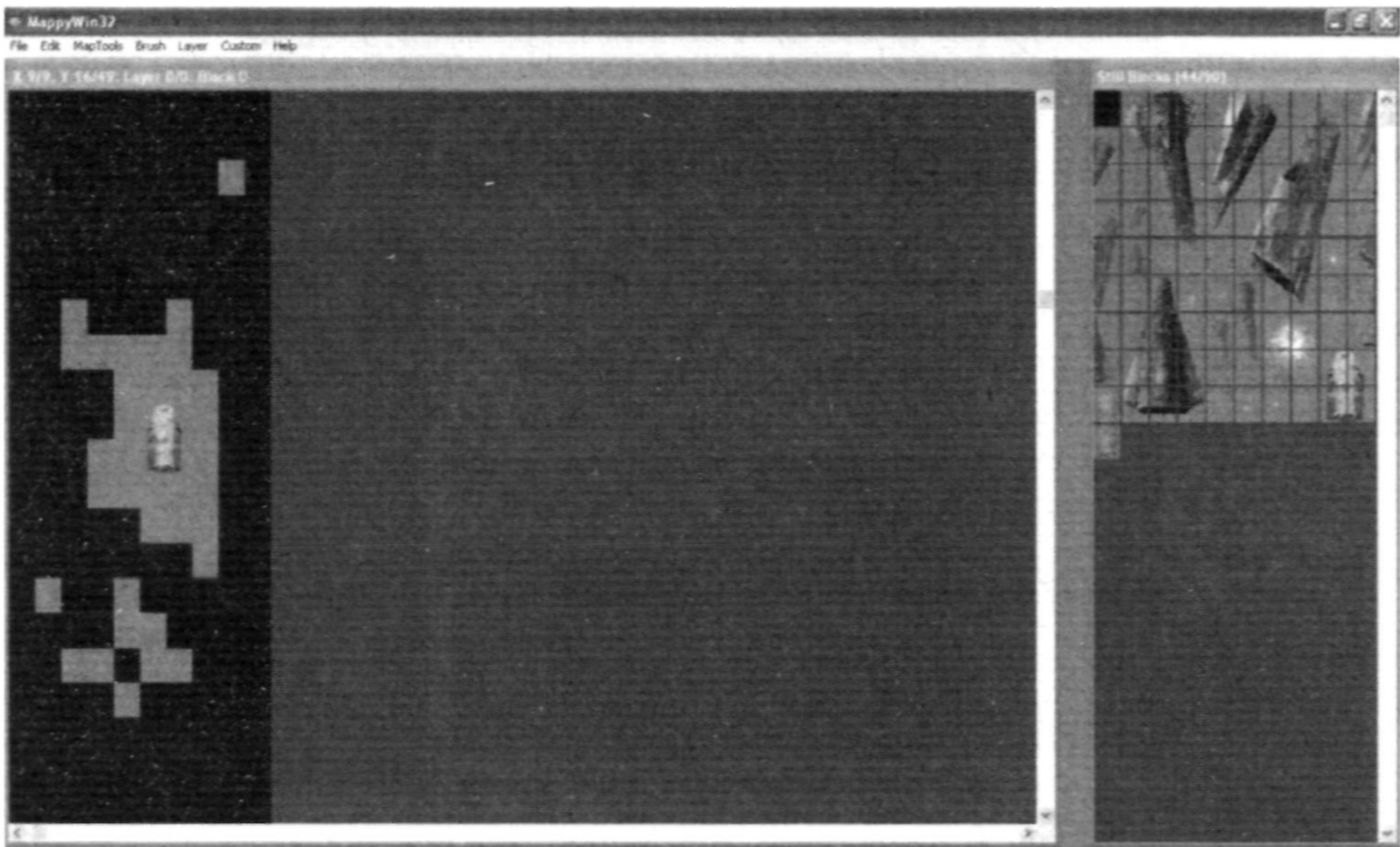


Рис. 11.3. Загруженное изображение в редакторе карт

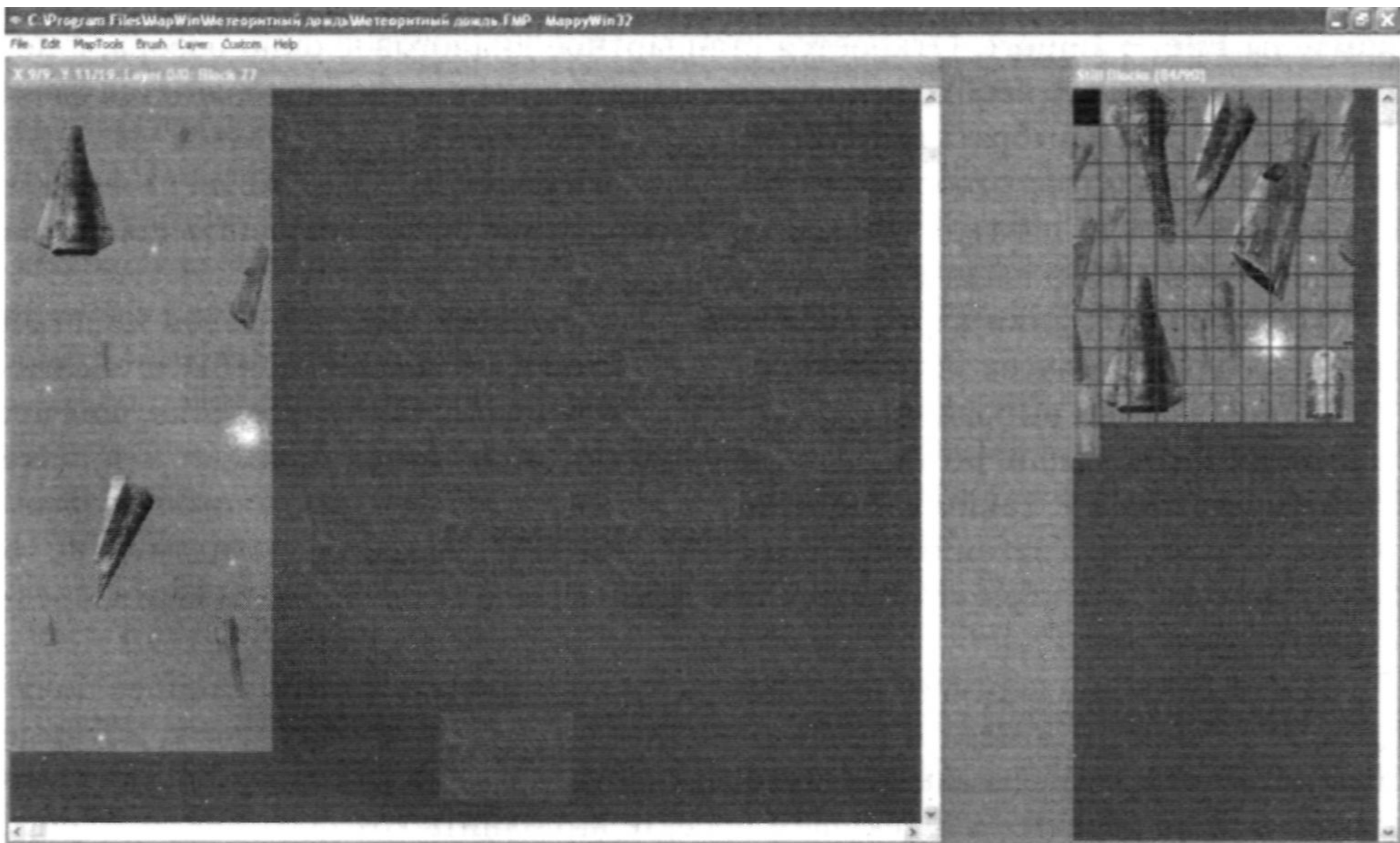


Рис. 11.4. Карта игры «Метеоритный дождь»


```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
/**
 *
 * @author Stanislav Gornakov
 * @version 1.0
 */

public class Background extends TiledLayer {
    static final int WIDTH = 24;
    static final int HEIGHT = 32;
    static final int COLUMNS_WIDTH = 10;
    static final int ROWS_HEIGHT = 50;

    public Background(int columns,int rows, Image image, int
    tileWidth, int
    tileHeight) {
        super(columns, rows, image, tileWidth, tileHeight);
    }

    public void createBackground(){

        int[] mapBackground = {

            21, 21, 48, 87, 87, 21, 84, 4, 5, 6,
            21, 51, 21, 21, 21, 44, 21, 14, 15, 16,
            21, 21, 21, 41, 21, 21, 21, 24, 25, 21,
            21, 21, 21, 21, 44, 21, 21, 34, 48, 21,
            21, 84, 21, 21, 21, 42, 21, 21, 21, 58,
            21, 48, 21, 85, 21, 21, 21, 30, 21, 21,
            21, 21, 21, 21, 21, 21, 39, 40, 21, 21,
            21, 1, 2, 3, 21, 21, 49, 50, 21, 51,
            21, 11, 12, 13, 58, 21, 21, 86, 21, 21,
            53, 21, 22, 23, 21, 53, 21, 21, 21, 21, 9,
            21, 21, 32, 33, 21, 42, 21, 21, 21, 19,
            21, 51, 21, 43, 21, 21, 21, 21, 68, 29,
            21, 42, 21, 21, 21, 53, 21, 21, 21, 21,
            21, 9, 10, 21, 21, 21, 21, 48, 21, 21,
            58, 19, 20, 51, 42, 21, 21, 21, 21, 44,
            21, 29, 21, 21, 21, 21, 21, 21, 53, 21,
            21, 66, 67, 21, 21, 21, 21, 21, 21, 21,
            21, 76, 77, 21, 21, 84, 21, 52, 21, 21,
            21, 53, 21, 21, 21, 21, 61, 62, 63, 21,
            21, 21, 80, 21, 21, 21, 71, 72, 73, 21,
```

```

21, 21, 90, 21, 21, 51, 81, 82, 83, 21,
21, 51, 21, 21, 21, 68, 21, 21, 21, 21,
21, 21, 21, 64, 21, 21, 21, 21, 21, 21,
21, 84, 21, 74, 21, 21, 21, 53, 58, 21,
21, 21, 48, 21, 21, 21, 21, 21, 21, 21,
21, 21, 21, 21, 42, 21, 21, 21, 51, 21,
21, 31, 21, 54, 55, 21, 44, 21, 87, 87,
21, 21, 21, 21, 65, 21, 21, 21, 87, 87,
21, 68, 21, 21, 75, 21, 84, 87, 87, 87,
21, 21, 21, 21, 51, 21, 87, 87, 86, 87,
21, 21, 48, 87, 87, 21, 84, 4, 5, 6,
21, 51, 21, 21, 21, 44, 21, 14, 15, 16,
21, 21, 21, 41, 21, 21, 21, 24, 25, 21,
21, 21, 21, 21, 44, 21, 21, 34, 48, 21,
21, 84, 21, 21, 21, 42, 21, 21, 21, 58,
21, 48, 21, 85, 21, 21, 21, 30, 21, 21,
21, 21, 21, 21, 21, 21, 39, 40, 21, 21,
21, 1, 2, 3, 21, 21, 49, 50, 21, 51,
21, 11, 12, 13, 58, 21, 21, 86, 21, 21,
53, 21, 22, 23, 21, 53, 21, 21, 21, 9,
21, 21, 32, 33, 21, 42, 21, 21, 21, 19,
21, 51, 21, 43, 21, 21, 21, 21, 68, 29,
21, 42, 21, 21, 21, 53, 21, 21, 21, 21,
21, 9, 10, 21, 21, 21, 21, 48, 21, 21,
58, 19, 20, 51, 42, 21, 21, 21, 21, 44,
21, 29, 21, 21, 21, 21, 21, 21, 53, 21,
21, 66, 67, 21, 21, 21, 21, 21, 21, 21,
21, 76, 77, 21, 21, 84, 21, 52, 21, 21,
21, 53, 21, 21, 21, 21, 61, 62, 63, 21,
21, 21, 80, 21, 21, 21, 71, 72, 73, 21
};

for (int i = 0; i < mapBackground.length; i + + ) {
    int column = i % COLUMNS_WIDTH;
    int row = i / COLUMNS_WIDTH;
    setCell(column, row, mapBackground[i]);
}

```

11.5. Загружаем в игру карту

В игре «Метеоритный дождь» вся игровая стратегия построена на том, что игрок должен дойти до конца уровня и остаться живым. Поскольку игровая карта у нас движется сверху вниз, то окончание игровой карты и будет служить окончанием уровня. То есть при старте игрового процесса мы запустим движение карты сверху вниз. Затем как только карта закончится, а точнее ее нулевые координаты совпадут с нулевыми координатами экрана, мы остановим движение карты. Если у игрока к этому моменту останутся жизни, это значит, что он благополучно дошел до конца текущего уровня. В итоге и получается, что окончание карты - это есть окончание уровня, и как только это событие произойдет, мы будем выводить на экран информационную табличку с количеством набранных очков. Дополнительно на две клавиши выбора телефона будут назначены новые команды. На левую клавишу - переход в меню, а на правую - продолжение игры (рис. 11.6). Здесь, кстати, вырисовывается механизм перехода с уровня на уровень. Лакмусовой бумажкой в окончании уровня, как мы помним, служит окончание карты, поэтому можно создать целочисленную переменную, которая будет увеличиваться в этом месте на единицу. Затем при старте новой игры можно передавать эту переменную в качестве параметра в методы `createGame()` и `setGame()`, где, допустим, на базе оператора `switch` избирать очередной уровень для загрузки и выполнения.

Что касается механизма досрочного окончания игры по причине ликвидации корабля, то здесь тоже все предельно просто. В нашей игре навстречу кораблю будут лететь метеориты, каждое столкновение с метеоритом отнимет у корабля жизнь. Если до окончания уровня корабль столкнется с метеоритами больше, чем ему будет отведено для этого жизней, то миссия считается незаконченной и на экран уже выводится табличка с надписью «Вы проиграли». Эту часть механизма мы рассмотрим в следующих главах, а сейчас вернемся к исходному коду класса `MainGameCanvas` и посмотрим на весь исходный код класса, представленный в листинге 11.2. По традиции, все нововведения в исходном коде выделены жирным шрифтом.

```
/*
 * MainGameCanvas.java
 * Листинг 11.2
 */
```



Рис. 11.6. Таблички с надписями об окончании уровня


```
*/

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**
 *
 * @author Stanislav Gornakov
 * Aversion 1.0
 */

public class MainGameCanvas extends GameCanvas implements
Runnable {
private GameMidlet midlet = null;
private Graphics graphics = null;
private volatile Thread animationThread = null;
private LayerManager gameManager = null;
private int fps = 50;
int screenWidth = 0;
int screenHeight = 0;
private Background background = null;
int gameState = 0;
private Image imageGameContinue = null;
private Image imageComandContinue = null;
private Image imageComandMenu = null;

public MainGameCanvas(GameMidlet midlet) throws Exception
{
    super(true);
    this.midlet = midlet;
    setFullScreenMode(true);
    graphics = getGraphics();
    screenWidth = this.getWidth();
    screenHeight = this.getHeight();
    gameManager = new LayerManager();
    createGame();
}

public void start(){
    animationThread = new Thread(this);
    animationThread.start();
}

public void run() {
```

```
Thread currentThread = Thread.currentThread();
try {
while (currentThread == animationThread) {
    long startTime = System.currentTimeMillis();
    if (isShown()) {
        updateGame();
        draw();
        flushGraphics ();
    }
    long endTime = System.currentTimeMillis() -
        startTime;
    if (endTime < fps) {
        synchronized (this) {
            wait (fps - endTime);
        }
    } else {
        currentThread.yield();
    }
}
} catch (InterruptedException ie) {
}
}

public void stop() {
    animationThread = null;
}

public void keyPressed(int keyCode) {

    switch(gameState) {

    case 0:

        if(keyCode == -6 || keyCode == -7) {
            this.stop();
            midlet.gameMenu();
        }
        break;

    case 1:

        if(keyCode == -6){
            this.stop();
            midlet.gameMenu();
        }
    }
}
```

```
        }
        if(keyCode == -7){
            this.stop();
            midlet.loadingGame();
        }
        break;
    }
}

private void draw() {
    graphics.setColor(250, 250, 250);
    graphics.fillRect(0, 0, screenWidth, screenHeight);
    gameManager.paint(graphics, 0, 0);

    switch(gameState) {

        case 1:

            graphics.drawImage (imageGameContinue,
screenWidth/2,
            screenHeight/2,
            Graphics.VCENTER | Graphics.HCENTER);

            graphics.drawImage (imageComandMenu, 0,
screenHeight-
            imageComandMenu.getHeight(), 0);

            graphics.drawImage (imageComandContinue,
screenWidth-
            imageComandContinue.getWidth(), screenHeight-
            imageComandContinue.getHeight(), 0);

            break;
        }

    }

}

public void createGame() throws Exception {
    /*******
    // игровая карта
    /*******
    try{
        Image imageBackground = Image.createImage("/
Backgrounding");
```

```

        background = new
            Background(Background.COLUMNS_WIDTH,
                Background.ROWS_HEIGHT, imageBackground,
                Background.WIDTH,
                Background.HEIGHT);
        background.setVisible(false);
    }catch (Exception ex) {
        System.err.println("Background it is not loaded");
    }
    //*****

    // изображения
    //*****

    try{
        imageGameContinue = Image.createImage("/
            Continue.png");
        imageComandMenu = Image.createImage("/
            comandMenu.png");
        imageComandContinue = Image.createImage("/
            comandContinue.png");
    }catch (Exception ex) {
        System.err.println("Image it is not loaded");
    }
}

public void setGame(){
    gameState = 0;
    background.createBackground();
    background.setPosition(0, -background.getHeight() +
        screenHeight);
    background.setVisible(true);
    //*****

    // ИГРОВЫЕ КОМПОНЕНТЫ
    //*****

    gameManager.append(background);
}

public void updateGame() {
    //*****

    // ДВИЖЕНИЕ ИГРОВЫХ КОМПОНЕНТОВ
    //*****

    if(background.getY() > - 1){
        background.move(0, 0);
        gameState = 1;
    }
}

```



```
    } else{  
        background.move(0, 1);  
    }  
}  
}
```

Начнем изучать код построчно и обратимся к области глобальных переменных, где происходит объявление новых объектов и переменных.

```
private Background background = null;  
int gameState = 0;  
private Image imageGameContinue = null;  
private Image imageComandContinue = null;  
private Image imageComandMenu = null;
```

Объект `background` класса `Background` представляет в программном коде игровую карту. Переменная `gameState` послужит нам верой и правдой для определения смены игровых состояний, о чем мы подробнее поговорим через пару абзацев, а следующие три объекта класса `Image` добавляют в игру три новых графических файла с изображениями табличек для экрана окончания игры (рис. 11.6).

Далее программа входит в тело конструктора класса `MainGameCanvas`, в конце которого следует вызов метода `createGame()`. В этот метод добавляется новый исходный код, с помощью которого происходит загрузка в игру игровой карты и информационных табличек. Методика загрузки изображений вам знакома по предыдущим главам книги, поэтому останавливаться подробно на этом не имеет смысла, за исключением одной новой строки кода.

```
background.setVisible(false);
```

Метод `setVisible()` с параметром `false` позволяет не отражать на экране любой графический объект, загруженный в игру, и лишь вызов `setVisible()` с параметром `true` покажет объект на экране телефона. Это своего рода шапка-невидимка, которая скрывает на определенное время графический объект на экране. Вы спросите, а зачем это нужно?

Дело в том, что любой графический объект, загруженный в программу, автоматически рисуется на экране телефона. Даже если для вывода объекта не назначить координаты его вывода, он все равно будет нарисован на экране, а координаты в данном случае будут равны нулю по обеим осям. Для того чтобы этого не происходило, на любой графический файл, представленный в игре объектом, надевается волшебная шапка-невидимка с лицом человека, похожим на метод `setVisible(false)`.

Далее в методе `setGame()` мы загружаем в игру карту и устанавливаем ее на следующую позицию.

```
background.setPosition(0, -background.getHeight() +  
screenHeight);
```

Суть этой строки кода очень проста. По оси X у нас идет нулевая координата, поскольку размер карты совпадает с размером дисплея, а по оси Y мы «закидываем» карту верх (перемещение карты в игре идет сверху вниз), но при этом оставляем кусок карты, равный по вертикали размеру дисплея `-background.getHeight() + screenHeight`. Это простая математика, возьмите калькулятор и отнимите от нулевых координат дисплея размер карты и прибавьте размер дисплея.

Напоминание. Нулевые координаты экрана - это левый верхний угол дисплея, где положительная ось X идет вниз, а отрицательная ось - вверх. В свою очередь, нулевые координаты карты - это левый верхний угол изображения, где также положительная ось X идет вниз, а отрицательная ось - вверх.

После всех этих действий происходит цикличное выполнение метода `updateGame()`, в котором происходит движение карты сверху вниз методом `background.move(0, 1)`.

```
public void updateGame() {
    /*******

    // движение игровых компонентов

    // -1 в данном случае позволяет нам подстраховаться и
    не показать конец
    // карты или черную кромку карты на экране телефона,
    иначе будет некрасиво
    if(background.getY() > -1 /* 0 */) {
        background.move(0, 0);
        gameState = 1;
    } else{
        background.move(0, 1);
    }
}
```

Дополнительно в методе `updateGame()` создается проверка условия на достижение конца, а точнее начала карты. Если нулевые координаты экрана совпадут с нулевыми координатами карты, то вызывается метод `move(0, 0)` с нулевыми значениями, что соответствует остановке движения карты в игре. Как только это происходит, переменной `gameState` присваивается значение, равное единице, и в методе `draw()` начинается выполнение блока за оператором `case 1`.

Примечание. Вот здесь, кстати, самое время изменить вашу гипотетическую переменную, отвечающую за уровни игры. Например, вот так: `level +=1` - и затем передать значение этой переменной в качестве параметра в методы `createGame()` и `setGame()`.

В этот момент на экран телефона на фоне игры выводятся все три информационные таблички, а в методе `keyPressed()` начинает работать блок кода, следующий

за оператором `case 1`, что приводит к переназначению игровых команд на две клавиши выбора.

```
public void keyPressed (int keyCode) {

    switch(gameState) {

    case 0:

        if (keyCode == -6 || keyCode == -7) {
            this.stop();
            midlet.gameMenu();
        }
        break;

    case 1:

        if(keyCode == -6) {
            this.stop();
            midlet.gameMenu();
        }
        if(keyCode == -7) {
            this.stop();
            midlet.loadingGame();
        }
        break;
    }
}
```

В итоге на левую клавишу выбора (-6) назначается выход в меню, а на правую клавишу выбора (-7) - продолжение игрового процесса. Пожалуй, это все, что можно сказать по исходному коду данного примера, и единственное, о чем еще хотелось бы упомянуть, - это о возможности использования памяти мобильного устройства для хранения различных данных.

11.6. Работа с памятью телефона

Как известно, после закрытия программы все данные уничтожаются, но очень часто необходимо сохранить в памяти устройства различные пользовательские данные, например количество прошедших игроком уровней, оставшиеся жизни и т. д. Для этих целей в Java 2 ME имеется специальный пакет `javax.microedition.rms`.

Мы не будем подробно изучать возможности этого класса, в *приложении 2* вы найдете всю необходимую информацию по этому классу, но я вам хочу показать два универсальных метода, с помощью которых вы сможете записывать любые данные в память телефона и считывать их из памяти.

Оба примера основаны на записи в память целочисленного значения (текущий уровень) и получения из памяти этого значения. Методы универсальны, и вы можете спокойно их применять в своей игре, но не забывайте при использовании этих методов подключить в начало исходного кода пакет `javax.microedition.rms`.

11.6.1. Запись данных в память

```
private void writeLevel(int currentLevel) {

RecordStore store = null;
ByteArrayOutputStream b = null;
DataOutputStream d = null;

try {
    store = RecordStore.openRecordStore("Level", true);
    b = new ByteArrayOutputStream();
    d = new DataOutputStream(b);
    d.writeInt(currentLevel);
    byte[] storeData = b.toByteArray();
    if (store.getNumRecords() == 0) {
        store.addRecord(storeData, 0, storeData.length);
    }
    else {
        store.setRecord(1, storeData, 0,
storeData.length);
    }
}
catch (IOException ioe) {
    System.err.println("Failed work Output Stream");
}
catch (RecordStoreException rse) {
    System.err.println("Failed write record store");
}
finally {
    if (d != null) {
        try {
            d.close();
        }
        catch (IOException ioe) {
            System.err.println("Failed close d");
        }
    }
    if (b != null) {
        try {
```



```
        b.close () ;
    }
    catch (IOException ioe) {
        System.err.println("Failed close b" ) ;
    }
}
if (store != null) {
    try {
        store.closeRecordStore();
    }
    catch (RecordStoreException rse) {
        System.err.println("Failed close store");
    }
}
}
```

11.6.2. Чтение данных

```
public int readLevel() {

    int curentLevel = 1;
    RecordStore store = null;
    ByteArrayInputStream b = null;
    DataInputStream d = null;

    try {
        store = RecordStore.openRecordStore("Level", false);
        byte[] dataStore = store.getRecord(1);
        b = new ByteArrayInputStream(dataStore);
        d = new DataInputStream(b);
        curentLevel = d.readInt();
    }
    catch (IOException ioe) {
        System.err.println("Failed work Input Stream");
    }
    catch (RecordStoreException ex) {
        System.err.println("Failed read record store Level");
    }
    finally{
        if (d != null) {
            try{
                d.close();
            }
        }
    }
}
```

```
        catch (IOException ex) {
            System.err.println("Failed close d" );
        }
    }
    if (b != null) {
        try {
            b.close();
        }
        catch (IOException ex) {
            System.err.println("Failed close b" );
        }
    }
    if (store != null) {
        try {
            store.closeRecordStore();
        }
        catch (RecordStoreException ex) {
            System.err.println("Failed close store");
        }
    }
}

// Если в памяти нет ни одной записи, то возвращается
// значение, равное нулю.
// Нулевого уровня быть не может, поэтому создано
// условие на проверку
// получаемого значения переменной curentLevel
if (curentLevel <= 0) {
    curentLevel = 1;
}
return curentLevel;
}
```

Глава 12. Создание и перемещение корабля по экрану

Графика в играх для любой платформы - это одна из важнейших составляющих. В Java 2 ME имеется множество классов для работы с графикой, один из таких классов - это класс `Sprite`. Класс `Sprite` из пакета `java.microedition.lcdui.game` профиля MIDP 2.0 дает возможность загружать графические элементы игры на экран телефона. Этот класс содержит набор конструкторов, методов и констант для создания всевозможных видов игровой анимации.

В этой главе мы изучим класс `Sprite`, а также создадим свой класс `Ship`, который будет являться подклассом класса `Sprite`, наследуя все возможности своего суперкласса. Дополнительно вы узнаете, как загружать и передвигать корабль с помощью джойстика или игровых клавиш телефона, и попутно мы рассмотрим механизм столкновения корабля с окончанием экрана телефона.

В конце главы вам дополнительно будет предложен исходный код демонстрационного примера **Demo**, с помощью которого будет представлен механизм передвижения корабля и фона одновременно в заданном направлении, на основании расчетов местоположения корабля. Такой механизм работы очень часто используется при создании стратегических игр или игр-ходилок.

12.1. Класс `Sprite`

Как вы знаете из главы 8, изображения в играх, представленные классом `Sprite`, могут быть как анимированными, так и неанимированными. *Неанимированное изображение* - это некий рисунок заданного размера, состоящий из одного кадра, или фрейма. Примером неанимированного изображения может послужить статический илидвигающийся объект без необходимости его анимации в игре. *Анимированное изображение* - это изображение, состоящее из определенного набора фреймов, с помощью которых можно реализовать механизм анимации того или иного объекта на экране телефона. Переход по имеющимся фреймам создает иллюзию анимации в игровом процессе, такой механизм работы используется в компьютерных и мобильных играх, а также в анимации мультипликационных фильмов.

Анимированное изображение может состоять из любого количества фреймов, задающих анимационную последовательность для персонажей игры. Отсчет фреймов изображения происходит от нуля, как это делается в массиве данных. Количество фреймов анимационной последовательности не ограничено. Все фреймы анимационной последовательности обязаны быть одинакового размера, и вы должны знать ширину и высоту одного фрейма. Размер фрейма используется

при создании объекта класса или подкласса `Sprite`. Располагать фреймы изображения можно горизонтально, вертикально или компоновать любым удобным для вас способом. Отсчет по фреймам изображения всегда происходит слева направо и сверху вниз. Посмотрите на рис. 12.1, где изображен шагающий робот.

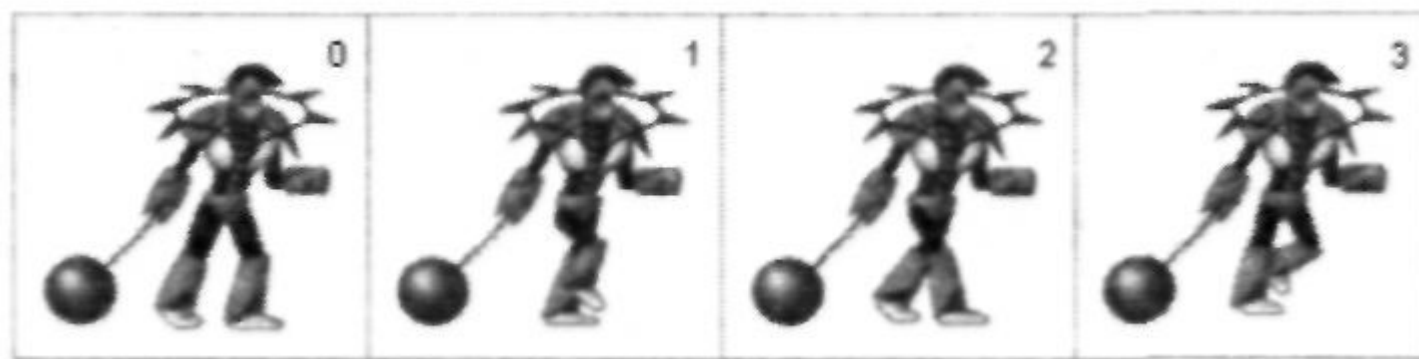


Рис. 12.1. Шагающий робот

Изображение шагающего робота на рис. 12.1 состоит из четырех фреймов анимационной последовательности, где каждый фрейм определяет одну из фаз движения робота. Циклическое передвижение по всем имеющимся фреймам изображения робота создаст эффект его ходьбы на экране телефона.

Класс `Sprite` имеет множество различных методов, с помощью которых можно установить циклическое передвижение по фреймам изображения, переместить кадр на один фрейм вперед или назад, задать с помощью цифровых значений выполнение анимационной последовательности в игре, показать выбранный фрейм и многое другое. Такой подход в анимации персонажей игры используется для создания мобильных игр на Java 2 ME MIDP 2.0.

12.1.1. Конструкторы класса `Sprite`

Класс `Sprite` имеет три разных конструктора для создания объектов этого класса или его подклассов. Каждый конструктор класса `Sprite` определяет вид создаваемого объекта этого класса.

Первый конструктор содержит всего один параметр и создает неанимированный объект.

```
public Sprite(Image image)
```

- `image` - это изображение, которое используется в игре.

Этот вид конструктора класса `Sprite` применяется при создании объекта, который содержит всего один фрейм исходного изображения и не нуждается в создании анимации в игре.

Второй конструктор класса `Sprite` уже содержит три параметра и служит для создания игрового анимационного объекта.

```
public Sprite(Image image,
               int frameWidth,
               int frameHeight)
```

- `image` - это изображение, которое используется в игре.
- `frameWidth` - ширина одного фрейма изображения.
- `frameHeight` - высота одного фрейма изображения.

Первый параметр `image` - это изображение, загружаемое в игру, а два других параметра - `frameWidth` и `frameHeight` - являются значениями ширины и высоты одного фрейма изображения.

И последний, третий конструктор класса `Sprite` создает новый объект, который является копией уже имеющегося в игре объекта класса или подкласса `Sprite`.

```
public Sprite(Sprite s)
```

- s - копия объекта класса или подкласса Sprite.

Копия имеющегося объекта наследует все без исключения возможности оригинала. Этот вид конструктора класса `Sprite` может быть необходим для создания одинаковых объектов одного класса, например для реализации многопользовательского режима игры по Bluetooth или GPRS.

12.1.2. Методы класса *Sprite*

Класс `Sprite` имеет множество хорошо продуманных методов, значительно облегчающих работу программиста при создании мобильных игр. С помощью методов класса `Sprite` можно создавать различные виды анимаций, устанавливать ту или иную фазу трансформации спрайта, определять игровые столкновения. Большинство основных методов класса `Sprite` мы задействуем в игре «Метеоритный дождь» и познакомимся с их работой на практике.

12.1.3. Константы класса *Sprite*

При трансформации спрайта в играх используются константы класса `Sprite`. В профиле MIDP 2.0 описаны восемь констант для трансформации изображений на экране телефона (см. рис. 12.2). В Java 2 ME MIDP 2.0 предусмотрены практически все имеющиеся варианты трансформации спрайта. Работа с константами трансформации очень проста: вызывая метод `setTransform()` с заданной константой для объекта, вы тем самым устанавливаете трансформацию для выбранного изображения в игре.

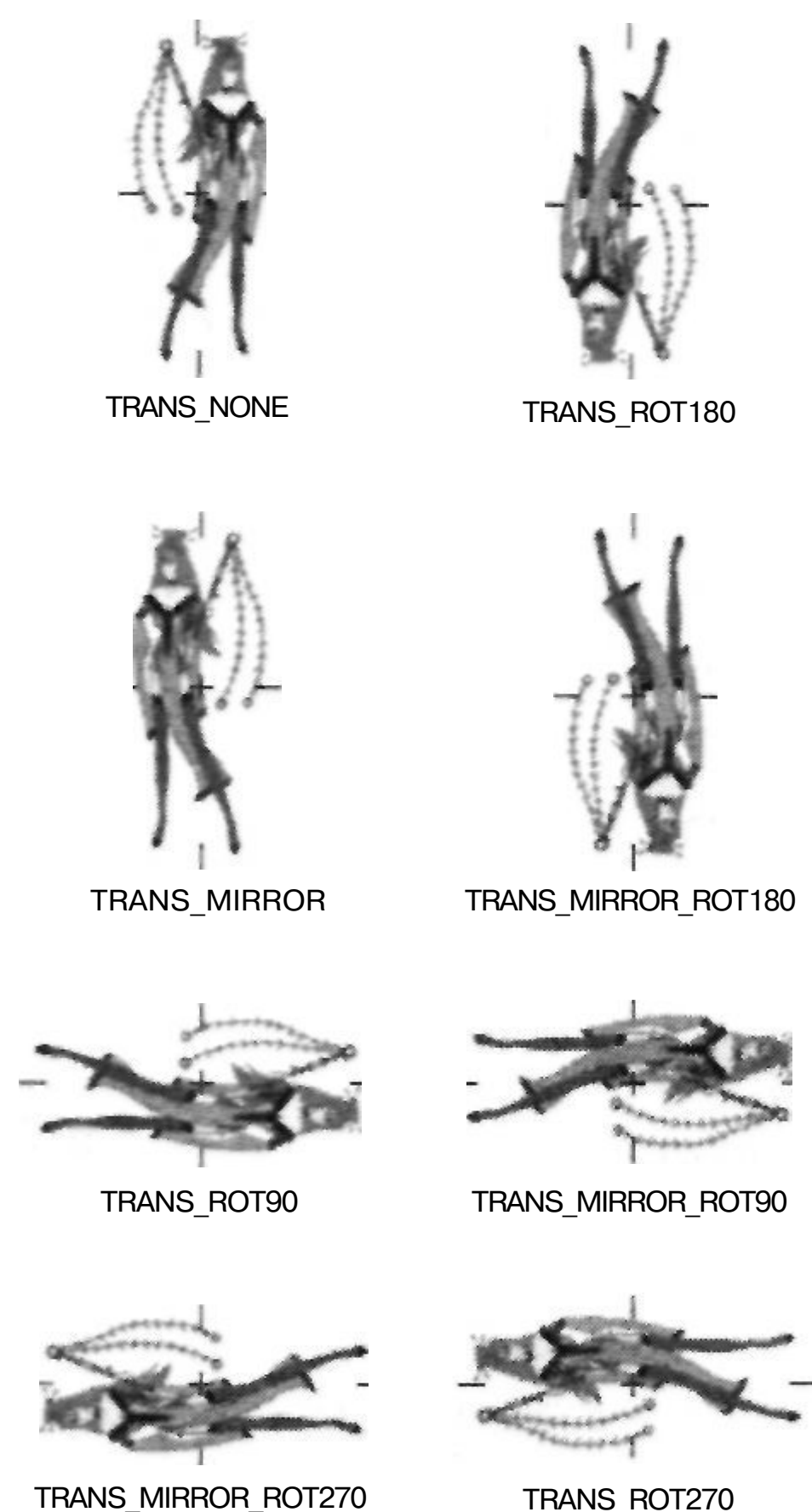


Рис. 12.2. Константы трансформации

системы Senock. Такая обшивка позволяет развивать кораблю огромную скорость и повышенное маневрирование. Корабль оснащен новейшими видами вооружения и генетической системой регенерации патронов и снарядов ко всем известным видам оружия в пределах 8000 галактик на основе различного сырья. Химический двигатель корабля дает возможность кораблю обходиться без дозаправки и смены комплектующих на протяжении более ста лет, пока жива биосоставляющая корабля.

Конечно, такой сюжет звучит неплохо и напоминает фантастическое произведение, вот только как все перечисленное воплотить на дисплее телефона размером 240 x 320 пикселей? Если бы мы с вами делали игру для компьютера или консоли, тогда все было бы проще. Большой экран располагает к созданию максимально детализированных персонажей и окружающего мира. Можно с уверенностью сказать, что рисовать игры для больших по размеру экранов значительно проще, и можно осуществить практически любые фантазии художника.

Ситуация с графикой для телефонов совершенно противоположна, в чем вы смогли убедиться, прочитав *главу 8*. Оптимальный размер корабля в игре «Метеоритный дождь» может быть не более 50 x 50 пикселей, исходя из разрешения дисплея 240 x 320 пикселей. Если сделать корабль большего размера, то играть в игру будет сложно, потому что любой снаряд, вылетевший в сторону корабля, который занимает половину экрана, непременно достигнет своей цели, да и на экране телефона слишком большой корабль будет смотреться несуразно.

72.2.7. Создаем корабль

Изображение корабля (Ship.png) в игре «Метеоритный дождь» состоит из пяти фреймов анимационной последовательности. Каждый фрейм должен символизировать ту или иную фазу движения корабля. При перемещении корабля в игре можно реализовать его движение в направлении вниз, вверх, влево и вправо, а также предусмотреть момент, когда корабль находится без движения.

Посмотрите на рис. 12.3, где показаны пять фреймов анимационной последовательности, необходимые для движения корабля на экране телефона. Перемещая корабль при помощи клавиш, в соответствии с выбранным направлением будет подставляться необходимый фрейм, создавая тем самым иллюзию движения корабля.



Рис. 12.3. Изображение корабля Ship.png

Примечание. Не забывайте о том, что отсчет фреймов анимационной последовательности начинается с нуля, а не с единицы!

12.2.2. Исходный код класса Ship.java

```
/*
 * Ship.java
 * Спецификация класса, описывающая создание и движение
 корабля
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class Ship extends Sprite {
    // объект класса MainGameCanvas
    private MainGameCanvas gameCanvas = null;
    // ширина одного фрейма корабля
    static final int WIDTH = 46;
    // высота одного фрейма корабля
    static final int HEIGHT = 51;
    // конструктор класса Ship
    public Ship(MainGameCanvas gameCanvas, Image image,
        int frameWidth, int frameHeight) {...}
    // движение корабля
    public void moveShip(int direction) {...}
}
```

UML-диаграмма класса Ship представлена на рис. 12.4. Как видно из этой диаграммы, класс Ship наследует возможности суперкласса Sprite. В начале исходного кода Ship.java создаются объект gameCanvas класса MainGameCanvas и две статические переменные WIDTH и HEIGHT. Переменные WIDTH и HEIGHT содержат соответственно ширину (46 пикселей) и высоту (51 пиксель) одного фрейма анимационной последовательности корабля в изображении Ship.png. Эти значения нам нужны как при создании объекта класса Ship, так и при определении столкновений в игре.

Объект gameCanvas будет необходим для доступа к переменным screenWidth и screenHeight, содержащим значения ширины и высоты текущего дисплея телефона. Можно воспользоваться жестко заданными числовыми значениями, но тогда для каждого телефона с другим разрешением экрана придется оптимизировать исходный код игры. Намного проще обратиться к переменным screenWidth и screenHeight класса MainGameCanvas и работать с этими переменными для создания более гибкого исходного кода игры.

По мере создания игры «Метеоритный дождь» мы будем постепенно добавлять новые элементы в класс Ship.java, модернизируя тем самым исходный код и улучшая игру. Перейдем к конструктору класса Ship.

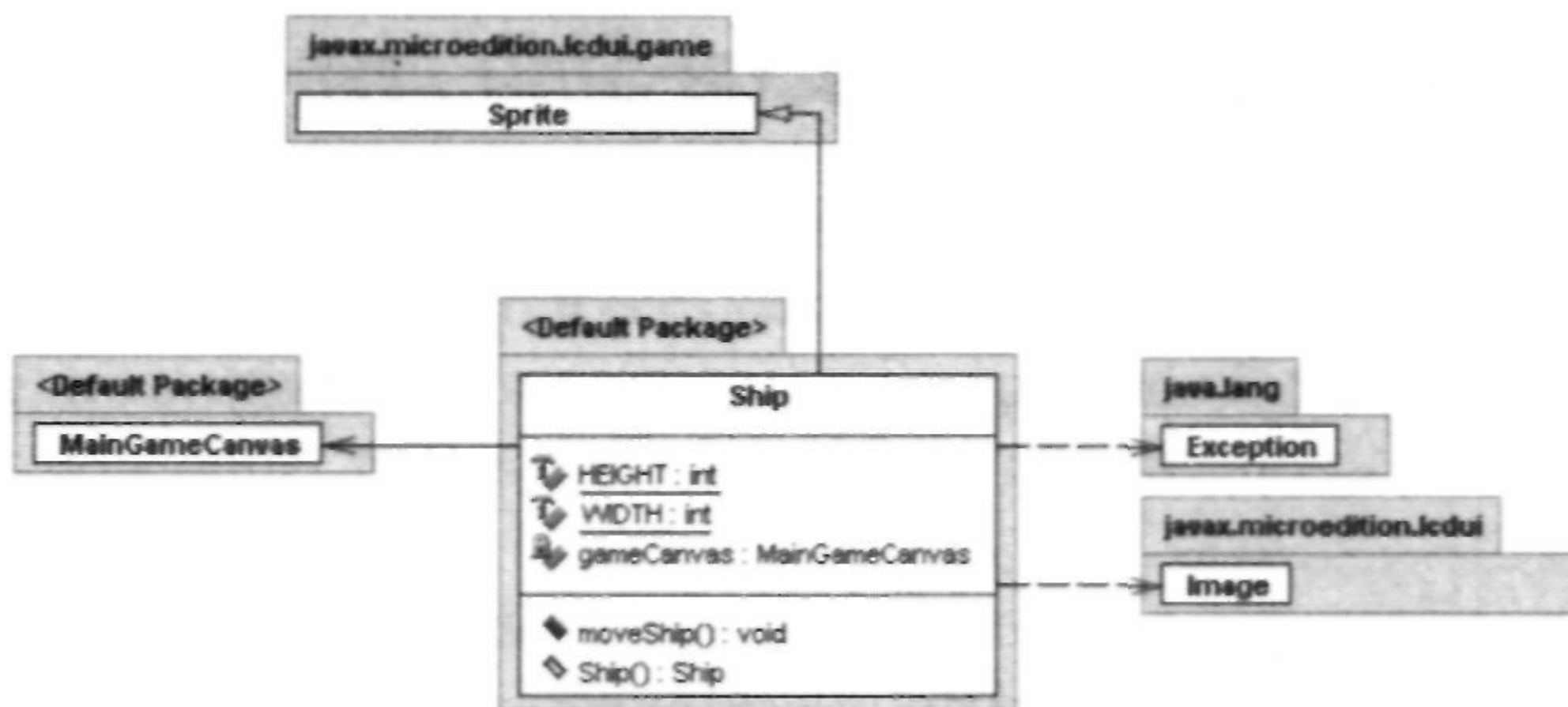


Рис. 12.4. UML-диаграмма класса Ship

Конструктор класса Ship

```

public Ship(MainGameCanvas gameCanvas, Image image, int
frameWidth,
int frameHeight) throws Exception {
    super(image, frameWidth, frameHeight);
    defineReferencePixel(WIDTH/2, HEIGHT/2);
    this.gameCanvas = gameCanvas;
}
  
```

Конструктор класса Ship содержит четыре параметра. Это объект gameCanvas для получения размеров текущего экрана телефона, объект image класса Image для представления изображения корабля в игре и две целочисленные переменные frameWidth, frameHeight, содержащие размер одного фрейма изображения Ship.png.

Первая строка исходного кода конструктора класса Ship вызывает метод super(), с помощью которого происходит наследование всех имеющихся возможностей суперкласса Sprite. Метод super() вызывается с тремя параметрами: image, frameWidth и frameHeight, - что позволяет создать анимацию в игровом процессе.

Загрузка изображения Ship.png происходит непосредственно в исходном коде класса MainGameCanvas, где находится основной цикл игры. Но тем не менее при желании вы можете и на этом этапе загрузить изображение Ship.png в конструкторе класса Ship. Сделать это можно, например, следующим образом:

```
super(Image.createImage("/Ship.png"), 46, 51);
```

Следующий метод в конструкторе класса Ship - это defineReferencePixel(), с помощью которого можно переместить опорную точку объекта.

Опорная точка объекта - это точка, от которой происходит отсчет в момент, например, поворота вокруг своей оси при трансформировании объекта. По умолчанию опорная точка находится в левом верхнем углу, то есть в локальной системе координат для корабля это точка (0, 0). Если следовать физическим законам, то лучше переопределить опорную точку в центр объекта. Посмотрите на рис. 12.5, где схематично изображено преимущество переноса опорной точки.

После переноса опорной точки в центр объекта поворот на месте на 90° выглядит убедительнее, чем у объекта без переопределенной опорной точки. Метод `defineReferencePixel()` переопределяет опорную точку объекта только для трансформаций, а при установке позиции на экране этот метод участия не принимает.

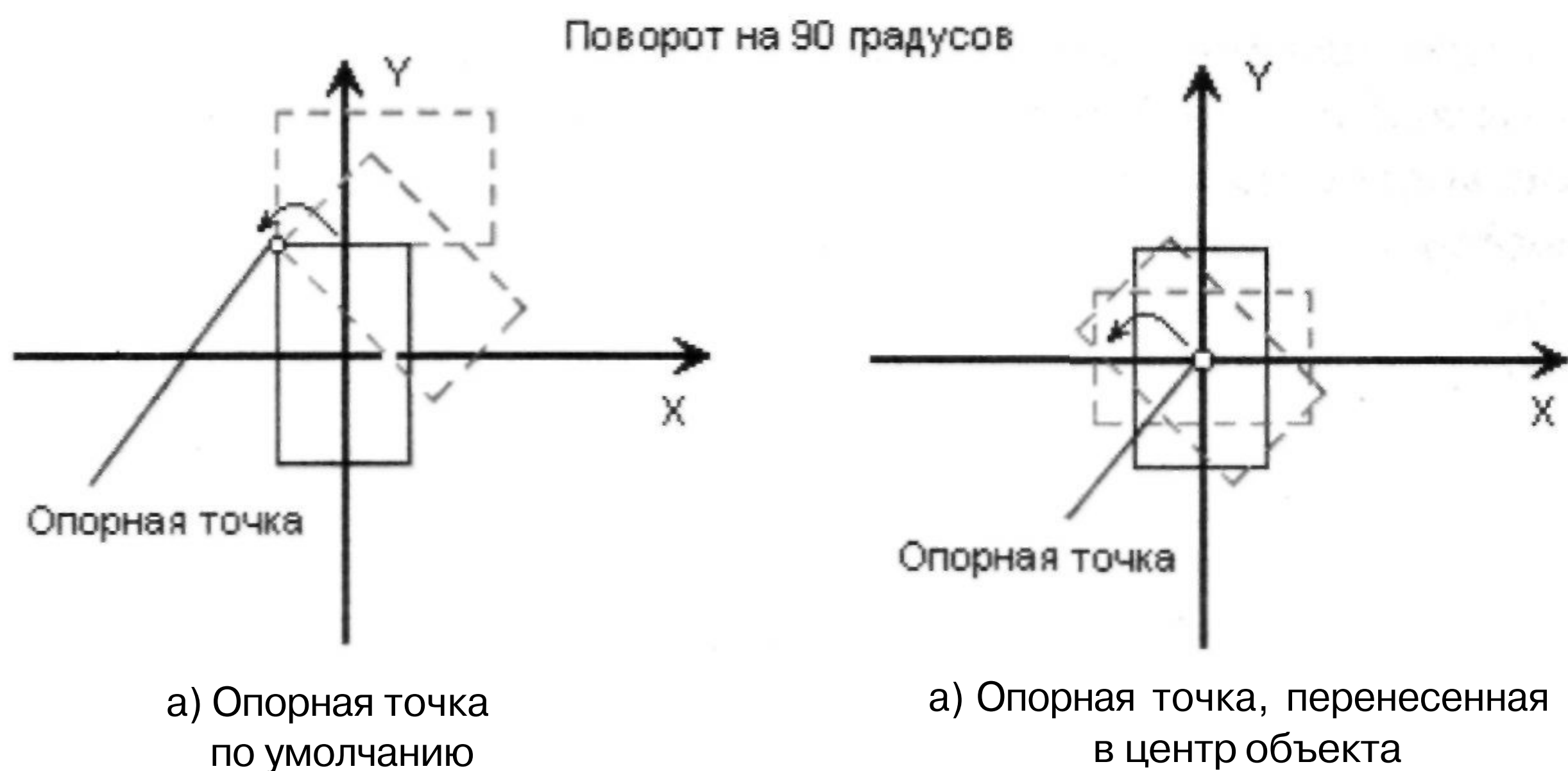


Рис. 12.5. Перенос опорной точки в центр объекта

Последняя строка конструктора класса `Ship` инициализирует объект `gameCanvas` класса `MainGameCanvas`.

```
this.gameCanvas = gameCanvas;
```

Класс `Ship` в окончательном виде будет иметь несколько методов, но на данном этапе нам необходим один метод `moveShip()`, для реализации движения корабля на экране телефона.

Метод `moveShip()`

Алгоритм действий по перемещению объектов на экране телефона построен на использовании метода `move(int x, int y)` класса `Sprite`. Два целочисленных параметра по осям *X* и *Y* задают *скорость перемещения* объекта в игре. Например, если вызвать метод `move(5, 0)` для нажатия джойстика в правую сторону, то при использовании этой команды объект переместится вправо на 5 пикселей от своей начальной точки местонахождения.

Выбор значения скорости для корабля или любого другого объекта в игре происходит, как правило, на основе тестирования игры на мобильном телефоне. Эмуляторы, работающие на компьютере, обычно функционируют в несколько раз быстрее, чем телефоны, поэтому определять на эмуляторе скорость объектов нельзя. При этом также не стоит забывать о том, что на телефонах могут быть установлены различные по мощности процессоры, а значит, и скорость работы самой игры может быть неодинаковой.

К выбору скорости для перемещения главного героя игры необходимо подходить с особой тщательностью. Дело в том, что очень много хороших и красивых игр из-за неправильно выбранной скорости главного героя проигрывают менее удачным играм. Если выбрать маленькую скорость для объекта, то пользователь

не будет успевать уклоняться от препятствий или летящих в него пуль. Будет казаться, что игра сильно тормозит, а главный герой двигается тяжело и неповоротливо, кроме разочарования, такая игра ничего не принесет. Если, в свою очередь, скорость корабля сделать очень высокой, то осуществлять контроль над кораблем будет чрезвычайно трудно.

Количество графики и исходного кода игры значительно увеличивает время на проход одного игрового цикла. Но более мощные телефоны работают гораздо быстрее, поскольку имеют мощный процессор и больше системной памяти. Мощный процессор и большое количество системной памяти несколько увеличивают скорость работы игрового цикла. Это очень заметно. В своих тестах над игрой «Метеоритный дождь» я использовал несколько разных телефонов и коммуникаторов, и почти для каждого устройства приходилось изменять скорость движения корабля, для того чтобы добиться желаемого результата.

Например, на смартфоне Nokia 6600 идеальная скорость движения корабля составила порядка 15 пикселей, тогда как для смартфона Nokia N73 и коммуникатора Qtek 9100 (Windows Mobile 5.0) скорости в 5-8 пикселей оказалось достаточно. Оптимальная скорость движения корабля для мощных моделей телефонов лежит в пределах 5-10 пикселей за один игровой цикл. На менее мощных телефонах оптимальной скоростью движения может быть скорость в 8-15 пикселей за один игровой цикл. Скорость корабля в нашей игре мы задаем в 15 пикселей, поскольку для эмулятора Wireless Toolkit - это оптимальное значение.

Перемещение объектов в пространстве происходит при помощи метода `move()`. Два целочисленных параметра по осям X и Y определяют, на какое количество пикселей можно переместить объект на экране. Чтобы было понятно, как происходит перемещение корабля на экране в игре «Метеоритный дождь», посмотрите на рис. 12.6, где схематично показана работа метода `move()`.

В телефонах, как и в компьютерных играх, для представления 2D-графики используется система координат, точка отсчета которой находится в левом верхнем углу (см. рис. 12.6). Поэтому при движении корабля, например, вправо необходимо увеличить значение по оси X на 15 пикселей `move(15, 0)`. Если нужно переместить корабль влево, то от текущего местоположения корабля уже нужно отнимать 15 пикселей по оси X. То есть в этом случае вызов метода выглядит так: `move(-15, 0)`. При движении по горизонтали значение по оси Y остается равным нулю. Соответственно при движении вверх или вниз нужно увеличивать или уменьшать значение по оси Y, а значение по X оставить равным нулю.

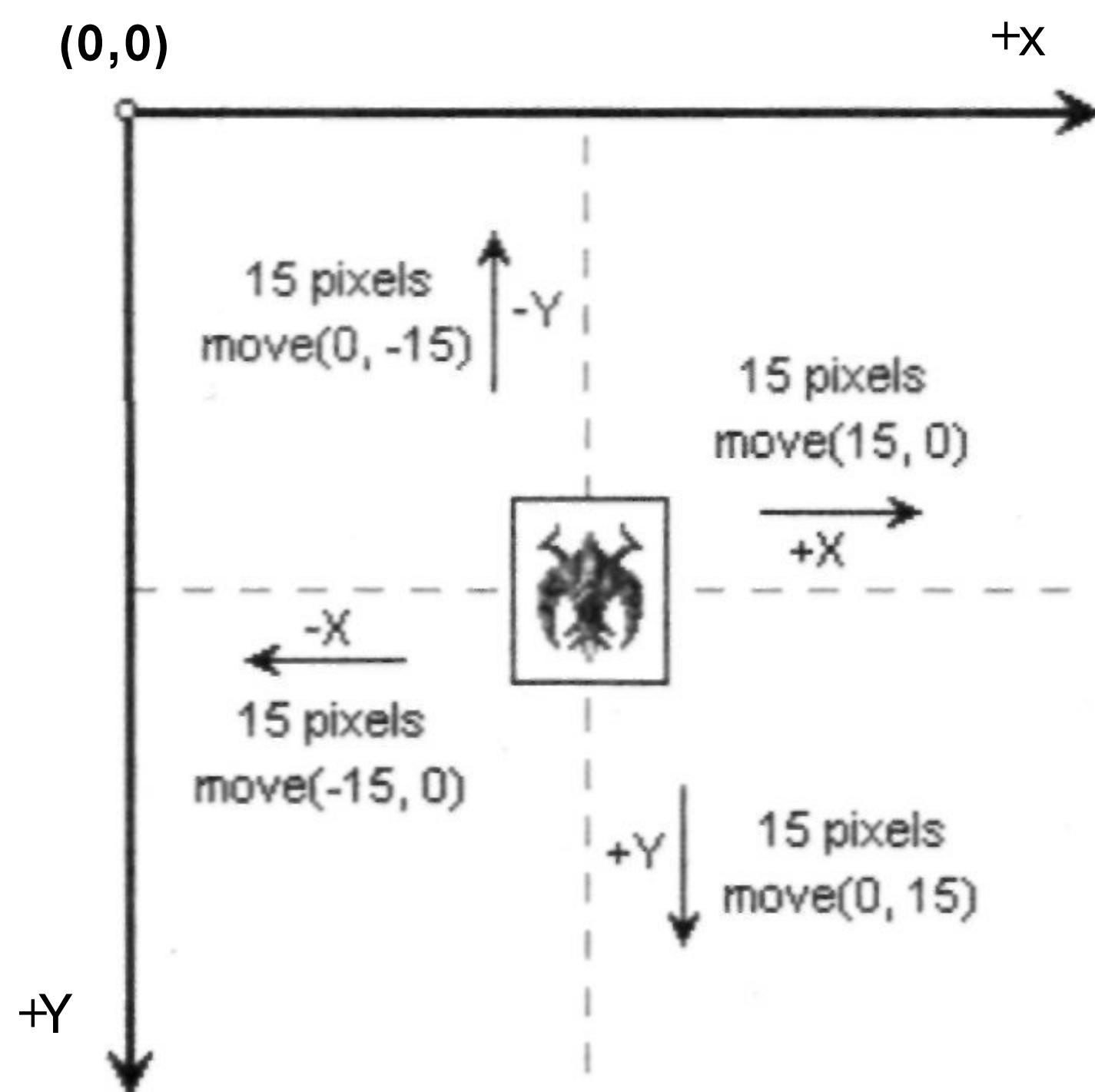


Рис. 12.6. Перемещение объекта методом `move()`

Если использовать оба целочисленных значения и по оси X, и по оси Y, то корабль будет перемещаться по диагонали с заданным углом. Игра «Метеоритный дождь» - это классическая стрелялка, и в таком перемещении необходимости нет, но если вы собираетесь делать, например, гонки на машинах с видом сверху, то такой вариант движения будет необходим.

Для передвижения корабля по экрану в классе Ship создан метод `moveShip()`, основанный на использовании оператора `switch`. Это не единственное решение, которое применимо в данном случае, и можно придумать множество алгоритмов для реализации поставленной задачи, но я решил задачу вот таким образом.

```
public void moveShip(int direction){
    // выбор направления
    switch(direction){
        // стоп
        case 0:
            this.setFrame(direction);
            this.move(0, 0);
        break;
        // движение вниз
        case 1:
            if(this.getY() + HEIGHT + 3 >
                gameCanvas.screenHeight){
                direction = 0;
            }
            else{
                this.setFrame(direction);
                this.move(0, 15);
            }
        break;
        // движение вверх
        case 2:
            if (this.getY() < 3){
                direction = 0;
            }
            else {
                this.setFrame(direction);
                this.move(0, -15);
            }
        break;
        // move left
        case 3:
            if (this.getX() < 3){
                direction = 0;
            }
            else {
```

```
        this.setFrame(direction);
        this.move(-15, 0);
    }
    break;
    // move right
    case 4:
        if (this.getX() + WIDTH + 3 >
            gameCanvas.screenWidth) {
            direction = 0;
        }
        else{
            this.setFrame(direction);
            move(15, 0);
        }
    break;
}
}
```

Метод `moveShip()` имеет один целочисленный параметр `direction`, который задает направление движения корабля. На рис. 12.3 были показаны пять фреймов для реализации движения корабля на экране телефона. Каждый фрейм пронумерован от 0 до 4 и соответствует определенному направлению:

- 0 – стоп;
- 1 – вниз;
- 2 – вверх;
- 3 – влево;
- 4 – вправо.

Точно так же в методе `moveShip()` с помощью цифровых значений для параметра `direction` и оператора управления `switch` задается движение корабля в разные стороны.

В классе `MainGameCanvas`, когда мы будем обрабатывать нажатие пользователем джойстика или специальных игровых клавиш, будет происходить вызов метода `moveShip()` со значением от 0 до 4. Если пользователь выберет команду вправо, то и метод `moveShip()` вызывается со значением параметра `direction`, равным 4.

```
ship.moveShip(4);
```

Оператор `switch` содержит пять вариантов (`case 0` – `case 4`) для выбора направления движения корабля. С помощью `case 0` корабль остается без движения, `move(0, 0)` и выбирается нулевой фрейм в анимационной последовательности изображения `Ship.png`.

```
this.setFrame(0);
```

Метод `setFrame()` класса `Sprite` позволяет программисту выбрать необходимый фрейм изображения в определенный промежуток времени.

Для перемещения корабля вниз переменная `direction` должна быть равна 1. В `case 1` происходят передвижение корабля вниз и обработка ситуации, когда корабль подходит к концу экрана. Для этих целей применяется логическая конструкция операторов `if/else`. На простом языке это звучит так: если корабль не дошел до конца экрана, то происходит движение вниз, а если корабль столкнулся с нижней кромкой экрана, то корабль останавливается.

```
// move down
case 1:
    if(this.getY() + HEIGHT + 3 > gameCanvas.screenHeight){
        direction = 0;
    }
    else{
        this.setFrame(direction);
        this.move(0, 13);
    }
break;
```

Если же не обрабатывать ситуацию, во время которой происходит столкновение корабля со всеми сторонами экрана, то корабль просто исчезнет из поля зрения, перемещаясь в виртуальной реальности где-то вне экрана телефона.

При обработке столкновения с нижней кромкой экрана применяется метод `getY()` (поскольку перемещение корабля происходит по оси `Y`). С помощью этого метода мы получаем координату по оси `Y` для текущей позиции корабля. Позиция корабля, как вы знаете, находится в верхнем левом углу спрайта, и для того чтобы столкновение происходило все-таки с низом корабля, нужно прибавить высоту изображения корабля (`HEIGHT`), равную 51 пикселю.

```
this.getY() + HEIGHT + 3 > gameCanvas.screenHeight
```

Три дополнительных пикселя прибавляются к высоте корабля для красоты, чтобы у игрока не создавалось иллюзии полного столкновения с окончанием экрана.

Размер высоты экрана телефона у различных моделей может быть разным, поэтому лучше использовать значение переменной `screenHeight`, в которой содержится высота текущего экрана, определенного в классе `MainGameCanvas` методом `getHeight()`. А поскольку `screenHeight` - это член класса `MainGameCanvas`, то и использование этой переменной происходит следующим образом: `gameCanvas.screenHeight`.

При столкновении корабля с нижней частью экрана переменной `direction` присваивается значение, равное 0, и работа метода `moveShip()` переходит к `case 0`, что равносильно остановке корабля. Если столкновение с нижней кромкой не обнаружено, то выбирается фрейм под номером 1 изображения `Ship.png` и вызов метода `move(0, 15)` для перемещения корабля вниз.

Для перемещения корабля вверх по экрану в методе `moveShip()` применяется `case 2`. Тогда движение корабля происходит так же по оси `Y`, но уже с отрицательным значением:

```
this.setFrame(direction);
this.move(0, -15);
```

Методом `setFrame()` выбирается фрейм 2, который соответствует движению корабля вверх. Обработка столкновения с верхней частью экрана происходит как и при движении вниз. В качестве точки столкновения с кораблем используется значение в три пикселя (как уже говорилось, это сделано для красоты).

При движении корабля влево и вправо для определения позиции корабля необходим метод `getX()`, потому что происходит передвижение корабля по оси X. Модель определения столкновения с левой и правой частями экрана, а также движение корабля по экрану аналогичны рассмотренному механизму по перемещению корабля вверх и вниз. Посмотрите на рис. 12.7, где показан принцип работы метода `moveShip()`.

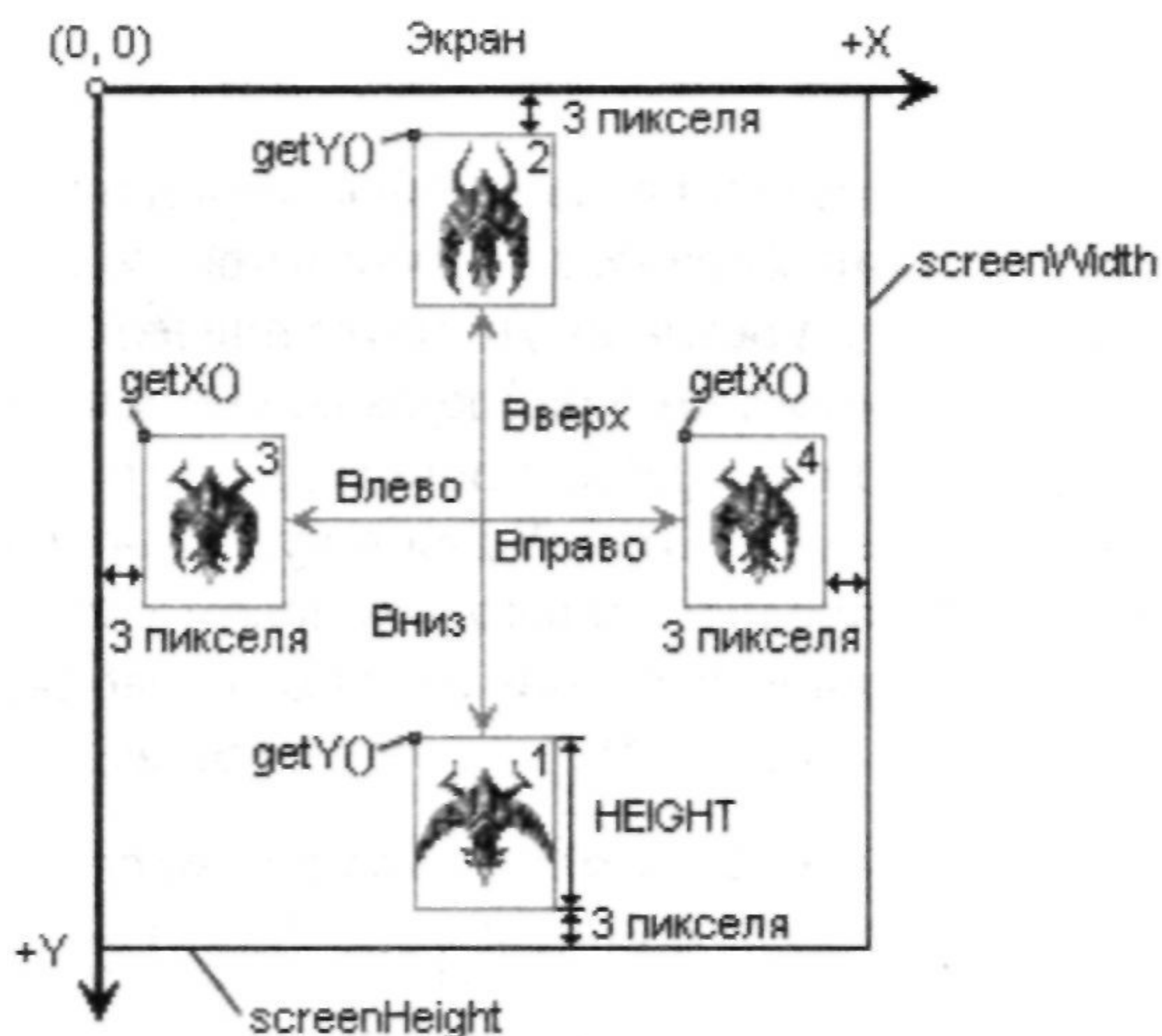


Рис. 12.7. Движение корабля по экрану телефона

После того как был создан класс `Ship` и описана его спецификация, можно перейти к исходному коду класса `MainGameCanvas` для создания и рисования корабля на экране телефона.

12.2.3. Создание объекта класса *Ship*

Алгоритм действий по рисованию изображения корабля на экране и обработке нажатий клавиш телефона пользователем сводятся к следующим шагам.

1. Объявление объекта `ship` класса `Ship`.
2. Инициализация объекта и загрузка изображения.

3. Определение позиции на экране и добавление объекта ship посредством менеджера уровней.
4. Обработка событий, полученных с клавиш телефона, для перемещения корабля по экрану.

Все перечисленные шаги по созданию объекта ship делаются в классе `MainGameCanvas`, исходный код которого выглядит следующим образом:

```
/*
 * MainGameCanvas.java
 * Класс MainGameCanvas создает и рисует игровые объекты.
 * Обработывает нажатие с клавиш телефона.
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class MainGameCanvas extends GameCanvas
implements Runnable {
    // midlet
    private GameMidlet midlet = null;
    // graphic context
    private Graphics graphics = null;
    // thread
    private volatile Thread animationThread = null;
    // менеджер уровней
    private LayerManager gameManager = null;
    // фон
    private Background background = null;
    // fps
    private int fps = 50;
    // ширина
    int screenWidth = 0;
    // высота
    int screenHeight = 0;

    public MainGameCanvas(GameMidlet midlet) throws Exception
    {
        // конструктор суперкласса
        super(true);
        // midlet
        this.midlet = midlet;
        // полноэкранный режим
        setFullScreenMode(true);
        // graphic context
```

```
graphics = getGraphics();
// ширина
screenWidth = this.getWidth();
// высота
screenHeight = this.getHeight();
// менеджер слоев
gameManager = new LayerManager();
// создаем игровые компоненты
createGame();
}

// создание и запуск потока
public void start(){...}
// игровой цикл
public void run(){...}
// остановка потока
public void stop(){...}
// обработка soft key
public void keyPressed(int keyCode) {...}
// рисование игры на экране
private void draw(){...}
// создание объектов игры
public void createGame() throws Exception {...}
// определение игровых позиций
public void setGame() {...}
// обновление игры
public void updateGame() {...}
}
```

Перейдем непосредственно к пошаговому созданию и рисованию объекта ship на экране телефона.

Объявление объекта класса Ship

В исходном коде класса MainGameCanvas объявим объект ship класса Ship:

```
private Ship ship = null;
```

Объявление объекта происходит глобально для всего исходного кода класса MainGameCanvas. Объект ship будет очень часто использоваться для обнаружения столкновений, выстрелов и других действий, поэтому необходимо именно глобальное создание объекта ship.

Инициализация объекта и загрузка изображения

В методе createGame(), который был образован специально для создания объектов игры, создается объект ship.

```
public void createGame() throws Exception {
```



```
//*****
// карта
//*****

...

// создание ship

try{
    Image imageShip = Image.createImage("/Ship.png");
    ship = new Ship(this, imageShip, Ship.WIDTH,
        Ship.HEIGHT);
    ship.setVisible(false);
    imageShip = null;
}catch (Exception ex){
    System.err.println("Ship.png it is not loaded");
}
}
```

В создании объекта `ship` используется конструктор класса `Ship` с четырьмя параметрами, дающий возможность создавать игровую анимацию. В параметре `imageShip` содержится изображение `Ship.png`, загруженное методом `createImage` класса `Image`. Ширина и высота одного фрейма изображения `Ship.png` определяются параметрами `Ship.WIDTH` и `Ship.HEIGHT`.

В строке кода

```
ship.setVisible(false);
```

мы исключаем отображение корабля на экране при инициализации игровых объектов, но в методе `setGame()` при старте игры метод `setVisible()` будет вызван уже со значением `true` для представления корабля на экране.

По окончании создания объекта `ship` изображение, содержащееся в `imageShip`, следует обнулить. При создании объекта используется конструкция `try/catch` для обработки исключительных ситуаций, возникающих при загрузке изображений в исходном коде класса `MainGameCanvas`.

Определение позиции на экране

Для определений позиций объектов в игре «Метеоритный дождь» специально создан метод `setGame()`, где для объекта `ship` происходит выбор позиции на дисплее телефона.

```
public void setGame(){
    //*****
    // ship
    //*****
    // позиция корабля
    ship.setPosition(screenWidth/2 - ship.WIDTH/2,
```

```
screenHeight - ship.HEIGHT - 10);
//ship visible
ship.setVisible(true);
//*****

// background
//*****

...

// добавляем игровые компоненты

// добавляем ship
gameManager.append(ship);
// добавляем background
gameManager.append(background);
}
```

Метод setPosition() устанавливает позицию объекта в игре по осям X и Y.

```
this.setPosition(gameCanvas.screenWidth/2 - WIDTH/
2,gameCanvas.screenHeight -
HEIGHT - 10);
```

Первый параметр - это точка по оси X, второй - по оси Y, и если придерживаться числовых значений, то для телефонов с разрешением экрана 240 x 320 расчет позиции будет выглядеть так:

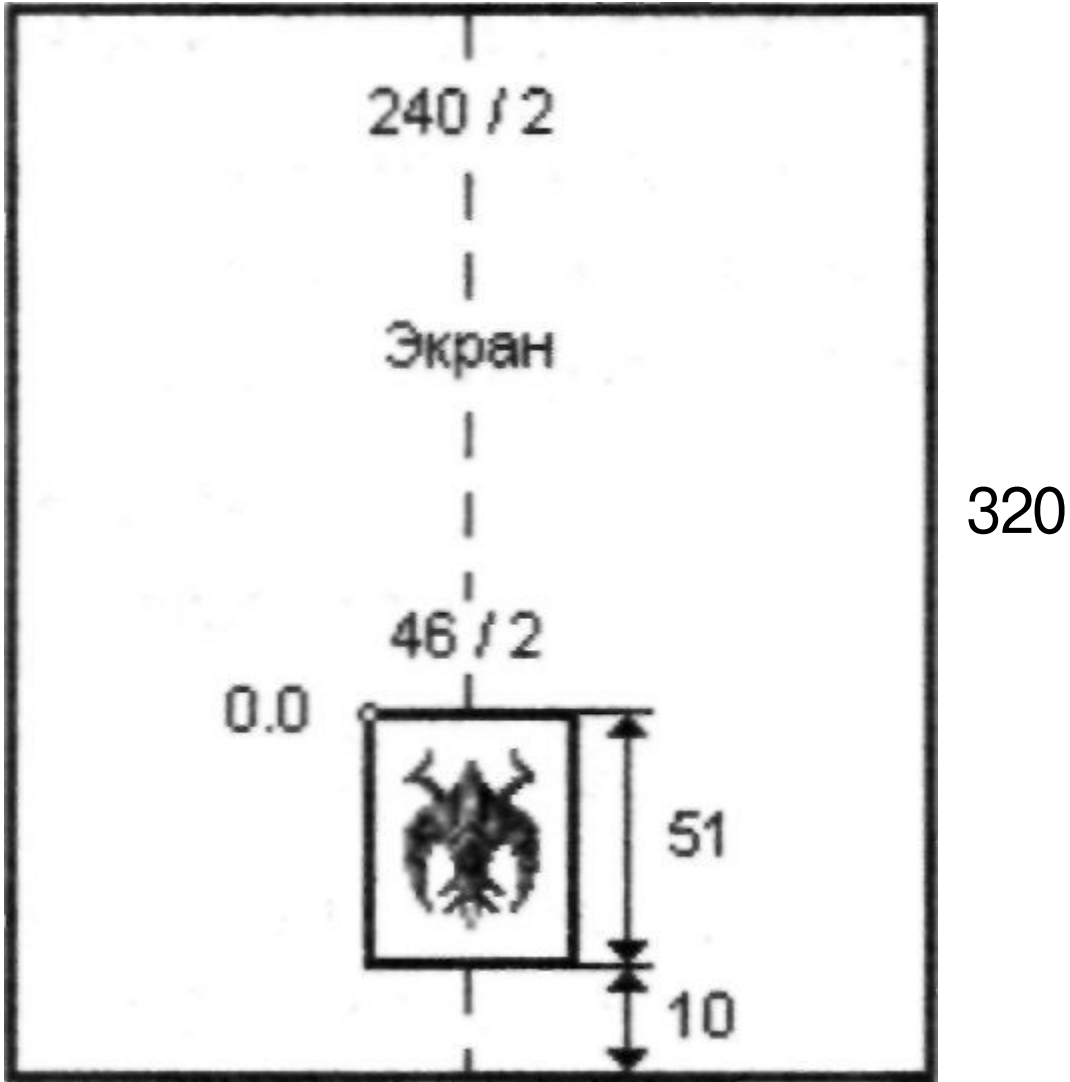
$x = 240 / 2 - 46 / 2$
 $y = 320 - 51 - 10$

Левый верхний угол фрейма изображения Ship.png является началом отсчета. Чтобы установить корабль по ширине экрана телефона точно в центре, необходимо разделить пополам ширину дисплея и ширину одного фрейма (46 пикселей) корабля. По оси Y корабль устанавливается в нижней области экрана, а поскольку начало отсчета находится в левом верхнем углу изображения, то от размера высоты дисплея отнимается также высота изображения (51 пиксель). Еще 10 пикселей по оси Y отнимаются для того, чтобы корабль при инициализации не находился на нижней кромке экрана (рис. 12.8).

Для того чтобы добавить созданный объект в игру и нарисовать корабль на экране телефона, необходимо воспользоваться методом append() класса LayerManager.

```
gameManager.append(ship);
```

Рис. 12.8. Выбор позиции корабля на экране телефона



Заметьте, что добавление корабля в игру происходит раньше, чем произошло добавление фона. Как уже говорилось в предыдущей главе, это важный фактор, и каждый объект, который был добавлен методом `append()` в игру раньше, будет нарисован на экране над предыдущими изображениями. Если поменять местами строки исходного кода, например, следующим образом:

```
gameManager.append(background);
gameManager.append(ship);
```

то сначала произойдет загрузка на экран корабля, а потом - фона игры, и в итоге вы увидите только фон, который закроет собой изображение корабля. Поэтому следите за порядком добавления в игру создаваемых объектов. Правильный вариант для загрузки корабля и фона следующий:

```
gameManager.append(ship);
gameManager.append(background);
```

Обработка событий

И последнее, что нужно сделать, - это создать в методе `updateGame()` класса `MainGameCanvas` конструкцию кода, отвечающую за перемещение корабля по экрану.

```
public void updateGame () {
    //*****
    // обработка событий, получаемых с клавиатуры
    //*****
    int keyStates = getKeyStates();
    // вверх
    if ((keyStates & DOWN_PRESSED) != 0) {
        ship.moveShip(1);
    // вверх
    } else if ((keyStates & UP_PRESSED) != 0) {
        ship.moveShip(2);
    // влево
    } else if ((keyStates & LEFT_PRESSED) != 0) {
        ship.moveShip(3);
    // вправо
    } else if ((keyStates & RIGHT_PRESSED) != 0) {
        ship.moveShip(4);
    // стоп
    } else {
        ship.moveShip(0);
    }
    //*****
    // ДВИЖЕНИЯ
    //*****
    // движение фона
```

```
...  
}
```

Обработка событий, полученных с клавиш телефона, происходит стандартным образом с помощью метода `getKeyStates()`, переменной `keyStates` и констант:

- `DOW_PRESSED` - движение вниз;
- `UP_PRESSED` - движение вверх;
- `LEFT_PRESSED` - движение влево;
- `RIGHT_PRESSED` - движение вправо.

Перечисленные константы назначены для джойстика телефона, а также клавишам под номерами:

- 8 - вниз;
- 2 - вверх;
- 4 - влево;
- 6 - вправо.

Для нажатия клавиш или джойстика телефона назначен вызов метода `moveShip()` с определенным целочисленным значением, соответствующим направлению движения корабля. Для улучшения восприятия исходного кода, отвечающего за обработку событий с клавиш телефона, можно использовать константы, объявленные глобально в исходном коде класса `MainGameCanvas`, например следующим образом:

```
private final STOP = 0;  
private final DOWN = 1;  
private final UP = 2;  
private final LEFT = 3;  
private final RIGHT = 4;
```

В этом случае, например при движении корабля вправо, метод `moveShip()` должен быть вызван следующим образом:

```
ship.moveShip(RIGHT);
```

Вызов метода `updateGame()` в классе `MainGameCanvas` происходит в игровом цикле метода `run()` при постоянном обновлении состояния игры.

```
public void run() {  
    Thread currentThread = Thread.currentThread();  
    try {  
        while (currentThread == animationThread) {  
            long startTime = System.currentTimeMillis();  
            if (isShown()) {  
                updateGame();  
                draw();  
                flushGraphics();  
            }  
        }  
    }  
}
```



```
        long endTime = System.currentTimeMillis() -
            startTime;
        if (endTime < fps) {
            synchronized (this) {
                wait (fps - endTime);
            }
        } else {
            currentThread.yield();
        }
    }
} catch (InterruptedException ie) {
}
}
```

В классе `GameMidlet`, как и в прошлом примере, рассмотренном в *главе 10*, в методе `startApp()` происходят создание и загрузка класса `Splash`, после чего управление игрой передается классу `Loading`, `Menu` и `MainGameCanvas`. Этот механизм работы мы подробно рассматривали в предыдущих главах. Таким образом, в игре «Метеоритный дождь» происходят создание, движение и обработка столкновений корабля на экране телефона. Исходный код текущего проекта находится в папке `\Chapter12\Meteoric_rain_12`.

12.3. Проект Demo

В игре «Метеоритный дождь» фон игры, или карта, по своей ширине соответствует ширине экрана телефона. Если ширина экрана телефона равна 240 пикселям, то и ширина фона так же равна этому значению. Но в некоторых играх фон игры можно сделать шире размера экрана, и тогда пишется дополнительный исходный код, с помощью которого вместе с движением корабля одновременно происходит и передвижение фона игры. На этом механизме построено множество игр, поэтому давайте дополнительно рассмотрим пример `Demo`, где происходит одновременное движение корабля и фона игры.

Алгоритм действий по одновременному передвижению корабля и фона заключается в следующем. Например, пользователь с помощью джойстика или игровых клавиш двигает корабль вправо. При движении корабля вместе с ним с такой же скоростью одновременно передвигается фон игры, но в противоположную сторону. При этом корабль во время движения должен находиться в центре экрана. Как только правая сторона экрана подходит к концу правой стороны фона, движение фона останавливается, но корабль по-прежнему двигается в правую сторону. По достижении конца экрана корабль также останавливается. На рис. 12.9 представлен рассмотренный механизм одновременного движения корабля и фона.

Проект `Demo` находится в папке `\Chapter12\Demo` и содержит исходный код, иллюстрирующий работу приведенного примера. В качестве каркаса программы используется исходный код примера «Метеоритный дождь» с классами `Background.java`, `Ship.java`, `MainGameCanvas.java`, `GameMidlet.java` и `Loading.java`.



Рис. 12.9. Одновременное движение корабля и фона

Механизм работы программы чрезвычайно прост. При запуске программы на экране телефона появляется заставка «Loading» (класс `Loading`). Во время показа заставки происходит создание, инициализация игровых объектов. Далее управление программой передается классу `MainGameCanvas`, где и происходит работа всей программы.

Рассмотрим исходный код всех классов примера **Demo**, за исключением исходного кода класса `Loading`, поскольку структура этого класса ничем не отличается от структуры классов из предыдущих глав.

12.3.1. Класс `Background.java`

```

/*
 * Background.java
 * Фон игры
 */
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class Background extends TiledLayer!

static final int WIDTH = 24;
static final int HEIGHT = 32;
static final int COLUMNS_WIDTH = 20;
static final int ROWS_HEIGHT = 20;

public Background( int columns, int rows,
Image image, int tileWidth, int tileHeight){
    super(columns, rows, image, tileWidth, tileHeight);
    createBackground();
}

public void createBackground(){

```

```

int[ ] mapBackground = {
    ...
};

for(int i = 0; i < mapBackground.length; i++) {
    int column = i % COLUMNS_WIDTH;
    int row =(i - column) / COLUMNS_WIDTH;
    setCell(column, row, mapBackground[i]);
}
}

boolean Area(int x, int y, int width, int height) {
    int rowMin = y / HEIGHT;
    int rowMax = (y + height - 3) / HEIGHT;
    int columnMin = x / WIDTH;
    int columnMax =.(x + width - 3) / WIDTH;

    for (int row = rowMin; row <= rowMax; ++row) {
        for (int column = columnMin; column <= columnMax;
            ++column) {
            int cell = getCell(column, row);
            if ( (cell < 0)) {
                return true;
            }
        }
    }

    return false;
}
}

```

UML-диаграмма класса `Background` изображена на рис. 12.10. В исходный код класса `Background` добавляется новый метод `Area()`, с помощью которого происходит вычисление площади представляемого на экране фона во время движения корабля.

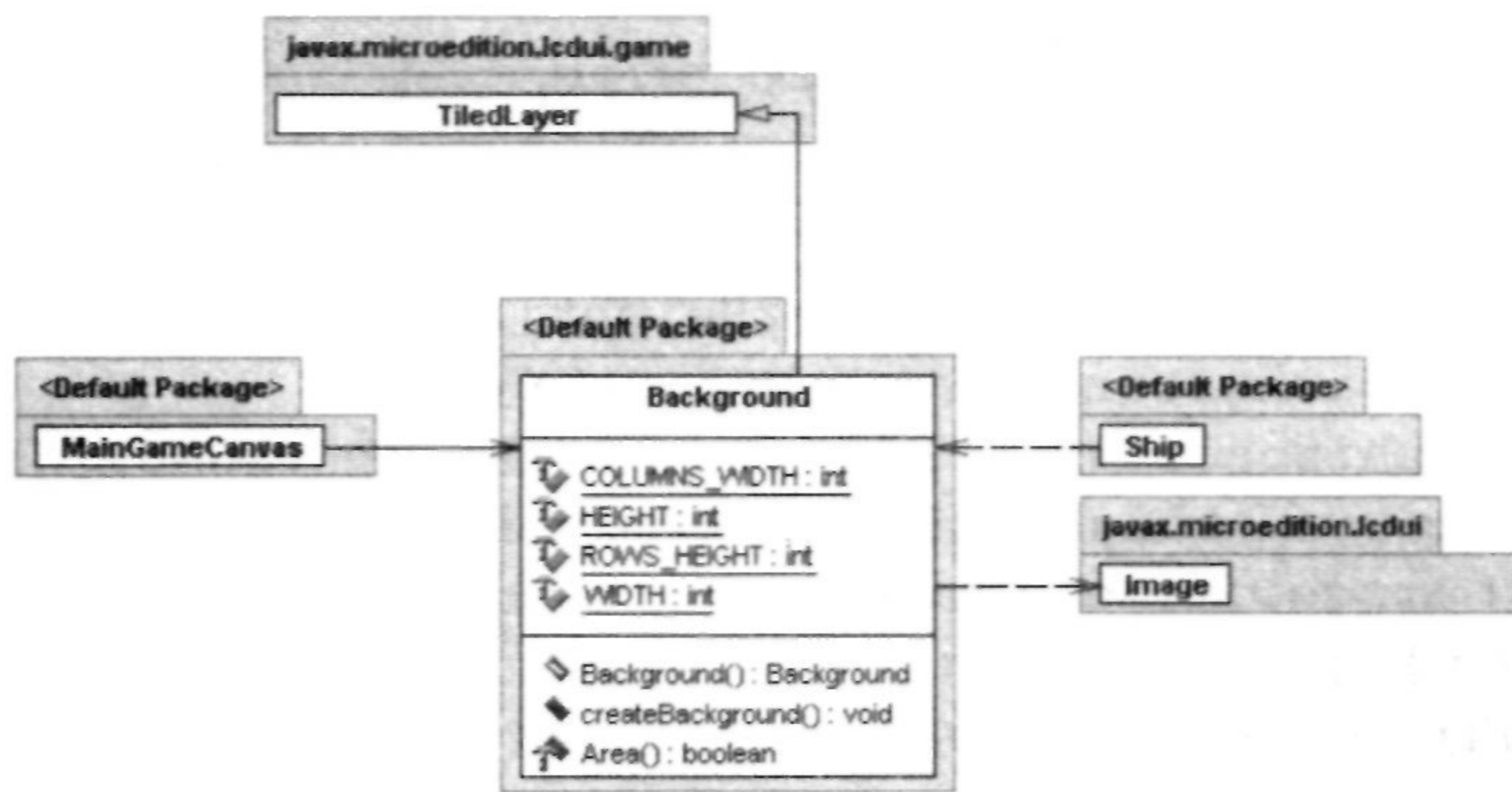


Рис. 12.10. UML-диаграмма класса Background

Также для упрощения общей структуры примера метод `createBackground()` вызывается непосредственно в конструкторе класса `Background`.

При помощи метода `Area()` происходит расчет площади фона, представленного на экране телефона в текущий момент. Метод `Area()` имеет четыре параметра и достаточно прост. Первые два параметра `x` и `y` - это позиция корабля в пространстве (левый верхний угол изображения корабля). Параметры `width` и `height` - соответственно ширина и высота фрейма изображения `Ship.png`. Метод `Area()` мы будем использовать в классе `Ship.java` при движении корабля.

12.3.2. Класс *Ship.java*

```
/*
 * Ship.java
 * Спецификация класса, описывающая создание и движение
 корабля.
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class Ship extends Sprite{

    static final int WIDTH = 34;
    static final int HEIGHT = 39;

    public Ship(Image image, int frameWidth,int frameHeight){
        super(image, frameWidth, frameHeight);
    }

    public void moveShip(int direction, Background
background){
        boolean stop = false;
        switch(direction){
            // вверх
            case 1:
                if ((this.getY() + HEIGHT <
                    background.getHeight()) &&
                    !background.Area(this.getX(),
this.getY() + HEIGHT, WIDTH, 3) && moving(0, 5)){
                    this setFrame(direction);
                    stop = true;
                }
                break;
            // вниз
            case 2 :
```



```
        if ((this.getY() > 0) &&
            !background.Area(this.getX(),
                              this.getY() - 3, WIDTH, 3) && moving(0, -5)){
            this.setFrame(direction);
            stop = true;
        }
    break;
    // влево
    case 3:
        if ((this.getX() > 0) &&
            !background.Area(this.getX() - 3, this.getY(), 3,
                              HEIGHT)
            && moving(-5, 0)){
            this.setFrame(direction);
            stop = true;
        }
    break;
    // вправо
    case 4:
        if ((this.getX() + WIDTH < background.getWidth()) &&
            !background.Area(this.getX() + WIDTH,
                              this.getY(), 3, HEIGHT) && moving(5, 0)){
            this.setFrame(direction);
            stop = true;
        }
    break;
    }
    if (!stop) {
        this.setFrame(0);
        this.move(0, 0);
    }
}

private boolean moving(int xSpeed, int ySpeed){
    move(xSpeed, ySpeed);
    return true;
}
}
```

В классе `Ship` был изменен метод `move Ship()` - добавлен параметр `background` и несколько модернизирован механизм движения корабля, а также создан новый метод `moving()`. UML-диаграмма класса `Ship` представлена на рис. 12.11.

Как видно из исходного кода класса `Ship`, при движении корабля в любую из сторон происходит сравнение текущей позиции с окончанием экрана и фона игры. При перемещении корабля в одну из сторон используется метод `setFrame()` для установки направления движения корабля соответствующего фрейма изображения.

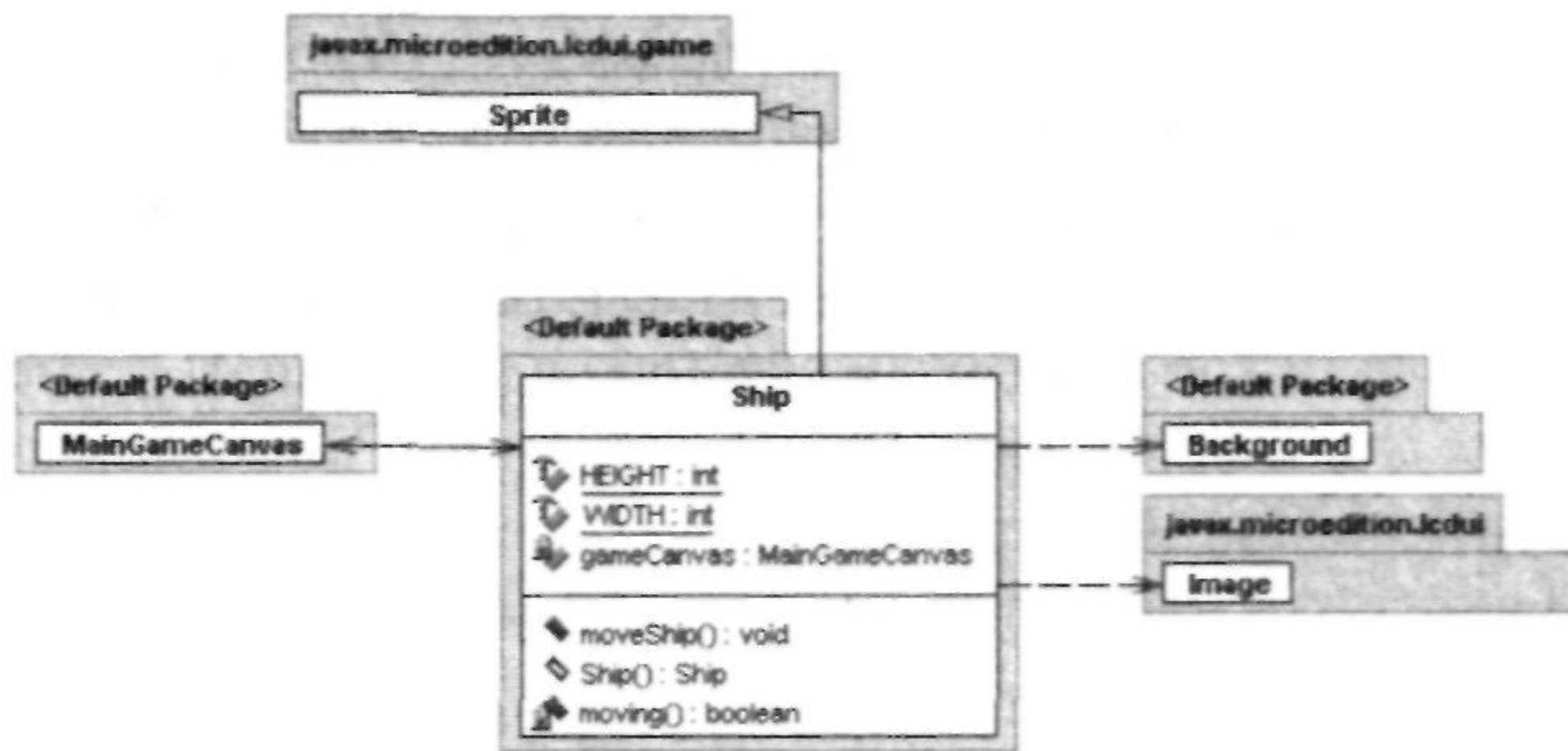


Рис. 12.11. UML-диаграмма класса Ship

В методе `moving()` задается скорость движения корабля на экране. В качестве скорости корабля специально выбрано значение в 5 пикселей, для того чтобы можно было внимательно рассмотреть фазы движения корабля и фона.

12.3.3. Класс *MainGameCanvas.java*

```

/*
 * MainGameCanvas.java
 * Класс MainGameCanvas создает и рисует игровые объекты
 * Обрабатывает нажатие с клавиш телефона
 */
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class MainGameCanvas extends GameCanvas implements
Runnable {
    // midlet
    private GameMidlet midlet = null;
    // graphic context
    private Graphics graphics = null;
    // thread
    private volatile Thread animationThread = null;
    // менеджер слоев
    private LayerManager gameManager = null;
    // карта
    private Background background = null;
    // корабль
    private Ship ship = null;
    // fps
    private int fps = 50;
    // ширина
    int screenWidth = 0;
    // высота

```

```
int screenHeight = 0 ;
// направление
private final int STOP = 0;
private final int DOWN = 1 ;
private final int UP = 2;
private final int LEFT = 3;
private final int RIGHT = 4;

public MainGameCanvas(GameMidlet midlet) throws Exception
{
    // constructor super class
    super(true);
    // midlet
    this.midlet = midlet;
    // полноэкранный режим
    setFullScreenMode(true);
    // graphic context
    graphics = getGraphics();
    // ширина
    screenWidth = this.getWidth();
    // высота
    screenHeight = this.getHeight();
    // менеджер слоев
    gameManager = new LayerManager() ;
    // игровые компоненты
    createGame();
    //*****

    // добавляем компоненты в игру
    //*****

    // корабль
    gameManager.append(ship);
    // карта
    gameManager.append(background);
}

// создание и запуск потока
public void start () {...}
// игровой цикл
public void run () {...}
// остановка потока
public void stop() {...}
// обработка soft key
public void keyPressed (int keyCode) {...}
// рисование игры на экране
```

```
private void draw() {
    graphics.setColor(0, 0, 0);
    graphics.fillRect(0, 0, screenWidth, screenHeight);
    int x = origin(ship.getX() + ship.WIDTH / 2,
background.getWidth(), screenWidth);
    int y = origin(ship.getY() + ship.HEIGHT / 2,
background.getHeight(), screenHeight);
    graphics.setClip(x, y, background.getWidth(),
background.getHeight());
    graphics.translate(x, y);
    gameManager.paint(graphics, 0, 0);
    graphics.translate(-x, -y);
    graphics.setClip(0, 0, screenWidth, screenHeight);
}
// создание объектов игры
public void createGame() throws Exception {
    /*******
    // карта
    /*******
    try{
        Image imageBackground = Image.createImage("/
        Background.png");
        background = new
            Background(Background.COLUMNS_WIDTH,
Background.ROWS_HEIGHT, imageBackground,
        Background.WIDTH,
Background.HEIGHT);
    }catch (Exception ex) {
        System.err.println("Background it is not loaded");
    }
    /*******
    // корабль

    try{
        Image imageShip = Image.createImage("/Ship.png");
        ship = new Ship(this, imageShip, Ship.WIDTH,
            Ship.HEIGHT);
        ship.setPosition(background.getWidth 0/2 -
            Ship.WIDTH/2,
            background.getHeight()/2 -Ship.HEIGHT/2);
        imageShip = null;
    }catch (Exception ex) {
        System.err.println("Ship.png it is not loaded");
    }
}
```



```
}
// update game
public void updateGame() {
    /*******
    // клавиши
    /*******

    int keyStates = getKeyStates();
    int direction = (keyStates == DOWN_PRESSED) ? DOWN :
        (keyStates == UP_PRESSED) ? UP :
        (keyStates == LEFT_PRESSED) ? LEFT :
        (keyStates == RIGHT_PRESSED) ? RIGHT : STOP;
        ship.moveShip(direction, background);
}
// определяет точку отсчета
private int origin(int focus, int fieldLength, int
screenLength) {
    int origin;
    if (screenLength >= fieldLength) {
        origin = (screenLength - fieldLength) / 2;
    }
    else if (focus <= screenLength / 2) {
        origin = 0 ;
    }
    else if (focus >= (fieldLength - screenLength / 2)) {
        origin = screenLength - fieldLength;
    }
    else {
        origin = screenLength / 2 - focus;
    }
    return origin;
}
}
```

UML-диаграмма класса `MainGameCanvas` изображена на рис. 12.12. В начале исходного кода `MainGameCanvas` добавляются пять констант для каждой фазы движения корабля.

```
// направление
private final int STOP = 0;
private final int DOWN = 1;
private final int UP = 2;
private final int LEFT = 3;
private final int RIGHT = 4;
```

Здесь можно было использовать и числовые значения, как это мы делаем в игре «Метеоритный дождь», но хочется показать различные варианты исходного кода.

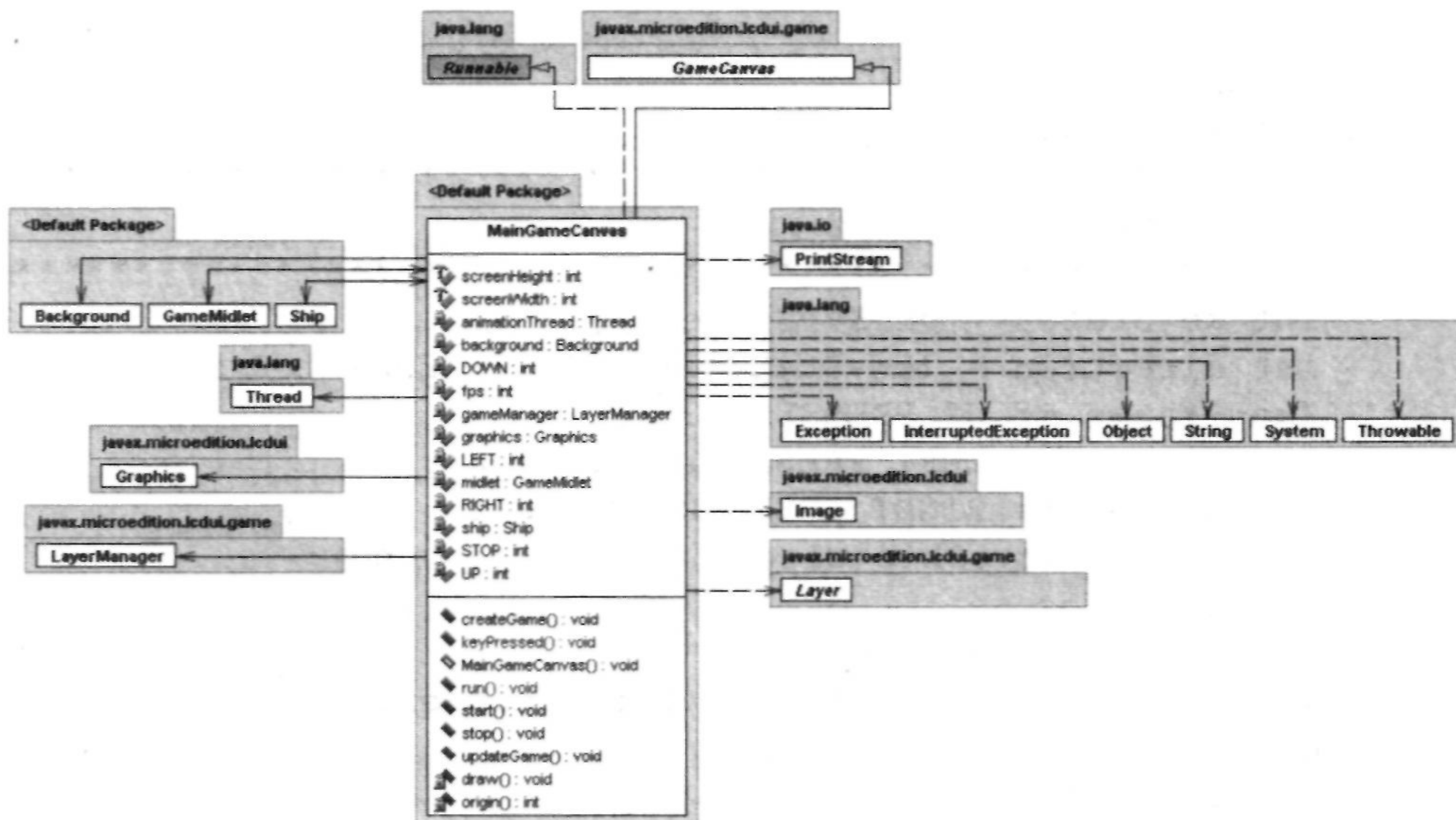


Рис. 12.12. UML-диаграмма класса MainGameCanvas

В класс MainGameCanvas добавлен новый метод `origin()`, с помощью которого определяются точки отсчета при рисовании уровня. Метод `origin()` имеет три параметра. Первый параметр - это точка в центре текущего фрейма изображения Ship.png. Второй параметр `fieldLength` содержит значение ширины или высоты экрана, и третий параметр `screenLength` содержит значение уже ширины или высоты дисплея телефона. Вызов метода `origin()` происходит в классе MainGameCanvas в методе `draw()` при определении координат для отсечения плоскости.

И последнее изменение коснулось метода `updateGame()`, где происходит изменение конструкции исходного кода для движения корабля по экрану телефона. Посмотрите, как элегантно решена задача по обработке нажатий джойстика или игровых клавиш телефона. Это еще один из множества вариантов обработки событий с клавиш телефона, получаемых от пользователя. В этой конструкции кода используется оператор `?` : для обработки событий, получаемых с клавиш телефона.

12.3.4. Класс GameMidlet.java

```
/*
 * GameMidlet.java
 * MIDlet game
 */

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class GameMidlet extends MIDlet {
```

```
// game canvas
private MainGameCanvas gameCanvas = null;

// конструктор
public GameMidlet() {
}
// старт
public void startApp() {
    Display.getDisplay(this).setCurrent(new Loading(this));
}
// пауза
public void pauseApp() {
}
// удаление
public void destroyApp(boolean unconditional) {
    if(gameCanvas!= null){
        gameCanvas.stop();
    }
}
// новая игра
public void newGame() throws Exception{
    try{
        gameCanvas = new MainGameCanvas(this);
        gameCanvas.start();
    } catch(Exception ioe) {
        System.out.println(ioe);
    }
    Display.getDisplay(this).setCurrent(gameCanvas);
}
// выход
public void exitGame() {
    destroyApp(false);
    notifyDestroyed();
}
}
```

Класс `GameMidlet` несколько видоизменен, в частности не используется заставка при загрузке всей игры и напрямую происходит создание класса `Loading`. После чего управление игрой передается классу `MainGameCanvas`. UML-диаграмма класса `GameMidlet` показана на рис. 12.13.

Откомпилируйте этот пример, запустите на эмуляторе или установите на телефон. Проанализируйте, как работает программа, внимательно изучите и поэкспериментируйте с исходным кодом проекта `Demo`.

В этой главе мы изучили класс `Sprite` из пакета `java.microedition.lcdui.game` профиля MIDP 2.0. Как вы убедились,

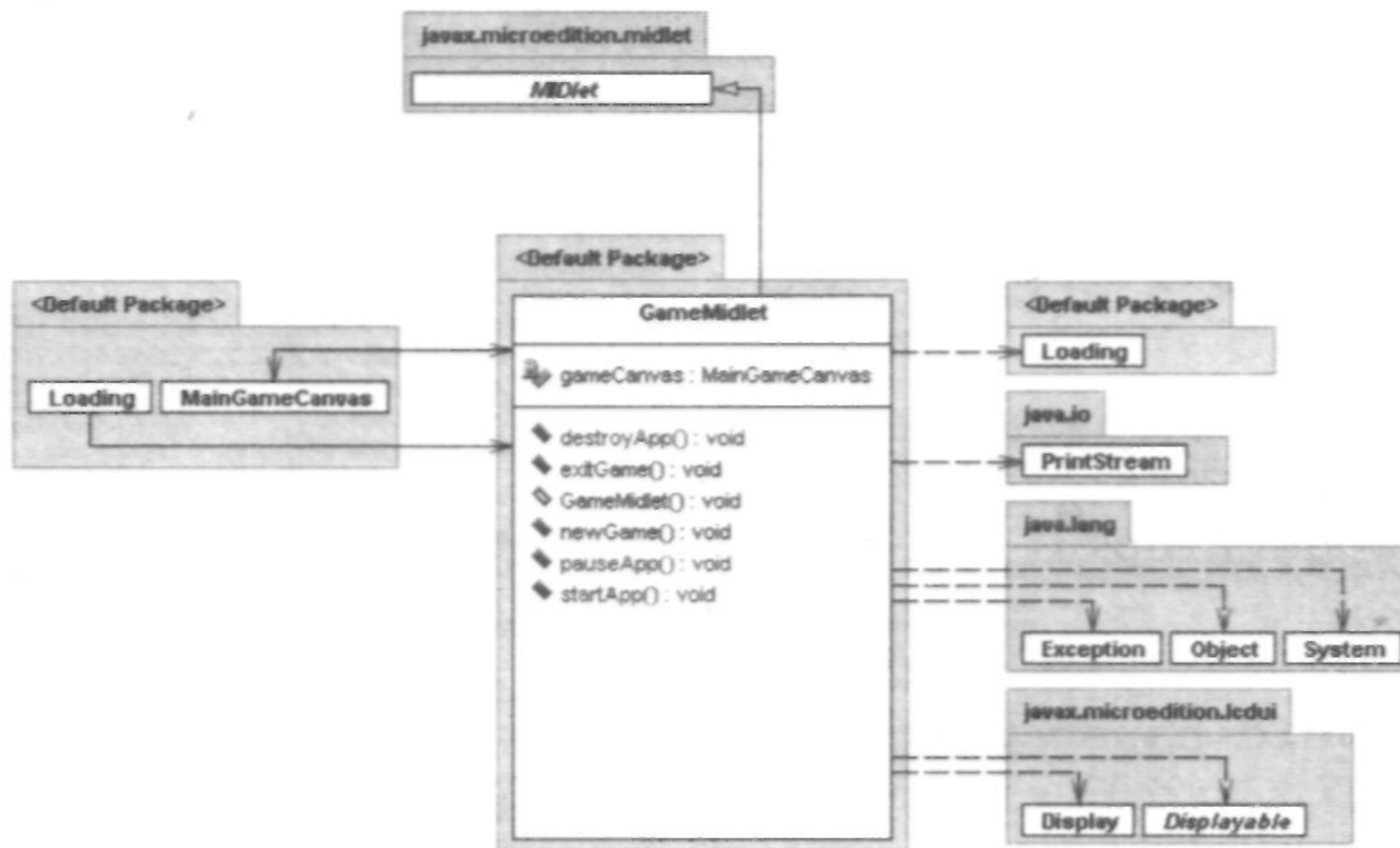


Рис. 12.13. UML-диаграмма класса GameMidlet

функциональные возможности класса `Sprite` достаточно мощные для решения различных задач. В программировании игр для мобильных телефонов на Java 2 ME класс `Sprite` играет очень важную роль, и фактически любой игровой объект представлен этим классом. Изучая эту главу, мы нарисовали корабль, состоящий из пяти фреймов анимационной последовательности для каждой фазы движения. Сконструировали и создали класс `Ship`, описали его спецификацию и создали объект `ship` в классе `MainGameCanvas`. В конце главы рассмотрели дополнительный демонстрационный пример `Demo`, где был показан механизм одновременного передвижения корабля и фона. Эта глава содержит очень много полезной информации и поможет вам при создании своих собственных проектов. В следующей главе мы продолжим модифицировать игру «Метеоритный дождь».

<http://palata-x.narod.ru>

Глава 13. Основы искусственного интеллекта

Большинство компьютерных и мобильных игр построены на различных конфликтах, возникающих в процессе игры между двумя, тремя и более игроками, где за одну из сторон может выступать *искусственный интеллект* самой игры. Поведение игровых персонажей в игре обусловлено множеством различных факторов или событий, которые могут изменяться каждую секунду. В этом ракурсе задача программиста игровой логики заключается в создании объектов, умеющих адекватно реагировать на различные игровые события.

Особой необходимости писать в мобильных играх системы искусственного интеллекта, основанные на нейронных сетях, нечеткой логике или генетических алгоритмах человеческого мозга, нет, все это применимо в других местах. Вместо этого в играх для мобильных телефонов используется набор различных алгоритмов, позволяющих создавать вполне пригодный, а главное, разумный игровой интеллект. Имитация разумности персонажей вашей игры - это и есть основная и главная цель, которую вам предстоит решать по мере создания игры.

Искусственный интеллект - это очень сложная и широко освещаемая тема многих изданий, например у издательства «ДМК-Пресс» есть хорошая книга по этой теме с названием «Программирование искусственного интеллекта в приложениях». Мы же в этой главе отойдем от создания игры «Метеоритный дождь» и рассмотрим основные приемы, используемые при формировании игровой логики. Для того я создал несколько характерных примеров, где используются детерминированные и шаблонные алгоритмы искусственного интеллекта.

Примеры:

- **Движение в заданном направлении** - одним из простейших способов движения объекта является его перемещение с определенной скоростью в заданном направлении. В этом разделе мы поговорим о простых игровых алгоритмах, необходимых для реализации движения объекта в заданном направлении.
- **Движение за объектом** - этот раздел содержит несколько расширенную информацию по перемещению объектов в пространстве, что позволит вам создавать более разумных персонажей игры, которые будут неотступно преследовать игрока на протяжении всего игрового процесса или заданного промежутка времени.
- **Движение от объекта** - не всегда нужно неотступно следовать за игроком, иногда следует убежать, спрятаться и накопить силы для дальнейшей атаки. Поэтому в этом разделе мы рассмотрим пример исходного кода, с помощью которого можно убегать от своего преследователя по мере его приближения.

- **Случайное или хаотичное движение** - одним из интересных видов движения персонажей в игре является их независимое и случайное перемещение. Этот раздел познакомит вас с реализацией движения объекта в случайном направлении и случайной скоростью.
- **Простые шаблоны** - очевидно, что игровые шаблоны включают в себя уже более высокий уровень игрового интеллекта, поэтому этот раздел знакомит вас с парадигмой шаблонного программирования игровой логики, а представленный пример исходного кода покажет всю элегантность решения задач на базе шаблонов.
- **Шаблоны с обработкой событий** - шаблоны с возможностью обработки игровых событий - это одно из мощных средств создания игровой логики. В этом разделе мы изучим демонстрационный пример, реализующий шаблон с обработкой событий, которые возникают во время игры.
- **Смена состояний игровых объектов** - при рассмотрении этого вопроса мы на конкретном примере смоделируем смену состояний игрового объекта на основе распределенной логики и счетчика прошедших игровых циклов.

13.1. Структура классов

в демонстрационных примерах

Для всех демонстрационных примеров этой главы используется одна и та же структура классов, но в каждом проекте изменяется исходный код метода `moveBall()` класса `Boll`. Метод `moveBall()` представляет тот или иной образец игровой логики, задекларированный нами в названии этой главы.

На рис. 13.1 графически представлен принцип работы программ. Полный листинг исходных кодов всех примеров вы найдете на компакт-диске в папке `Code\Chapter13\AI1-AI8`. Но прежде чем перейти к изучению примеров, я предлагаю в нескольких разделах кратко разобрать общую структуру классов демонстрационных примеров, чтобы всем было понятно, как работает программа.

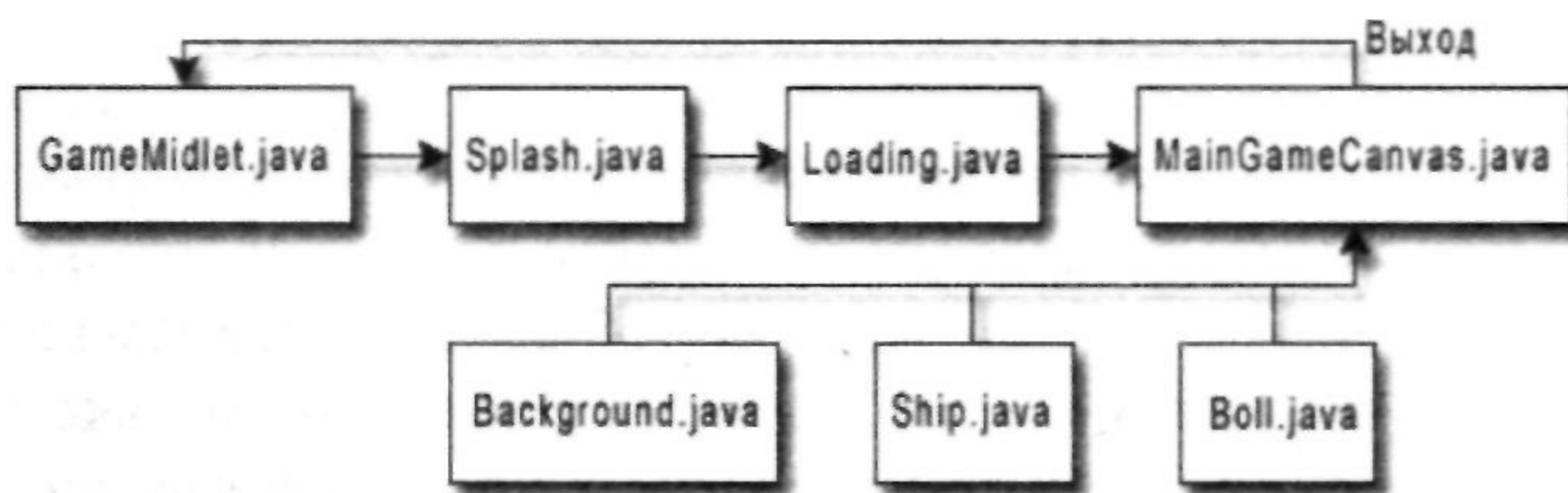


Рис. 13.1. Модель работы демонстрационных примеров

13.1.1. Класс *GameMidlet*

```

/*
 * GameMidlet.java
 * Основной класс мидлета

```

```
*/

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

/**
 * @author Stanislav Gornakov
 * @version 1.0
 */

public class GameMidlet extends MIDlet {
    // объект класса MainGameCanvas
    private MainGameCanvas gameCanvas = null;
    // объект класса loading
    private Loading loading = null;
    // конструктор
    public GameMidlet() {...}
    // старт программы
    public void startApp() {...}
    // пауза
    public void pauseApp() {...}
    // выгрузка игры из памяти устройства
    public void destroyApp(boolean unconditional) {
        garbageCollection();
    }
    // инициализация игровых объектов
    public synchronized void initializationGame() {...}
    // загрузка игры
    public void loadingGame() {...}
    // запуск игры
    public void newGame() {...}
    // сборка мусора
    private void garbageCollection() {...}
    // выход
    public void exitGame() {...}
}
```

После установки программы на телефон пользователь запускает программу, и первым в работу вступает класс `GameMidlet`, UML-диаграмма этого класса представлена на рис. 13.2.

В конструкторе класса `GameMidlet` происходит объявление двух объектов классов `Loading` и `MainGameCanvas`. Эти два объекта необходимы для инициализации и запуска программы. В методе `startApp()` происходит создание класса `Splash`, и на экране телефона рисуется первая игровая заставка.

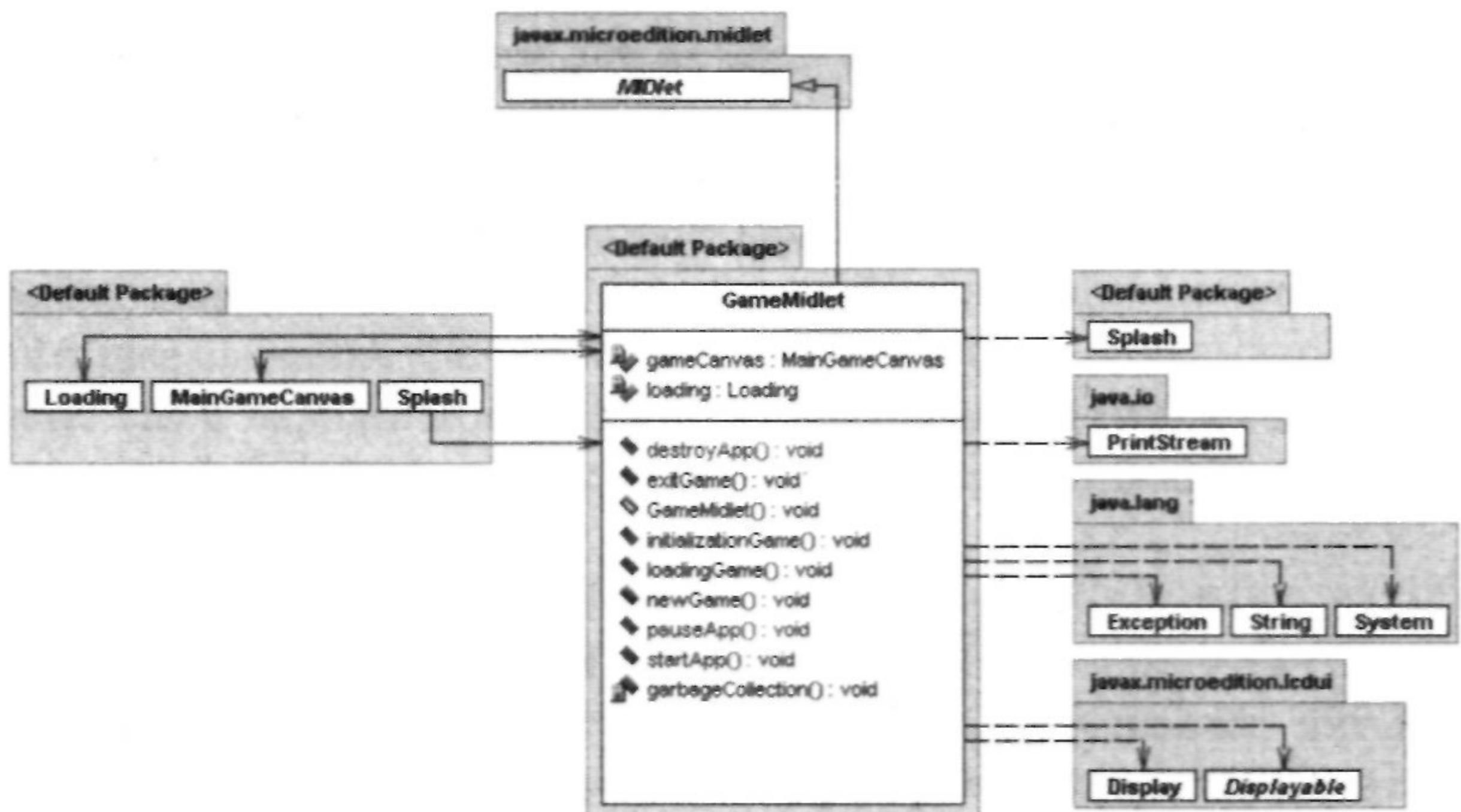


Рис. 13.2. UML-диаграмма класса GameMidlet

В момент показа заставки происходит вызов метода `initializationGame()`, в котором создаются объект `gameCanvas` класса `MainGameCanvas`, объект `loading` класса `Loading`, а также следует вызов метода `loadingGame()`.

Метод `loadingGame()` рисует на экране графическую заставку «Loading», которая скрывает на некоторое время текущие манипуляции с игровыми объектами. В момент показа заставки все игровые объекты устанавливаются на свои позиции посредством вызова метода `newGame()`, после чего начинается работа класса `MainGameCanvas`.

Методы `destroyApp()`, `garbageCollection()` и `cgarbageCollection()` класса `GameMidlet` выполняют полный комплекс услуг по очистке памяти телефона от игровых объектов и выходу из запущенной программы.

13.1.2. Класс Splash

```

/*
 * Splash.java
 * Заставка программы
 */

```

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

```

 * @author Stanislav Gornakov
 * Aversion 1.0
 */

```

```

class Splash extends Canvas implements Runnable {
    private GameMidlet midlet = null;
    private Image imageSplash = null;

```



```

import javax.microedition.lcdui.*;

/**
 * @author Stanislav Gornakov
 * Aversion 1.0
 */

class Loading extends Canvas implements Runnable {
    private GameMidlet midlet = null;
    private volatile Thread thread = null;
    private Image image = null;
    // конструктор класса
    Loading (GameMidlet midlet) {...}
    // запускает поток
    public void start () {...}
    // игровой цикл
    public void run () {...}
    // останавливает системный поток
    public void stop () {...}
    // рисуем на экране
    public void paint (Graphics graphics) {...}
}

```

Класс `Loading` необходим для загрузки и запуска игрового процесса всей программы, на рис. 13.4 изображена UML-диаграмма этого класса. Когда управление программой переходит к этому классу, на экране появляется заставка «Loading», которая будет показываться в течение двух секунд. За время показа заставки происходят загрузка программы и запуск программы с помощью вызова метода `newGame()` класса `GameMidlet`.

В методе `newGame()` происходит запуск работы приложения, где управление работой программы переходит к классу `MainGameCanvas`. Как и в предыдущем классе `Splash`, если для загрузки игровых объектов программе понадобится больше двух секунд, то это время будет выделено системой.

13.1. 4. Класс *Background*

```

/*
 * Background.java
 * Игровая карта
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**

```

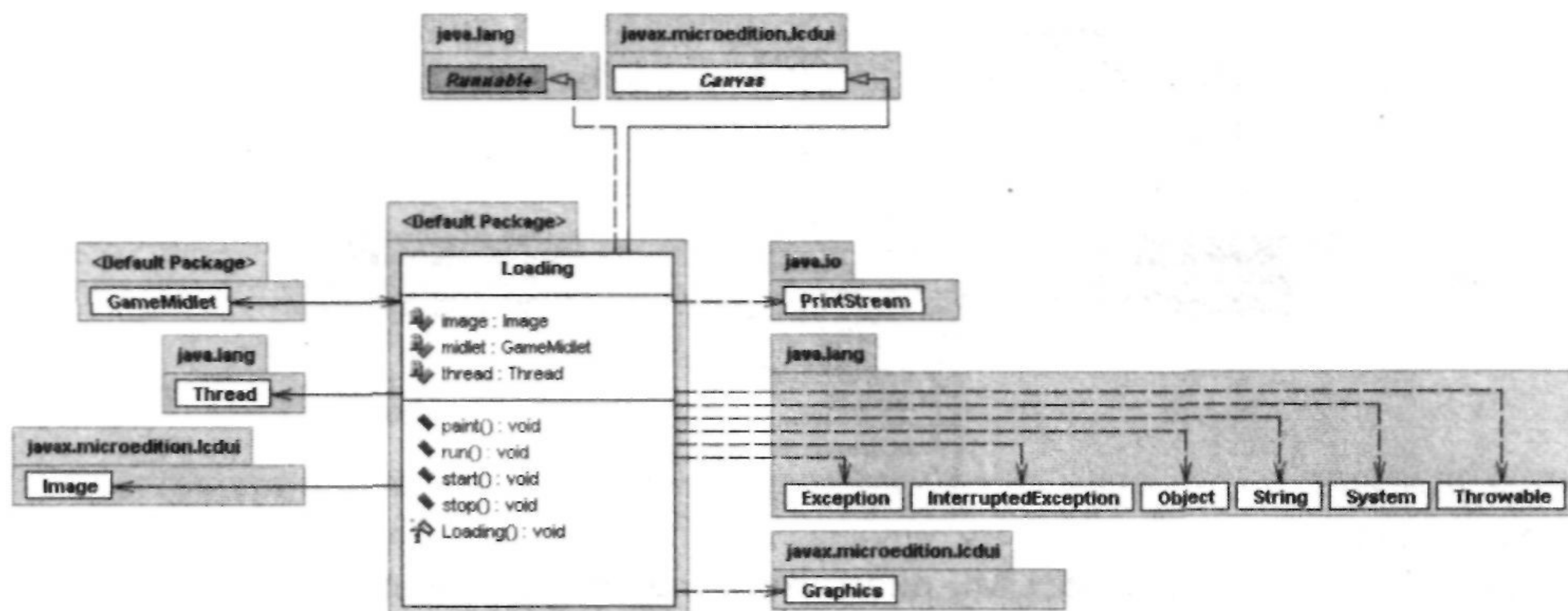


Рис. 13.4. UML-диаграмма класса Loading

```

/*
 * @author Stanislav Gornakov
 * (Aversion 1.0
 */

public class Background extends TiledLayer {
    static final int WIDTH = 24;
    static final int HEIGHT = 32;
    static final int COLUMNS_WIDTH = 10;
    static final int ROWS_HEIGHT = 20;
    // конструктор
    public Background(int columns, int rows, Image image, int
    tileWidth, int tileHeight) {
        super(columns, rows, image, tileWidth, tileHeight);
    }
    // создаем карту
    public void createBackground () {...}
    // движение карты в игре
    public void moveBackground (int screenHeight){...}
}

```

Класс Background представляет собой фоновый рисунок, или игровую карту, которая загружается на заднем плане игры. UML-диаграмма класса Background изображена на рис. 13.5. Механизм загрузки карты стандартен и идентичен механизму, применяемому в игре «Метеоритный дождь». Единственное, о чем следует упомянуть, - это о размерах игровой карты.

Карта создана специально под дисплей телефона с физическим размером 240 x 320 пикселей. В фоновом изображении используются ячейки размером 24 x 32 пикселей, а сама карта состоит из 10 столбцов и 20 строк. При этом 10 столбцов по 24 пикселя - это как раз и есть размер экрана в 240 пикселей, а вот 20 строк по 32 пикселя уже равны 640 пикселям, то есть получается, что это два вертикальных

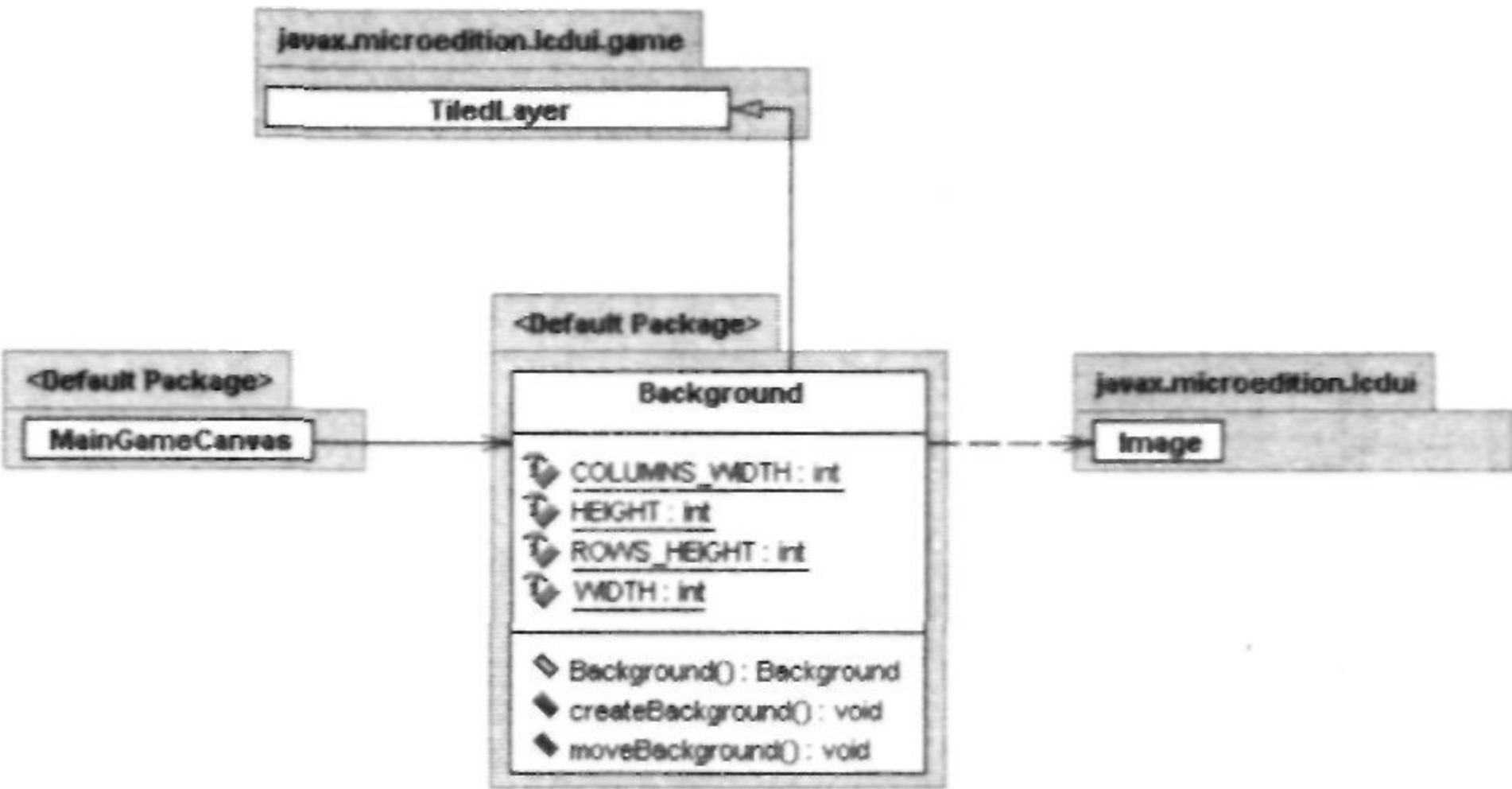


Рис. 13.5. UML-диаграмма класса Background

физических размера экрана телефона размером в 320 пикселей. Сделано это намеренно, чтобы показать вам возможную схему зацикленного перемещения карты в игре. Смысл этой схемы чрезвычайно прост.

Создается игровая карта, равная двум вертикальным размерам экрана телефона (320 пикселей), и устанавливается в позицию -320 пикселей по оси Y (относительно экрана телефона), как показано на рис. 13.6. Затем начинается перемещение игровой карты вниз (см. рис. 13.6), в нашем случае для этого создан метод moveBackground() класса Background, который передвигает карту по оси Y со скоростью в один пиксель. Как только начальные координаты карты совпадают с начальными координатами экрана телефона, то игровой фон опять устанавливается в позицию по оси Y -320 пикселей, что организует механизм циклического движения карты в игре.

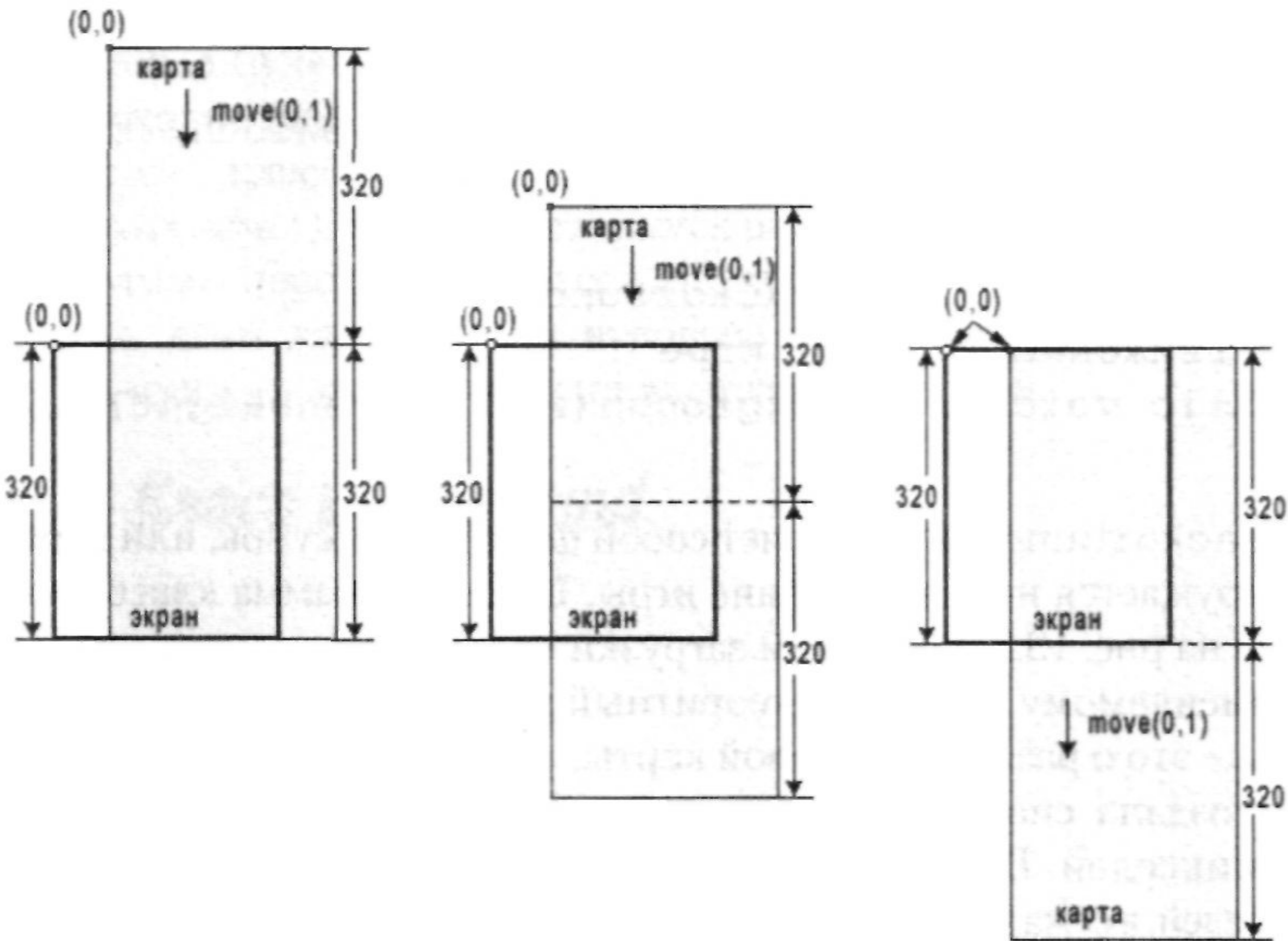


Рис. 13.6. Установка и перемещение игрового фона

Механизм циклического перемещения карт очень часто используется в различных играх, правда, при этом присутствует скучная однообразность игры, заключенная в использовании одного и того же фонового рисунка. Для улучшения игры в данном случае можно попробовать загружать разные карты для игровых уровней, но, конечно, ничто не может сравниться с полноценной игровой картой, сделанной специально под каждый уровень.

13.1.5. Класс Ship

В рассматриваемых демонстрационных примерах класс `Ship` ничем не отличается от аналогичного класса `Ship`, используемого в предыдущей главе и во всей игре в частности, поэтому нет особого смысла подробно рассматривать этот класс. В *главе 12* мы подробно изучали данный класс. Напомню, что исходные коды всех примеров вы найдете на компакт-диске в папке **Code\Chapter13\AI-AI8**. Подробная UML-диаграмма класса `Ship` представлена на рис. 13.7.

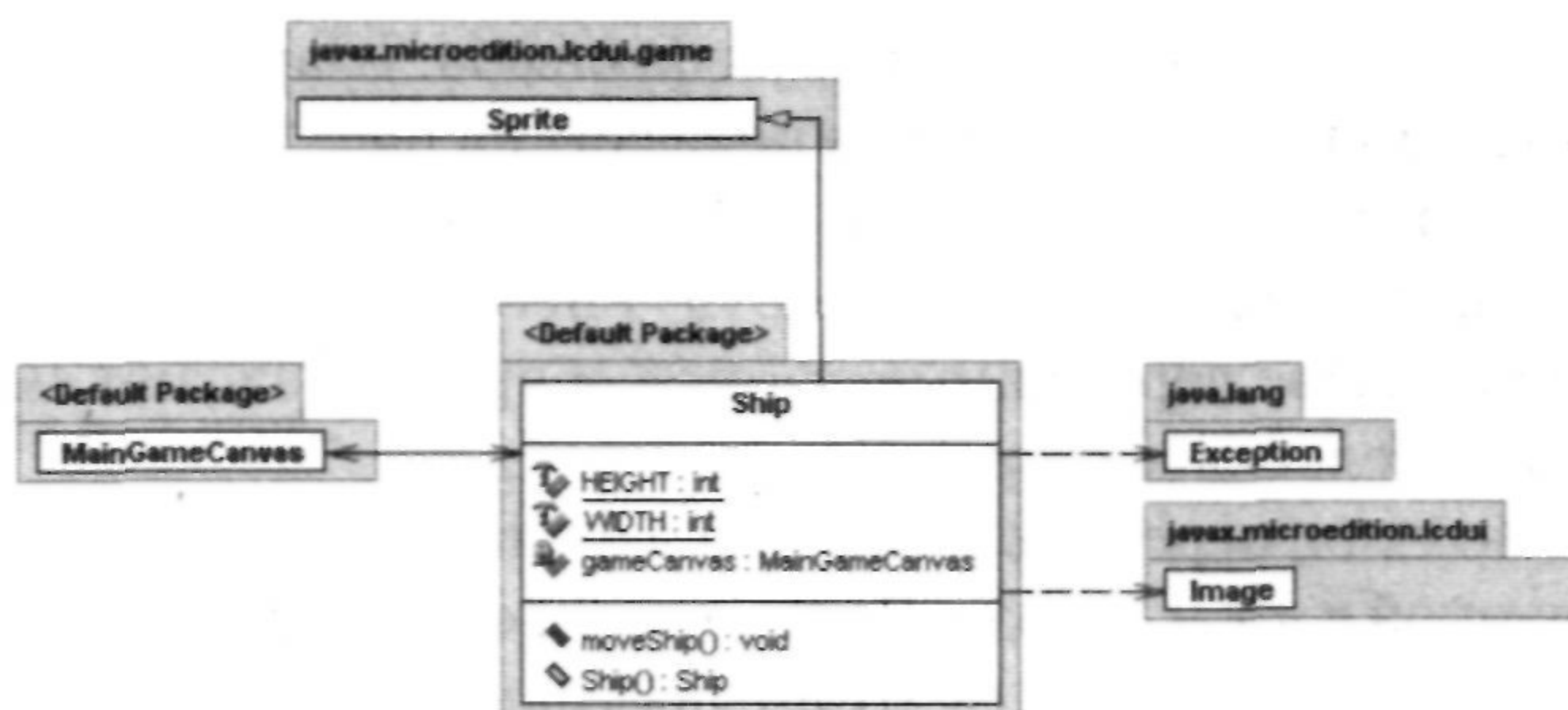


Рис. 13.7. UML-диаграмма класса Ship

13.1.6. Класс *Boll*

```

/*
 * Boll.java
 * Игровой объект
 * Реализует алгоритмы игрового интеллекта
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**
 * @author Stanislav Gornakov
 * Aversion 1.0
 */

public class Boll extends Sprite{
    // gameCanvas

```

```
MainGameCanvas gameCanvas = null;
// ширина, высота и скорость
int WIDTH = 0;
int HEIGHT = 0;
int speedX = 0 ;
int speedY = 5;
// конструктор класса
public Boll(MainGameCanvas gameCanvas, Image image, int
frameWidth, int frameHeight) {...}
// логика движения объекта в игре
public void moveBoll () {...}
// выводит информационную надпись
public void drawAIBoll (Graphics graphics) (...)
}
```

В демонстрационных примерах в качестве объекта используется анимированное изображение шара, состоящее из двух фреймов. Шар создается с помощью класса `Boll`, UML-диаграмма которого показана на рис. 13.8.

Переменные этого класса `WIDTH` и `HEIGHT` содержат ширину и высоту одного фрейма изображения шара. Эти размеры будут необходимы нам для определения столкновений в игре. Две переменные `speedX` и `speedY` задают скорость движения шара на экране телефона за один игровой цикл.

Метод `moveBoll()` перемещает шар на экране в соответствии с игровой логикой, реализованной в этом методе. В каждом новом демонстрационном примере исходный код метода `moveBoll()` будет постоянно изменяться. Метод класса `drawAIBoll()` предназначен для рисования на экране названия запущенных проектов и другой отладочной и технической информации.

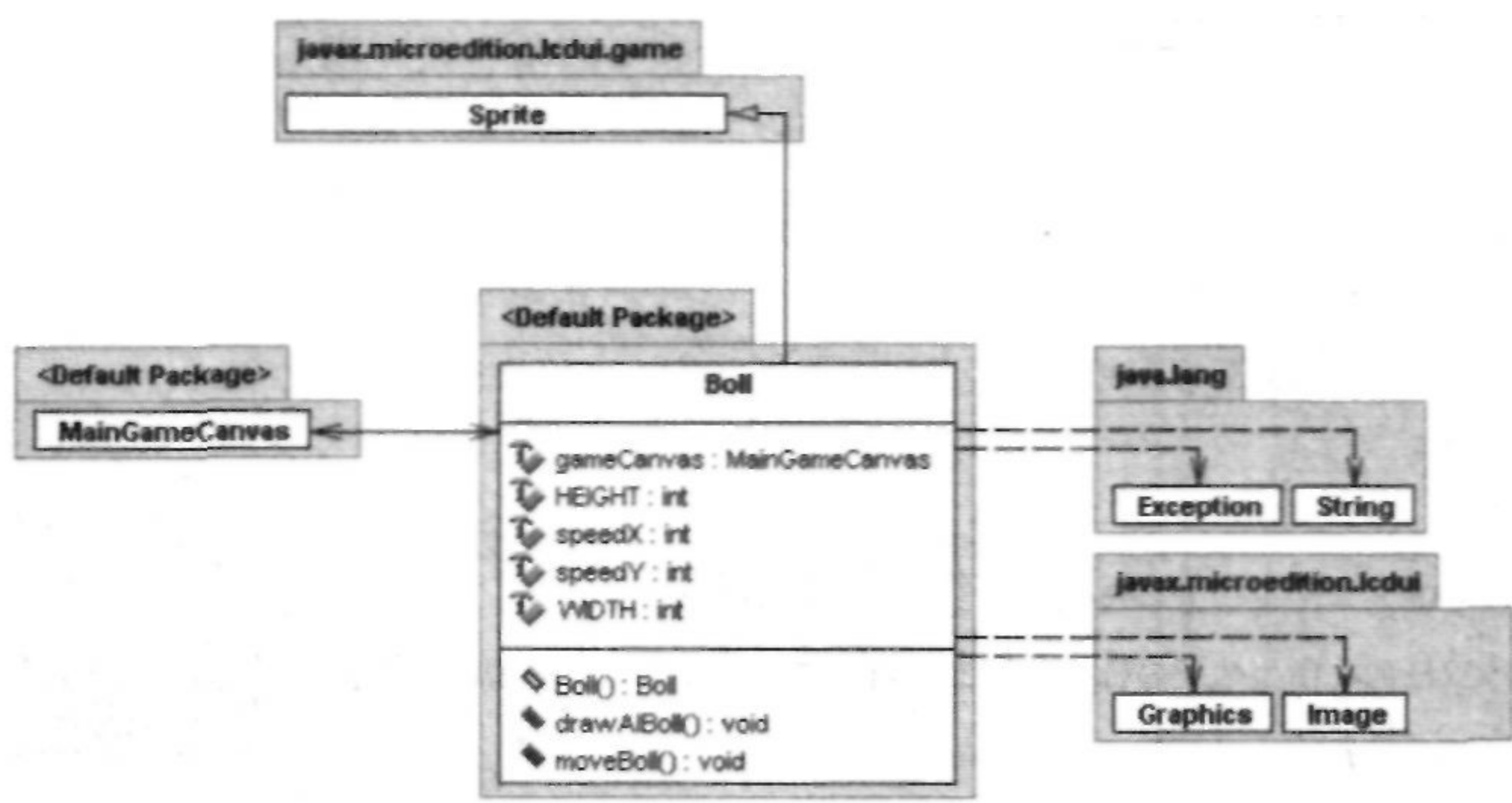


Рис. 13.8. UML-диаграмма класса Boll

13.1.7. Класс MainGameCanvas

```
/*
 * MainGameCanvas.java
 * Игровой класс
```

```
*/

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**
 * @author Stanislav Gornakov
 * (Aversion 1.0
 */

public class MainGameCanvas extends GameCanvas implements
Runnable {
    // мидлет
    private GameMidlet midlet = null;
    // графический контекст
    private Graphics graphics = null;
    // системный поток
    private volatile Thread animationThread = null;
    // менеджер уровней
    private LayerManager gameManager = null;
    // игровая карта
    private Background background = null;
    // корабль
    private Ship ship = null;
    // объект класса Boll
    private Boll boll = null;
    // fps
    private int fps = 50;
    // ширина экрана
    int screenWidth = 0;
    // высота экрана
    int screenHeight = 0;

    public MainGameCanvas(GameMidlet midlet) throws Exception
    {
        // конструктор суперкласса
        super(true);
        // мидлет
        this.midlet = midlet;
        // полноэкранный режим
        setFullScreenMode(true);
        // графический контекст
        graphics = getGraphics();
        // ширина экрана
```

```

screenWidth = this.getWidth();
// высота экрана
screenHeight = this.getHeight();
// создаем менеджер уровней
gameManager = new LayerManager();
// создаем игру
createGame();
}

// запускаем системный поток
public void start(){...}
// игровой цикл
public void run(){...}
// остановка потока
public void stop(){...}
// обработка подэкранных клавиш
public void keyPressed (int keyCode) {...}
// рисуем на экране
private void draw() {...}
// создаем игровые объекты
public void createGame () throws Exception{...}
// установка объектов на позиции
public void setGame () {...}

// обновление игры
public void updateGame(){
    /*******
    // обработка пользовательского ввода
    /*******

    int keyStates = getKeyStates();
    if((keyStates & DOWN_PRESSED) != 0){
        ship.moveShip(1);
    }else if((keyStates & UP_PRESSED) != 0){
        ship.moveShip(2);
    }else if((keyStates & LEFT_PRESSED) != 0){
        ship.moveShip(3);
    }else if((keyStates & RIGHT_PRESSED) != 0){
        ship.moveShip(4);
    }else{
        ship.moveShip(0);
    }
    /*******

    // движение объектов в игре
    /*******

    // движение шара

```



```

    boll.moveBoll();
    // движение карты
    background.moveBackground(screenHeight);
}
}

```

В классе `MainGameCanvas` происходят объявление, создание и инициализация объектов классов `Ship`, `Background` и `Boll`. UML-диаграмма этого класса изображена на рис. 13.9. Класс `MainGameCanvas` - это основной игровой класс, в котором происходит работа всей программы.

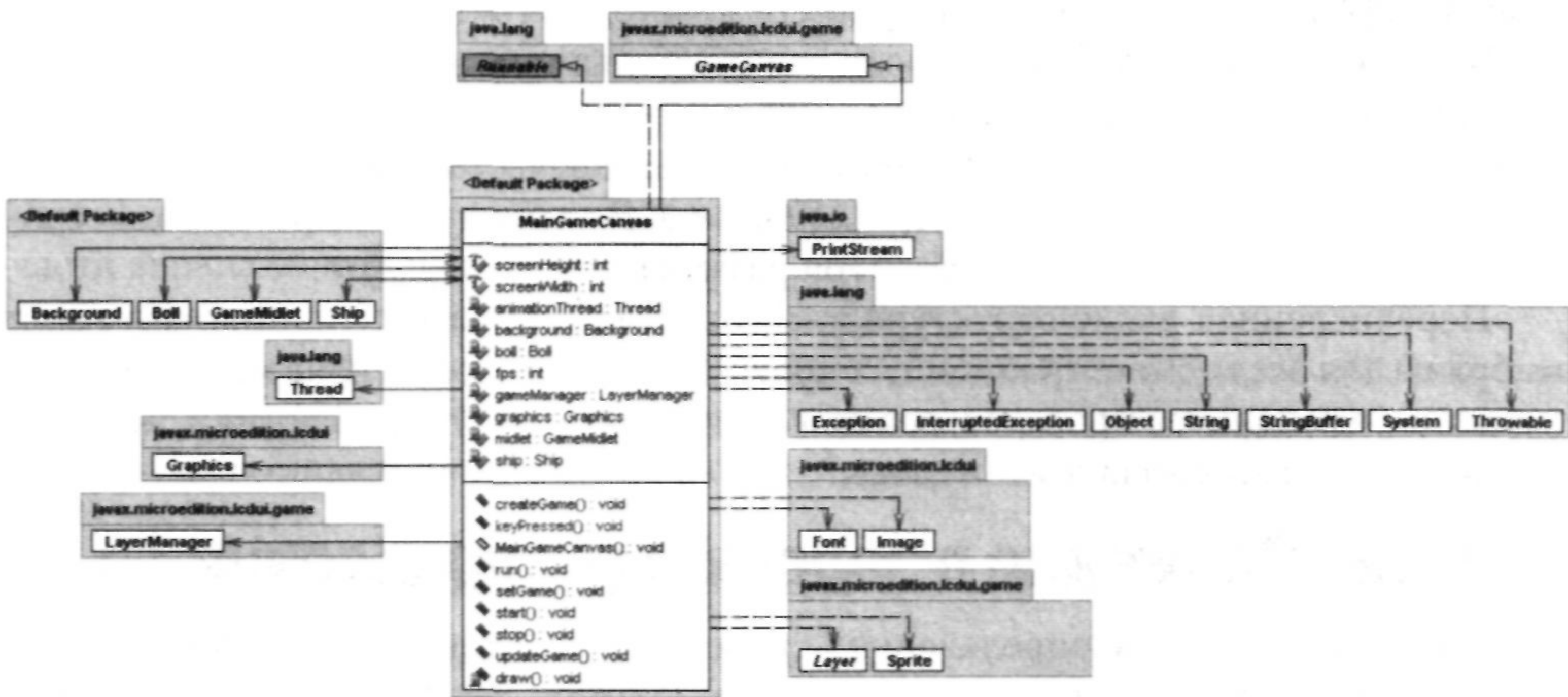


Рис. 13.9. UML-диаграмма класса `MainGameCanvas`

Метод `run()` запускает игровой цикл и отвечает за постоянное обновление игрового процесса. В методе `run()` происходит вызов методов `updateGame()` и `draw()`. Этот метод отвечает за обработку пользовательского ввода, движение карты и шара по экрану телефона.

Метод `run()` класса `MainGameCanvas` необходим для рисования графики и анимации на экране. В этот метод также был встроен информационный и отладочный механизм. Когда программа работает, на экране телефона в реальном режиме вы можете наблюдать за текущими координатами двух объектов, скоростью шара и другой технической информацией. Для рисования текста на экране телефона используются класс `Font` и метод `drawString()`.

```

graphics.setFont(Font.getFont(Font.FACE_SYSTEM,
Font.STYLE_BOLD,
    Font.SIZE_SMALL));
graphics.drawString("Boll X = " + boll.getX(), 2,
    Font.getDefaultFont().getHeight(), 0);
graphics.drawString("Boll Y = " + boll.getY(), 2,
    Font.getDefaultFont().getHeight()*2, 0);
graphics.drawString("Ship X = " + ship.getX(), 2,

```

```
Font.getDefaultFont().getHeight()*3, 0);
graphics.drawString("Ship Y = "+ship.getY(), 2,
Font.getDefaultFont().getHeight()*4, 0);
```

Метод `drawString()` имеет четыре параметра. Первый параметр - это строка текста, второй и третий параметры задают начальную точку вывода текста на экране по осям X и Y . В последнем параметре с помощью констант задается место вывода строки на экране. Этот параметр можно не использовать, достаточно установить его значение, равное 0.

Для рисования текста на экране используется маленький жирный шрифт. Текущая информация рисуется с отступом от левого края экрана на 2 пикселя по оси X . По оси Y мы смещаем точку вывода для каждой строки на размер, равный высоте шрифта. Чтобы каждая новая строка текста рисовалась под предыдущей строкой, высота шрифта умножается на количество вышестоящих строк, например `Font.getDefaultFont().getHeight()*3`. В этом случае точка вывода строки текста по оси Y будет составлять три размера высоты шрифта.

Перечисленная выше структура классов и модель работы программы были выбраны для всех демонстрационных примеров этой главы. Сейчас мы переходим к анализу имеющихся проектов, реализующих примеры игровой логики, задекларированной нами в начале этой главы.

13.2. Движение в заданном направлении

Движение объекта с определенной скоростью в заданном направлении является простейшим детерминированным алгоритмом. *Детерминированный алгоритм* - это заданное поведение конкретно взятого объекта в ходе игрового процесса. Детерминированные алгоритмы очень часто применяются в играх для различных платформ и относятся к простым алгоритмам, которые необходимы для создания несложных игровых интеллектов.

В первом примере мы рассмотрим простейшее движение объекта на экране телефона с заданной скоростью и направлением. Объект будет двигаться сверху вниз по оси Y со скоростью в 5 пикселей за один игровой цикл. Как вы уже знаете, для перемещения объектов в играх используется метод `move(int x, int y)`. Этот метод имеет два целочисленных параметра, с помощью которых устанавливаются скорость и направление движения объекта по осям X и Y . Чтобы шар двигался сверху вниз, его необходимо установить на позицию в верхней части экрана и вызвать метод `move(0, 5)`, установив тем самым скорость движения объекта в 5 пикселей по оси Y . Ниже приведен исходный код метода `moveBoll()` класса `Boll`.

```
public void moveBoll() {
    // следующий фрейм
    this.nextFrame();
    // передвижение шарика
    this.move(0, 5);
    // обработка столкновения шарика с окончанием экрана
    if (this.getY() > gameCanvas.screenHeight) {
```

}

setPosition()

Экран

move(0, 5)

The diagram illustrates a ship's movement in a 2D coordinate system. A central circle represents the ship. Four arrows point outwards from this circle, each labeled with a speed value: `speedX=-3` (left), `speedX=3` (right), `speedY=-3` (up), and `speedY=3` (down). At the end of each arrow is a ship icon and a label for its current position: `ship.getX()` for the left and right positions, and `ship.getY()` for the top and bottom positions. A diagonal arrow also points from the center, labeled with `speedY=-3` and `speedX=3`. The coordinate system is defined by a horizontal axis labeled `+X` and a vertical axis labeled `+Y`, with the origin `0,0` at the top-left corner.

Из рис. 13.11 видно, что происходит постоянное сравнение позиций объекта и цели, и если цель находится впереди относительно объекта, то скорость объекта

задается положительными значениями, а если сзади - то отрицательными. Эта задача очень легко решается и на программном уровне. Для определения положения цели нам понадобится передать в метод `moveBoll()` искомый объект класса `Ship`, как это сделано в следующем исходном коде.

```
public void moveBoll(Ship ship /*Sprite sprite*/) {
    // следующий фрейм
    this.nextFrame();
    // проверяем, где корабль
    if (this.getX() > ship.getX()) {
        speedX = -3;
    } else if (this.getX() < ship.getX()) {
        speedX = 3;
    }
    if (this.getY() > ship.getY()) {
        speedY = -3;
    } else if (this.getY() < ship.getY()) {
        speedY = 3;
    }
    // движение шарика
    this.move(speedX, speedY);
}
```

Работа метода `moveBoll()` заключается в постоянном сравнении позиции шара и корабля с изменением направления движения шарика. Если текущая координата по осям `X` и `Y` у шара больше, чем у корабля, то скорость должна быть равна отрицательному значению `speedX = -3` и `speedY = -3`, если наоборот, то уже используется положительное значение скорости `speedX = 3` и `speedY = 3`.

Скорость, с которой шар движется к цели, необходимо тщательно подбирать. Если скорость движения шара до цели будет больше, чем у корабля, то корабль просто не сможет убежать от объекта, если, конечно/это не входило в ваши первоначальные планы. В связи с этим необходимо продумать и равномерно распределить все скоростные значения объектов и целей в игре.

Еще одной важной составляющей метода `moveBoll()` может послужить его неоднократное использование в игре. Для того чтобы исходный код метода `moveBoll()` был универсальным, лучше передавать в качестве параметра метода `moveBoll()` объект класса `Sprite`, который является суперклассом для всех спрайтовых классов. Универсальный подход в реализации метода `moveBoll()` позволит вам использовать этот метод многократно не только в этой игре, но и в других своих проектах. Напоминаю, что готовая программа и сам исходный код примера находятся на компакт-диске в папке **Code\Chapter13\AI2**.

13.4. Движение объекта от цели

Порой в играх необходимо предусмотреть обратный механизм движения объекта от цели. В случае с нашим примером шар должен убегать от корабля по мере его

приближения. Алгоритм решения этой задачи на самом деле очень простой, и все, что нужно сделать, - это модифицировать исходный код предыдущего примера. Посмотрите на рис. 13.12, где представлена примерная схема решения поставленной перед нами задачи.

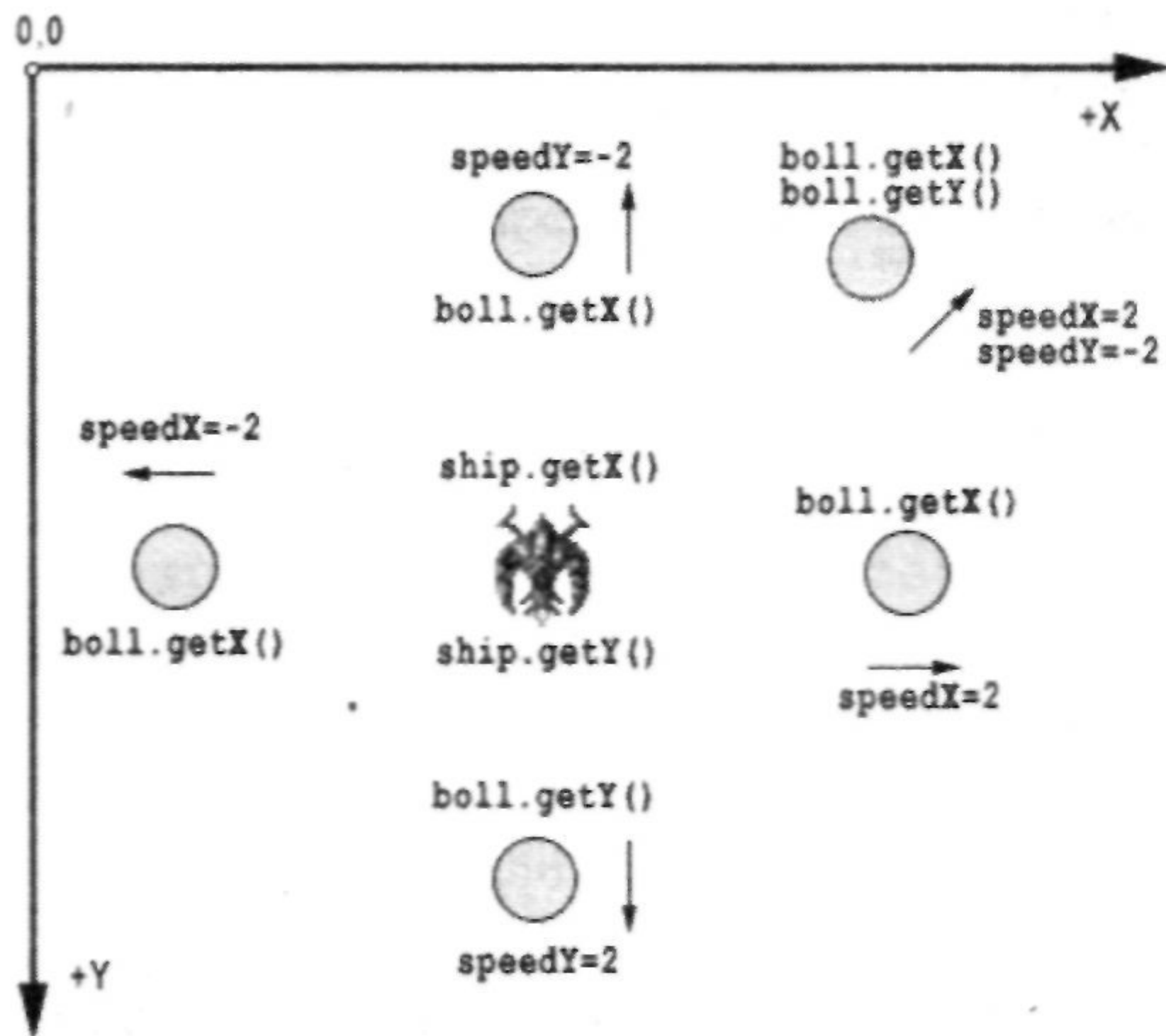


Рис. 13.12. Движение объекта от цели

Для того чтобы шар убегал от объекта, в методе `moveBoll()` необходимо изменить значения скоростей `speedX` и `speedY` по двум осям координат на противоположные. Если `this.getX() > sprite.getX()`, то скорость по оси `X` будет равна уже `speedX = 2`, а не отрицательному значению, как в предыдущем примере, что дает возможность шару двигаться в противоположном от корабля направлении. Посмотрите на видоизмененный исходный код метода `moveBoll()`, ориентированный на движение объекта от цели.

```
public void moveBoll(Sprite sprite){
    // следующий фрейм
    this.nextFrame();
    // определяем, где находится корабль
    if (this.getX() > sprite.getX()){
        speedX = 2;
    }else if(this.getX() < sprite.getX()){
        speedX = -2;
    }
    if (this.getY() > sprite.getY()){
        speedY = 2;
    }else if (this.getY() < sprite.getY()){
        speedY = -2;
    }
    // движение шара
```

```

this.move(speedX, speedY);
// столкновение с окончанием экрана
if (this.getX() + this.WIDTH < 0) {
    this.setPosition(gameCanvas.screenWidth/2,
        gameCanvas.screenHeight/2);
} else if (this.getX() > gameCanvas.screenWidth) {
    this.setPosition(gameCanvas.screenWidth/2,
        gameCanvas.screenHeight/2);
}
if (this.getY() + this.HEIGHT < 0) {
    this.setPosition(gameCanvas.screenWidth/2,
        gameCanvas.screenHeight/2);
} else if (this.getY() > gameCanvas.screenHeight) {
    this.setPosition(gameCanvas.screenWidth/2,
        gameCanvas.screenHeight/2);
}
}

```

Несколько слов еще хочется сказать о последнем блоке исходного кода метода `moveBall()`, где обрабатывается столкновение с окончанием экрана телефона. Поскольку шар удаляется от корабля и его движение неконтролируемое, то нам необходимо позаботиться о ситуации, которая возникает в момент исчезновения шарика с экрана.

Решить эту задачу можно множеством различных способов, но поскольку это демонстрационный пример, то необходимо, чтобы после исчезновения шара с экрана телефона он появлялся на экране снова и снова. Для этого был написан блок исходного кода, который регулирует появление шара на дисплее вновь и вновь после его выхода за пределы экрана.

Идея очень проста: как только шар исчезает с экрана телефона в любом направлении, метод `setPosition()` устанавливает шар в центр экрана, и так до бесконечности или до окончания работы программы. Полный исходный код этого примера вы найдете на компакт-диске в папке **Code\Chapter13\AI3**.

13.5. Движение в случайном направлении

В играх движение объектов может зависеть от различных внешних факторов, но очень часто необходимо использовать элементарный алгоритм перемещения объекта в пространстве, основанный на случайном выборе направления. Случайное движение объекта в игре происходит путем изменения его направления перемещения в пространстве. Самый простой способ изменения направления движения шара сводится к периодическому изменению значения скорости по осям X и Y с отрицательного значения на положительное и наоборот. На рис. 13.13 показано, как можно изменять скорость для смены направления движения шара.

Сложность реализации механизма случайного перемещения заключается в задании периодичности смены движения, а также в изменении скорости самого объекта.

в случайном направлении

Периодичность смены движения объекта напрямую зависит от игрового цикла. Метод `moveBall()` вызывается в методе `updateGame()` класса `MainGameCanvas`, который, в свою очередь, вызывается в каждом проходе игрового цикла метода `run()`, что составляет примерно 30 вызовов в одну секунду. Этот подход позволяет добиться плавной анимации в игре. Смену направления движения необходимо производить в игровом цикле, но смена движения шара не может быть 30 раз за секунду, то есть она не может изменяться 30 раз за одну секунду. Поэтому необходимо установить определенную периодичность изме-

Задавать периодичность смены движения объекта можно различными способами. Например, использовать подсчет количества пройденных циклов или изменять периодичность смены движения в случайном порядке. Подсчет пройденных циклов мы рассмотрим далее в этой главе, а в данном примере будет использован механизм периодичной смены направления движения, основанной на случайном выборе. Для этих целей нам нужно генерировать последовательность случайных чисел.

Для генерации последовательности случайных чисел в Java 2 ME используется класс `Random` из пакета `java.util`. Чтобы воспользоваться классом `Random`, в нашем примере необходимо в класс `Roll` в начале исходного кода подключить пакет `java.util` следующим образом:

```
import java.util.*;
```

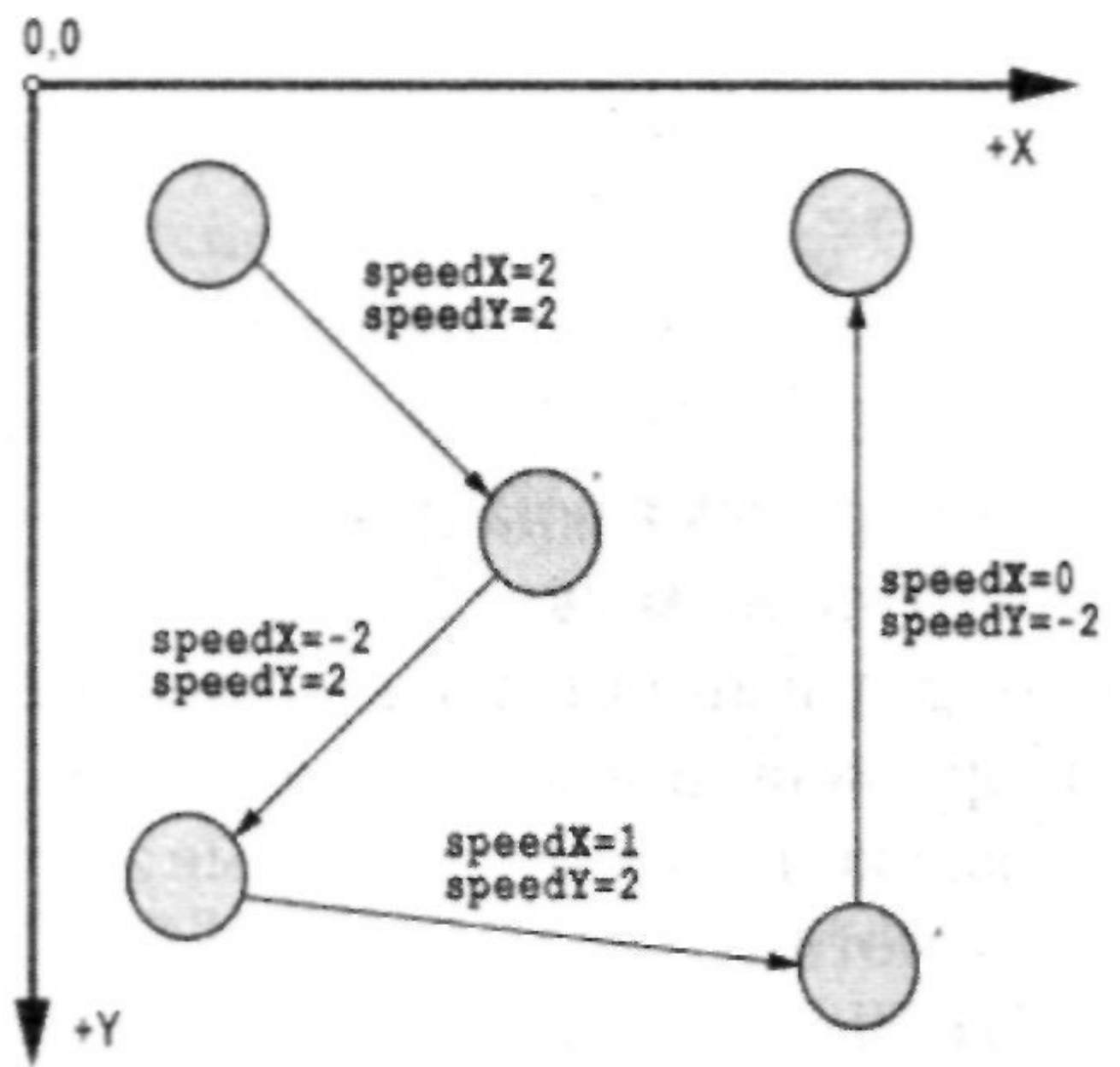
Затем глобально создать объект класса Random.

```
private Random random = null;
```

Далее в конструкторе класса `Ball` инициализировать созданный объект и использовать его в дальнейшем по ходу работы над исходным кодом программы.

```
random = new Random();
```

Суть способа случайной смены направления движения заключается в том, чтобы генерировать случайное целое число без знака и делить его по модулю на заданное вами целое число. То есть на каждом проходе игрового цикла будет генерироваться целое число, делиться по модулю и сравниваться с нулем. Если результат деления этой операции равен нулю, то следует изменение скорости или направления движения шара, если нет, то продолжают действовать установленные ранее значения скорости объекта. Таким образом мы добиваемся случайной периодичности смены движения шара. Для генерации случайного числа используется метод `nextInt()` класса `Random`, который генерирует случайную последовательность чисел.



```
if(random.nextInt() % 7 == 0){
    // смена скорости и направления движения
}else{
    // нет смены скорости и направления движения
}
```

Цифра семь в данном случае - это как раз и есть заданное нами целое число. Это число может быть любым. Чем больше эта цифра, тем, возможно, дольше будут выполняться установленные ранее значения скорости, а также при выборе заданной цифры лучше использовать нечетные цифры, поскольку, как показывает практика, четные значения почему-то генерируются гораздо чаще, чем нечетные.

Что касается механизма смены скорости объекта, то здесь тоже можно использовать различные схемы решения задачи. Самая простая и наиболее используемая схема - это простая генерация целых чисел в заданном диапазоне. Для подобных целей используется метод `nextInt()`, а для ограничения диапазона значений применяются два метода - `Math.min()` и `Math.max()`, возвращающие соответственно минимальное и максимальное полученные значения.

```
public void moveBall(){
    // следующий фрейм
    this.nextFrame();
    // выбор случайной скорости
    if(random.nextInt() % 7 == 0){
        speedX = Math.min(Math.max(speedX+random.nextInt()%3,
            -2),2);
        speedY = Math.min(Math.max(speedY+random.nextInt()%3,
            -2),2);
    }
    // движение шара
    this.move(speedX, speedY);
    // столкновение с окончанием экрана
    if(this.getX() + this.WIDTH < 0){
        this.setPosition(gameCanvas.screenWidth/2,
            gameCanvas.screenHeight/2);
    }else if(this.getX() > gameCanvas.screenWidth){
        this.setPosition(gameCanvas.screenWidth/2,
            gameCanvas.screenHeight/2);
    }
    if(this.getY() + this.HEIGHT < 0){
        this.setPosition(gameCanvas.screenWidth/2,
            gameCanvas.screenHeight/2);
    }else if(this.getY() > gameCanvas.screenHeight){
        this.setPosition(gameCanvas.screenWidth/2,
            gameCanvas.screenHeight/2);
    }
}
```



```
}  
}
```

При выборе случайного значения скорости используется операция деления по модулю на число 3. Число 3 установлено произвольно, и вы можете использовать любое другое целое число, но сама операция деления по модулю дает возможность перебора значений в диапазоне от -2 до 2. Если не использовать деление по модулю при изменении скорости, например вот так:

```
speedX = Math.min(Math.max(speedX + random.nextInt(), -2), 2);  
speedY = Math.min(Math.max(speedY + random.nextInt(), -2), 2);
```

то скорость будет изменяться только на два значения -2 и 2. Тогда как деление по модулю дает возможность выбора значений -2, -1, 0, 1, 2. Откройте демонстрационный пример **AI4** и поэкспериментируйте с выбором различных значений. На экране телефона в момент работы демонстрационного примера **AI4** вы сможете наблюдать за текущим изменением скорости шара.

В этом примере шар движется в случайном направлении, и мы не можем контролировать его перемещение, поэтому следует обязательно применять механизм обработки условий, когда шар выходит за пределы экрана телефона. Как всегда, полный исходный код этого проекта вы найдете на компакт-диске в папке **Code\Chapter13\AI4**.

13.6. Шаблоны

Одно из мощнейших орудий в создании искусственного интеллекта заключается в использовании шаблонов. *Шаблон* - это набор команд или инструкций, определяющих поведение игрового объекта. Создавая шаблон, вы фактически создаете набор инструкций, которые должен выполнить тот или иной объект в ходе всего игрового процесса. Создание шаблона строится на изученных нами детерминированных алгоритмах движения объектов.

Шаблон может содержать различные команды, например один шаблон реализует движение зигзагом, второй предусматривает движение объекта сверху вниз, третий - движение сверху вниз и налево, четвертый - сверху вниз и направо и т. д. Посмотрите на рис. 13.14, где представлен примерный механизм работы шаблонов.

Как видно из рис. 13.14, было создано четыре гипотетических шаблона, описывающих движение объекта. С помощью оператора `switch` происходит выбор одного из шаблонов, задающих поведение игровому объекту. Количество шаблонов может быть любым, здесь все зависит только от вашей фантазии. Создав определенный набор шаблонов на все случаи жизни, вы можете использовать их в игре на любом этапе.

Выбор того или иного шаблона в целом зависит от событий, происходящих в игре. Можно предусмотреть последовательный или случайный выбор одного из шаблонов. Что касается случайного выбора шаблона, то здесь все очень просто - необходимо использовать класс `Random` для генерирования целого числа

Рис. 13.14. Набор шаблонов

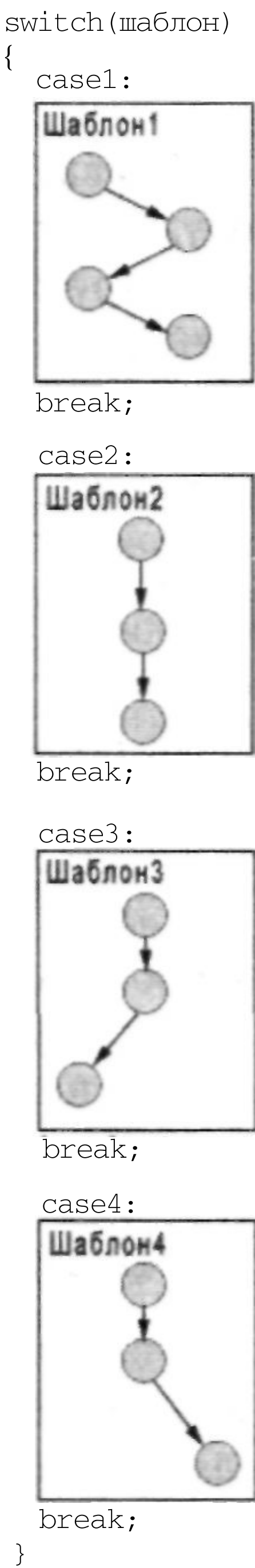
и на основе полученного значения выбирать соответствующий шаблон. Например, у вас имеются 15 шаблонов, чтобы выбрать случайно один из шаблонов, можно использовать следующую конструкцию исходного кода:

```
direction = Math.min(Math.max(direction +
    random.nextInt() % 3, 1), 15);
```

В этом коде происходит случайное генерирование целого числа, а ограничивающие рамки установлены в промежутке от 1 до 15, что как раз и соответствует 15 имеющимся у вас шаблонам. Для периодичности смены шаблонов можно использовать механизм, рассмотренный в предыдущем разделе, или задействовать счетчик циклов, который мы разберем далее в этой главе.

В демонстрационном примере этого раздела мы изучим последовательный перебор имеющихся у нас шаблонов. Таких шаблонов всего два, и они реализуют движение объекта сверху вниз и влево и движение сверху вниз и направо. На рис. 13.15 показано, как работают шаблоны и реализована система последовательной смены предлагаемых двух шаблонов. Исходный код проекта вы найдете на компакт-диске в папке **Code\Chapter13\AI5**.

```
public void moveBall(){
    // следующий фрейм
    this.nextFrame();
    // выбор шаблона
    switch(direction){
        // движение влево
        case 1:
            if(this.getY() > gameCanvas.screenHeight/2){
                speedX = -3;
                speedY = 2;
            }else{
                speedX = 0;
                speedY = 5;
            }
            break;
        // движение вправо
        case 2:
            if(this.getY() > gameCanvas.screenHeight/2){
                speedX = 3;
                speedY = 2;
            }else{
                speedX = 0;
                speedY = 5;
            }
            break;
    }
}
```



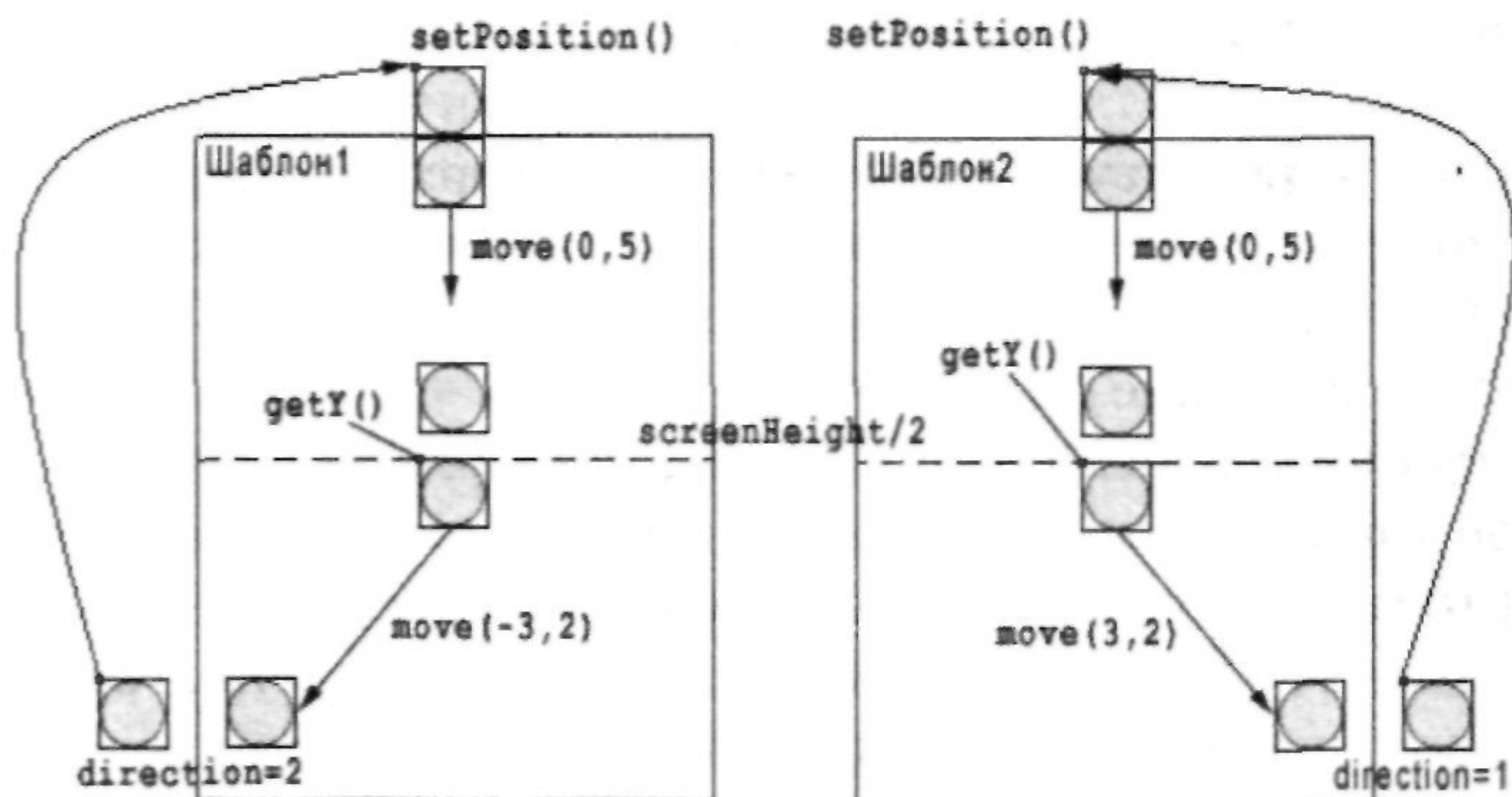


Рис. 13.15. Работа шаблонов и система смены последовательностей

```

        speedX = 0;
        speedY = 5;
    }
break;
// по умолчанию
default:
    speedX = 0;
    speedY = 5;
    direction = 1;
break;
)

// движение шара
this.move(speedX, speedY);
// столкновение с окончанием экрана
if (this.getX() + this.WIDTH < 0){
    this.setPosition(gameCanvas.screenWidth/2 -
        this.WIDTH/2,
        -this.HEIGHT);
    direction = 2;
}
if (this.getX() > gameCanvas.screenWidth) {
    this.setPosition(gameCanvas.screenWidth/2 -
        this.WIDTH/2,
        -this.HEIGHT);
    direction = 1;
}
}
}

```

Для выбора одного из шаблонов определена глобальная переменная `direction`, объявленная в начале исходного кода класса `Ball` со значением, равным 1.

Соответственно, при запуске программы изначально происходит выполнение первого шаблона.

Первый шаблон реализует движение шара сверху вниз и налево. Первоначально шар двигается по прямой сверху вниз со скоростью 5 пикселей по оси Y за один игровой цикл. По достижении половины экрана телефона скорость движения изменяется на значение $speedX = -3$ и $speedY = 2$, а соответственно, и изменяется направление движение шара, который начинает двигаться влево. Когда шар доходит до конца экрана с левой стороны, он устанавливается на ту позицию, откуда он стартовал, а затем переменной `direction` присваивается значение, равное 2, и, следовательно, работа программы переходит к выполнению второго шаблона.

Работа второго шаблона аналогична работе первого, с той лишь разницей, что движение объекта происходит сверху вниз и направо. Как только шар уходит за пределы экрана с правой стороны, переменной `direction` присваивается значение, равное 1, и происходит переход к выполнению первого шаблона, и так происходит до бесконечности или окончания работы программы.

Как видите, использование шаблонов в играх дает значительно больше гибкости игровому интеллекту, а главное, что вы можете реализовать практически любое поведение объекта в шаблоне и дать объекту команду на исполнение этих действий в необходимый момент.

13.7. Шаблоны с обработкой событий

Рассмотренные нами в предыдущем разделе шаблоны являются простыми шаблонами, и во время своей работы они не учитывают возможной смены текущих игровых событий. Для этих целей очень хорошо использовать шаблоны с обработкой событий или условий. Шаблон с обработкой событий фактически ничем не отличается от простого шаблона, с той лишь разницей, что в сам шаблон встроен исходный код, позволяющий обрабатывать внешние игровые события.

В нашем новом примере шар движется сверху вниз со скоростью 5 пикселей по оси Y до половины экрана телефона, после чего включается механизм, отслеживающий положение корабля. В этот момент шаблон начинает обрабатывать внешние события, происходящие в игре. Если корабль находится в левой части экрана, то шар движется в левую сторону, если корабль находится с правой стороны, то и шар перемещается в правую сторону экрана (см. рис. 13.16). Демонстрационный пример шаблона с обработкой условий вы найдете на компакт-диске в папке **Code\Chapter13\AI6**.

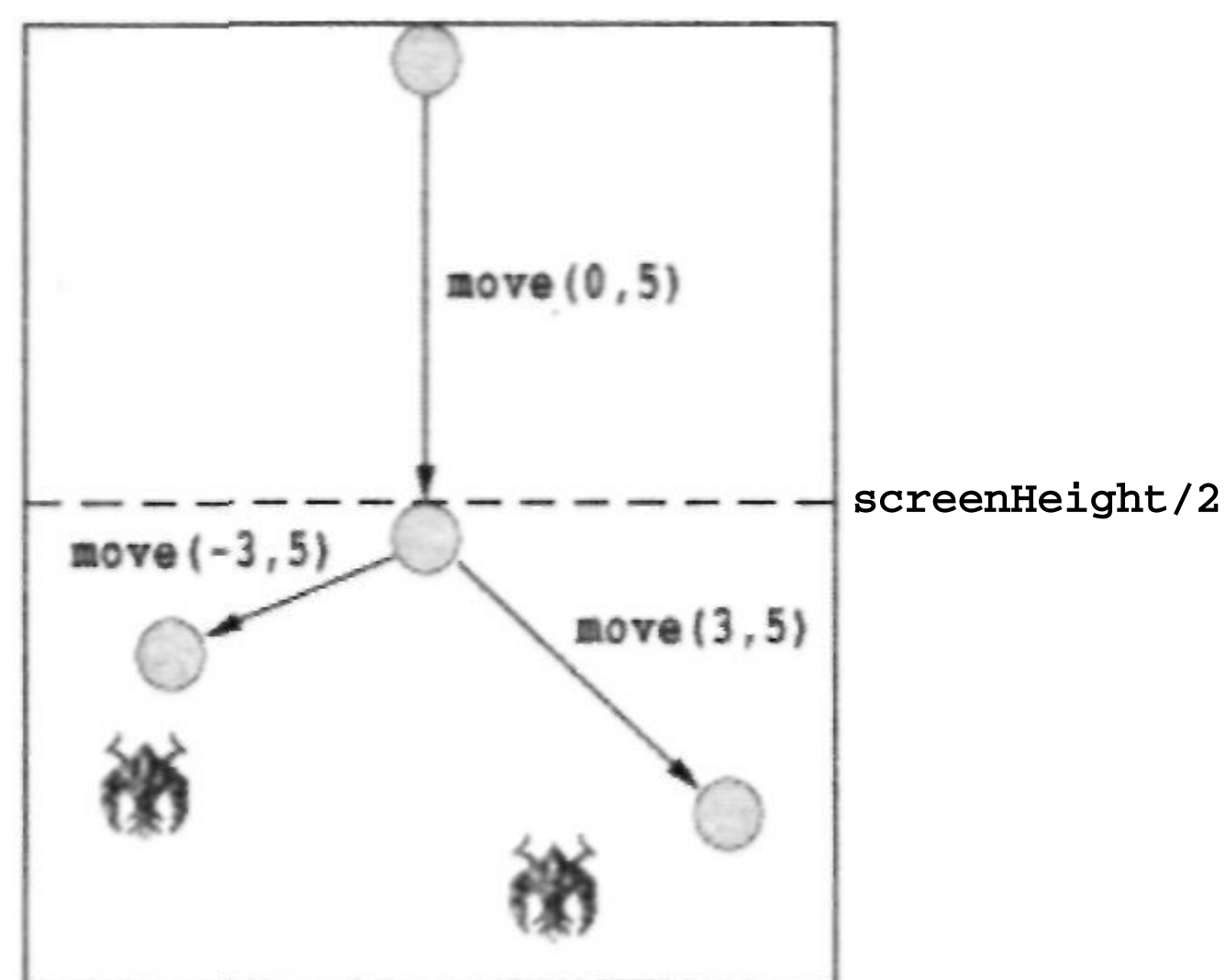


Рис. 13.16. Шаблон с обработкой событий

```
public void moveBall(Sprite sprite){
    // следующий фрейм
```



```
this.nextFrame();
// выбор действий
if (this.getY() > gameCanvas.screenHeight/2) {
    if(sprite.getX() > this.getX()){
        speedX = 3;
        speedY = 5;
    }else if(sprite.getX() < this.getX()){
        speedX = -3;
        speedY = 5;
    }
}else{
    speedX = 0;
    speedY = 5;
}

// движение шара
this.move(speedX, speedY);
// столкновение с окончанием экрана
if (this.getX() + this.WIDTH < 0) {
    this.setPosition(gameCanvas.screenWidth/2,
gameCanvas.screenHeight/2);
}else if (this.getX() > gameCanvas.screenWidth) {
    this.setPosition(gameCanvas.screenWidth/2,
gameCanvas.screenHeight/2);
}
if (this.getY() + this.HEIGHT < 0) {
    this.setPosition(gameCanvas.screenWidth/2,
gameCanvas.screenHeight/2);
}else if (this.getY() > gameCanvas.screenHeight) {
    this.setPosition(gameCanvas.screenWidth/2,
gameCanvas.screenHeight/2);
}
}
```

При детальном рассмотрении этого примера можно заметить, что шаблон с обработкой событий построен на основе двух шаблонов. Первый шаблон перемещает объект в заданном направлении, не обращая внимания на внешние факторы, но как только шар достигает половины экрана, происходит переход ко второму шаблону и шар уже начинает обрабатывать внешние игровые события и двигаться в сторону корабля. Этот пример использования двух шаблонов представляет нам алгоритм смены текущего состояния объекта и переход от одной подсистемы искусственного интеллекта к другой, реализованной в виде двух абсолютно разных шаблонов.

Шаблоны - это отличный инструмент для создания игрового интеллекта. Игровые шаблоны широко используются при создании компьютерных и мобильных

игр. Если хорошо продумать и реализовать механизм работы шаблонов, то можно создать отличную игровую логику, потратив при этом минимум усилий и времени.

13.8. Модель простой системы смены состояний

На базе полученных знаний в этом разделе мы смоделируем простую систему смены состояний игрового объекта. *Состояние игрового объекта* - это определенные действия объекта в игре, реализованные на базе шаблонов или простых алгоритмов движения объекта в пространстве. В этом проекте мы фактически будем использовать различные шаблоны, которые моделируют текущее состояние объекта. Смена состояния объекта в новом примере, как и было обещано, совершается на основе подсчета прошедших игровых циклов.

В проекте используются четыре разных состояния объекта, или четыре разных шаблона. В качестве шаблонов использованы приемы, изученные в этой главе. Первый шаблон - это заданное движение объекта слева направо, второй и третий шаблоны выполняют движение объекта за целью и от цели, а последний шаблон реализует случайное движение в пространстве.

Периодичность смены шаблонов основана на подсчете пройденных игровых циклов. Выбор определенного шаблона происходит при помощи оператора `switch` и переменной `direction`. Для того чтобы разобраться, как работает алгоритм периодичной смены шаблонов, повторим еще раз работу всего игрового цикла.

Игровой цикл постоянно обновляет состояние игры, и в этом состоит его основная задача. За одну секунду в среднем происходит смена 30 игровых циклов, или кадров игры. Количество смены циклов зависит от схемы, которой вы придерживаетесь при создании игрового цикла. В нашем примере это порядка 30 циклов за одну секунду. В итоге метод `moveBoll()` за одну секунду вызывается около 30 раз. Получается, что 30 вызовов метода `moveBoll()`, или 30 проходов игрового цикла, по времени равны примерно одной секунде. Чтобы каждый шаблон работал по 5 секунд (30 циклов x 5 секунд = 150 циклов), можно использовать счетчик циклов, посмотрите на исходный код примера.

```
int counter = 0 ;
int direction = 1 ;

public void moveBoll(Sprite sprite){
    // следующий фрейм
    this.nextFrame() ;
    // счетчик циклов
    if (++counter > 150) {
        if(direction >= 4){
            direction = 1;
        }else{
```

```
        direction++;
    }
    counter = 0;
}
// выбор
switch(direction){
// движение слева направо
case 1:
    this.move(speedX, 0);
    if (this.getX() < 0) {
        speedX = 5;
    }
    if (this.getX() > gameCanvas.screenWidth -
        this.WIDTH){
        speedX = -5;
    }
break;
// движение за целью
case 2:
    if (this.getX() > sprite.getX()){
        speedX = -1;
    }else if (this.getX() < sprite.getX()){
        speedX = 1;
    }
    if(this.getY() > sprite.getY()){
        speedY = -1;
    }else if (this.getY() < sprite.getY()){
        speedY = 1;
    }
    this.move(speedX, speedY);
break;
// движение от цели
case 3:
    if (this.getX() > sprite.getX()){
        speedX = 1;
    }else if (this.getX() < sprite.getX()) {
        speedX = -1;
    }
    if (this.getY() > sprite.getY()) {
        speedY = 1;
    }else if (this.getY() < sprite.getY()){
        speedY = -1;
    }
    this.move(speedX, speedY);
}
```

```

break;
// движение в случайном направлении
case 4 :
    if(random.nextInt() % 3 == 0){
        speedX
=Math.min(Math.max(speedX+random.nextInt()%3, -2), 2);
        speedY
=Math.min(Math.max(speedY+random.nextInt()%3, -2), 2);
    }
    this.move(speedX, speedY);
break;

default:
break;
}

// столкновение с окончанием экрана
if (this.getX() + this.WIDTH < 0) {
    this.setPosition(gameCanvas.screenWidth/2,
gameCanvas.screenHeight/2);
}else if(this.getX() > gameCanvas.screenWidth){
    this.setPosition(gameCanvas.screenWidth/2,
gameCanvas.screenHeight/2);
}
if (this.getY() + this.HEIGHT < 0) {
    this.setPosition(gameCanvas.screenWidth/2,
gameCanvas.screenHeight/2);
}else if (this.getY() > gameCanvas.screenHeight) {
    this.setPosition(gameCanvas.screenWidth/2,
gameCanvas.screenHeight/2);
}
}
}

```

Постоянное сравнение переменной `counter` со значением 150 задает периодичность смены шаблона. В этой конструкции исходного кода целочисленная переменная `counter` (счетчик циклов) равна 0, и на каждой итерации игрового цикла ее значение увеличивается на 1. Одновременно происходит сравнение переменной `counter` со значением 150 (30 циклов x 5 секунд). Если значение переменной `counter` меньше, чем 150, то переменная `direction` равна своему текущему значению, если `counter > 150`, то переменная `direction` увеличивается на 1. То есть по прошествии 150 циклов, или примерно 5 секунд игрового времени, переменная `direction` изменяет свое значение и происходит смена игрового шаблона с заданной нами периодичностью.

В примере у нас всего четыре шаблона, а это значит, что переменная `direction` не должна быть больше, чем 4, поэтому мы следим за ее значением с помощью операторов `if/else`. Как только выполняется работа всех четырех шаблонов, то есть

значение переменной `direction` становится больше, чем 4, то этой переменной присваивается значение, равное 1, и происходит выбор первого шаблона. Тем самым мы добиваемся циклической смены состояний объекта, основанной на подсчете прошедших игровых циклов. Полный листинг исходного кода этого примера находится на компакт-диске в папке `Code\Chapter13\AI7`.

13.9. Распределенная логика смены состояний объекта

В предыдущих разделах мы моделировали различные состояния объектов в игре. Выбор последовательности смены состояний объекта строился нами по-разному. Мы использовали переключение состояний с заданной периодичностью, случайный выбор состояния объекта и подсчет пройденных циклов. Во всех случаях использовался механизм равновероятного переключения состояний. Для создания более мощной системы игровой логики можно применять распределенную логику смены состояния объекта на основе расстояния от объекта до цели. Посмотрите на рис. 13.17, где в графическом виде представлен механизм смены состояний.

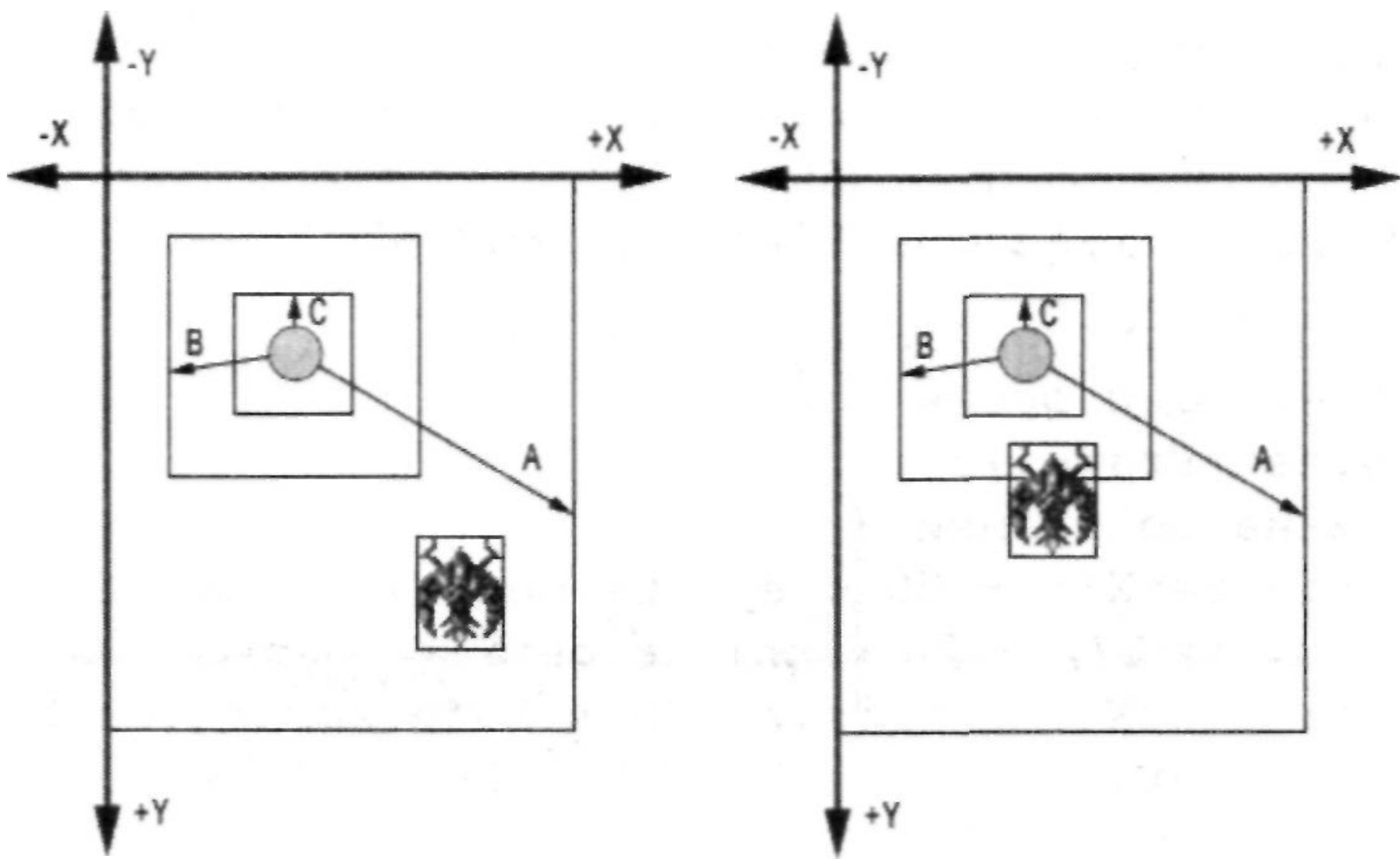


Рис. 13.17. Распределение вероятности смены состояний

Мы имеем три состояния шара, или три разных шаблона, для реализации заданного вида движения шара в пространстве, обозначенные на рис. 13.17 как А, В и С:

- А - движение шара в случайном направлении;
- В - шар движется в заданном направлении;
- С - шар убегает от корабля.

Исходный код примера на рис. 13.17 должен рассчитывать расстояние от шара до корабля, и об этом мы поговорим позже в этом разделе, когда будем рассматривать демонстрационный пример. Сам механизм смены состояний шара, зависящий от расстояний между шаром и кораблем, заключается в следующем.

Если корабль находится в зоне действия шаблона А, то шар перемещается в случайном направлении. Как только корабль приближается к шару и попадает в зону действия шаблона В, то шар начинает двигаться по экрану в заданном направлении. Когда корабль приблизится к шару совсем близко и попадет в зону действия шаблона С, то шар начнет удаляться от корабля или убегать. Получается, что в зависимости от расстояний между шаром и кораблем происходит автоматический выбор смены состояния игрового объекта.

Такой алгоритм действий позволяет формировать и моделировать очень сильные системы игровой логики. Вы можете создать любое количество состояний объекта, сформированного на основе шаблонов или простых алгоритмов движения объекта, и применять их в зависимости от расстояний между объектом и целью.

В демонстрационном примере мы рассмотрим работу двух состояний объекта, смена которых основана на расчете расстояния от шара до корабля. Перемещать корабль можно в любом направлении на экране. Первое состояние шара (1) оставляет шар на экране телефона без движения. Если корабль будет находиться в зоне действия состояния под номером 1, то шар остается без движения. Как только корабль приблизится к шару и попадет в зону действия второго состояния (2), то шар начнет приближаться и атаковать корабль. Если корабль выйдет из зоны действия второго состояния (№ 2) и попадет в зону действия первого состояния (1), то шар вновь перестанет двигаться (рис. 13.18).

Давайте рассмотрим метод `moveBoll()` класса `Boll`, где и реализована данная логика движения объекта. Полный листинг исходного кода этого примера находится на компакт-диске в папке **Code\Chapter13\AI8**.

```
public void moveBoll(Sprite sprite){
    // следующий фрейм boll
    this.nextFrame();
    // смена состояний
    if(this.getX() - 20 < sprite.getX() + sprite.getWidth()
    && this.getY() - 20 < sprite.getY() + sprite.getHeight()
    && this.getX() + this.WIDTH + 20 > sprite.getX()
    && this.getY() + this.HEIGHT + 20 > sprite.getY()){
        if (this.getX() > sprite.getX()){
            speedX = -1;
        }else if (this.getX() < sprite.getX()){
            speedX = 1;
        }
        if (this.getY() > sprite.getY()){
            speedY = -1;
        }else if (this.getY() < sprite.getY()){
            speedY = 1;
        }
    }else{
        speedX = 0 ;
        speedY = 0 ;
    }
}
```

```

    }

    // движение boll
    this.move(speedX , speedY);
    // столкновение с окончанием экрана
    if(this.getX() + this.WIDTH < 0){
        this.setPosition(gameCanvas.screenWidth/2,
            gameCanvas.screenHeight/2);
    }else if(this.getX() > gameCanvas.screenWidth){
        this.setPosition(gameCanvas.screenWidth/2,
            gameCanvas.screenHeight/2);
    }
    if (this.getY() + this.HEIGHT < 0) {
        this.setPosition(gameCanvas.screenWidth/2,
            gameCanvas.screenHeight/2);
    }else if(this.getY() > gameCanvas.screenHeight){
        this.setPosition(gameCanvas.screenWidth/2,
            gameCanvas.screenHeight/2);
    }
}
}

```

Для расчета расстояния между шаром и кораблем я использовал конструкцию исходного кода, которая создает вокруг шара ограничивающий прямоугольник (рис. 13.18). Если корабль не входит в зону этого ограничивающего прямоугольника, то шар остается без движения. Как только корабль попадет в зону действия ограничивающего прямоугольника, шар начинает атаковать и приближаться к кораблю.

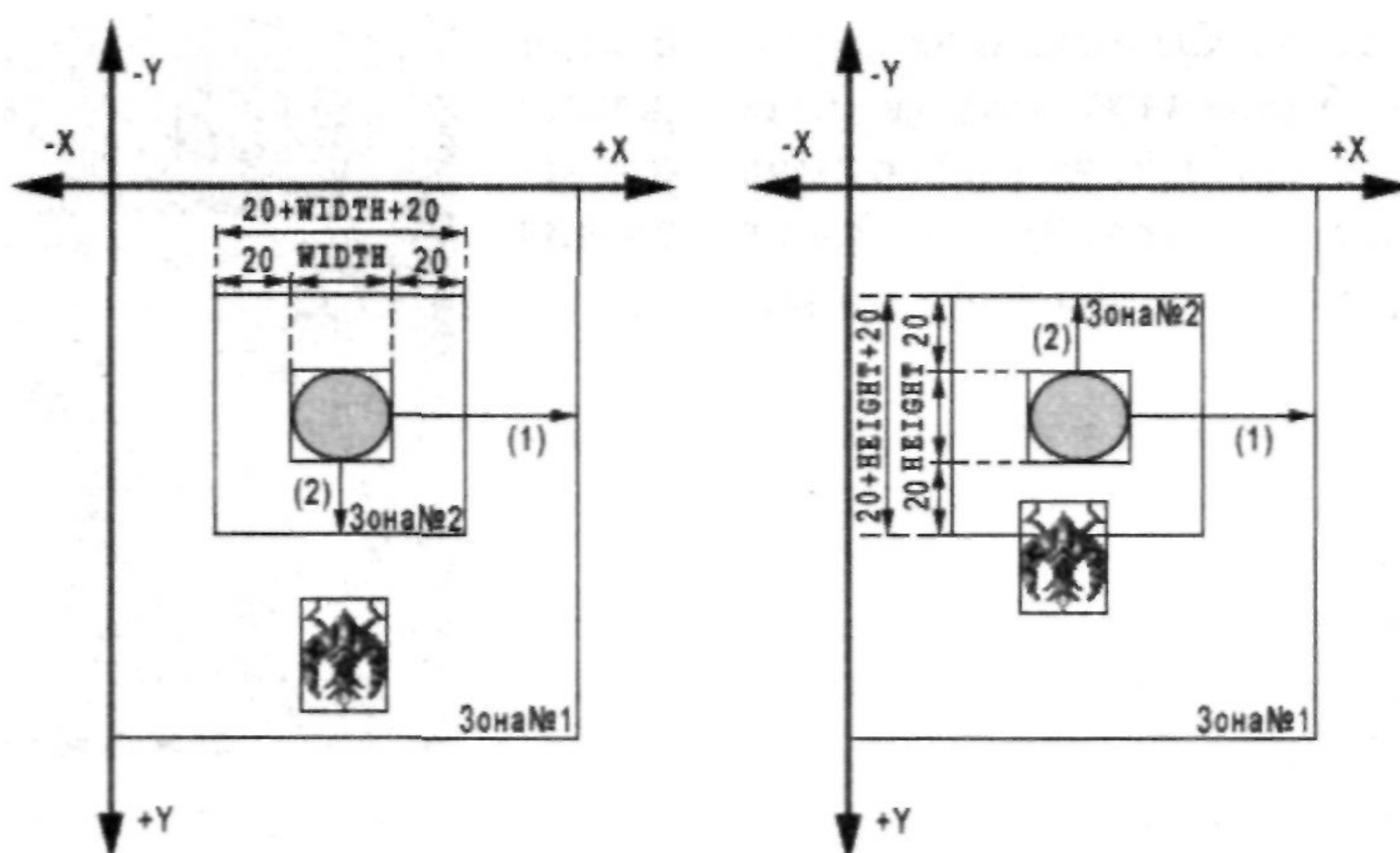


Рис. 13.18. Смена состояний шара

Создание ограничивающего прямоугольника построено на базе использования методов `getX()` и `getY()`. Посмотрите на рис. 13.18, где показаны

размеры ограничивающего прямоугольника. Как видно из рисунка, прямоугольник по ширине равен в пикселях размеру:

20 (или `ball.getX() - 20`) + ширина одного фрейма шара + 20 (или `ball.WIDTH + 20`)

По высоте:

20 (или `ball.getY() - 20`) + высота одного фрейма шара + 20 (или `ball.HEIGHT + 20`)

Для проверки попадания корабля в зону действия состояния № 1 используется следующая конструкция исходного кода:

```
if (this.getX() - 20 < sprite.getX() + sprite.getWidth()
    && this.getY() - 20 < sprite.getY() + sprite.getHeight()
    && this.getX() + this.WIDTH + 20 > sprite.getX()
    && this.getY() + this.HEIGHT + 20 > sprite.getY()) { ... }
```

В этом коде происходят построение ограничивающего прямоугольника вокруг шара и постоянное сравнение позиции корабля с границами созданного прямоугольника. Когда корабль не попадает в зону действия прямоугольника, скорость шара равна 0, а как только корабль находится в зоне действия прямоугольника, шар начинает двигаться к кораблю со скоростью в 1 пиксель за игровой цикл.

Запустите демонстрационный пример AI8 с компакт-диска и посмотрите на работу программы. Для того чтобы ограничивающий прямоугольник вам было видно, в методе `draw()` класса `MainGameCanvas` был написан исходный код, использующий метод `drawRect()` для рисования на экране прямоугольника (рис. 13.19).

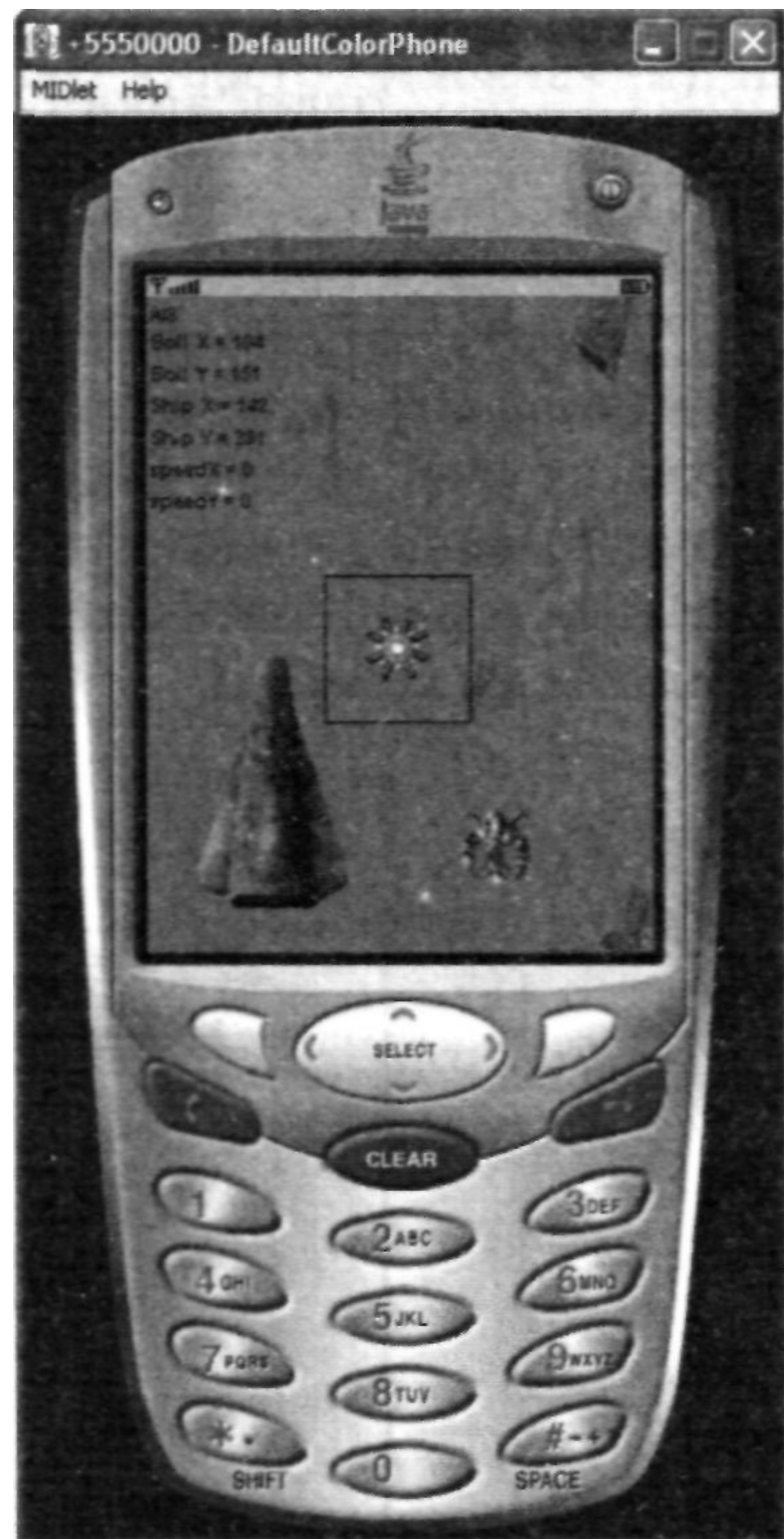


Рис. 13.19. Распределенная логика смены состояний объекта в действии



<http://palata-x.narod.ru>

Глава 14. Движение спрайтов в пространстве

Мы подошли к финишной прямой в этапах реализации игры «Метеоритный дождь». Эта и следующая главы будут содержать много достаточно сложного материала для начинающих программистов, так что самое время не расслабляться и уделить прочтению этих глав столько времени, сколько понадобится для понимания модификаций исходного кода игры.

В этой главе в игру «Метеоритный дождь» добавятся метеориты, летящие навстречу кораблю. Для создания в программе метеоритов будет создан отдельный класс `Meteorite` и загружены два графических изображения. Оба изображения имеют анимационную последовательность, состоящую из пяти фреймов. Два этих рисунка метеоритов идентичны друг другу по итоговому изображению и набору фреймов, но отличаются по своей цветовой гамме.

Вторым важным нововведением игры станет реализация механизма стрельбы корабля. В момент полета корабль будет производить выстрелы в летящие ему навстречу метеориты, и таким образом, кроме возможности уклоняться от метеоритов, корабль приобретет способность их взрывать. На данном этапе это все, но уже в следующей главе мы приступим к изучению темы игровых столкновений.

14.1. Метеориты

Все метеориты в игре летят навстречу кораблю, то есть сверху вниз, двигаясь в том же направлении, что и игровая карта, но со значительно большей скоростью. В главе, посвященной основам искусственного интеллекта, вы изучили много различных алгоритмов движения объектов. В этой главе мы на практике используем один из видов перемещения спрайтов в пространстве. Логика движения метеоритов состоит в следующем.

В момент старта игры метеориты устанавливаются на игровые позиции за экраном телефона в его верхней части (отрицательная плоскость оси Y , как показано на рис. 14.1). Промежуток между метеоритами составит 200 пикселей. Как только игра стартует, все метеориты начинают перемещаться вниз навстречу кораблю со скоростью 15 пикселей. При достижении конца экрана в его нижней части каждый метеорит вновь «закидывается» вверх в отрицательную часть оси Y на 200 пикселей и опять перемещается вниз навстречу кораблю с той же скоростью. В итоге в игровом процессе создается циклическое движение метеоритов, которое будет продолжаться до окончания уровня игры или пока у корабля не закончатся жизни.

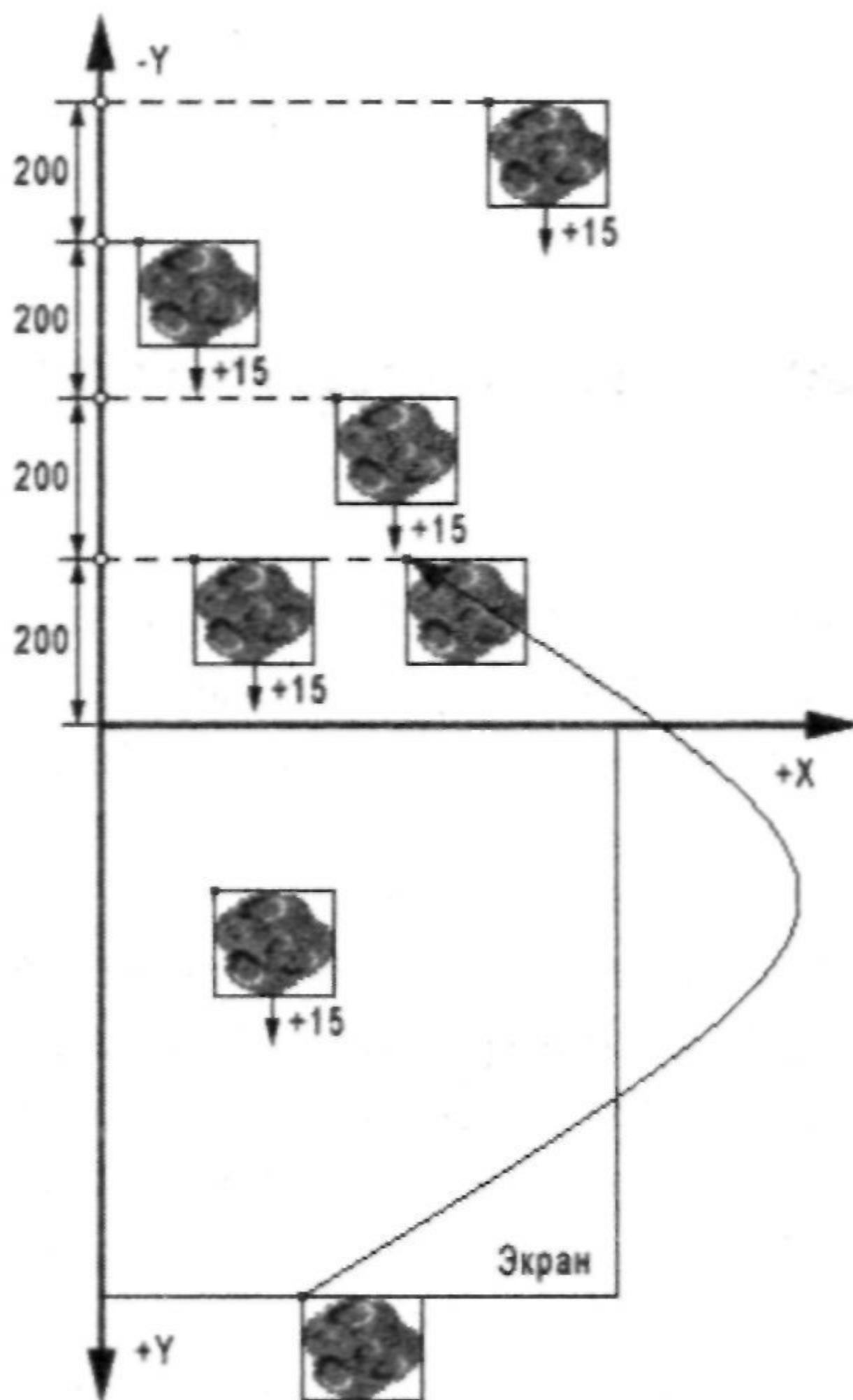


Рис. 14.1. Выбор позиции для метеоритов

14.1.1. Класс *Meteorite*

В игре любой значительный персонаж всегда лучше всего представлять отдельным классом. Впоследствии вашу наработку таких классов можно использовать в других своих играх, экономя тем самым время, силы и денежные ресурсы. В игре «Метеоритный дождь» метеориты представлены отдельным классом *Meteorite*. Давайте посмотрим на исходный код этого класса, пример которого находится в листинге 14.1.

```
/*
 * Meteorite.java
 * ЛИСТИНГ 14.1
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/*
 * @author Stanislav Gornakov
 * Aversion 1.0
 */
```

```
public class Meteorite extends Sprite{
    int xSpeed = 0;
    int WIDTH = 0;
    int HEIGHT = 0;

    public Meteorite(Image image, int frameWidth, int
    frameHeight) throws Exception {
        super(image, frameWidth, frameHeight);
        this.WIDTH = frameWidth;
        this.HEIGHT = frameHeight;
        defineCollisionRectangle(5, 5, WIDTH - 5, HEIGHT - 5);
        defineReferencePixel(this.WIDTH/2, HEIGHT/2);
    }

    void moveMeteorite(int ySpeed){
        this.move(xSpeed, ySpeed);
        this.nextFrame();
    }
}
```

Класс `Meteorite` наследует возможности системного класса `Sprite`, поэтому этому классу доступны все операции по анимации спрайтов и их движению в пространстве. Исходный код класса `Meteorite` не содержит новшеств, с которыми вы незнакомы, за исключением метода `defineCollisionRectangle()`. Этот метод необходим для определения более целенаправленного столкновения между объектами, и в следующей главе мы обязательно уделим внимание этому вопросу.

Единственный метод `moveMeteorite()` класса `Meteorite` задает движение метеоритам по оси `Y`. Скорость метеоритов передается в метод параметром `int ySpeed`. Сделано это специально, для того чтобы, например, на следующем уровне можно было увеличить скорость движения метеоритов, создав тем самым более сложное прохождение текущего уровня. В нашей игре данный механизм не использовался, но поскольку этот класс содержит некоторые части от моего большого игрового шаблона классов, то этот механизм был оставлен. Дополнительно можно передавать в метод скорость и по оси `X`, если в вашей игре предусмотрено перемещение объектов в пространстве по этой оси координат.

14.7.2. Движение метеоритов

Описав класс `Meteorite`, мы перемещаемся в исходный код класса `MainGameCanvas`, где нам необходимо создать объекты, представляющие метеориты, установить их на игровые позиции и двигать в пространстве. В области глобальных переменных класса `MainGameCanvas` объявим массив объектов класса `Meteorite`, а также создадим дополнительный объект класса `Random`.

```
private Meteorite [] meteor = new Meteorite[4];
private Random random = null;
```


В игре имеются четыре метеорита. Этого количества объектов вполне достаточно, чтобы равномерно заполнить экранное пространство телефона, поскольку все метеориты двигаются в цикличном режиме. Кроме массива объектов, в области глобальных переменных объявляется объект класса `Random`. С помощью объекта этого класса в программном коде будет генерироваться последовательность случайных чисел, необходимая для выбора метеоритам случайной позиции в игровом пространстве. Подробнее об этом механизме позже в этой главе.

Примечание. Для работы с классом `Random` необходимо подключить в исходный код программы пакет `java.util`.

В конструкторе класса `MainGameCanvas` происходит создание объекта `random`,

```
random = new Random();
```

Далее нам необходимо создать в исходном коде класса `MainGameCanvas` массив объектов класса `Meteorite`, загрузив попутно для каждого объекта свое графическое изображение. Эти действия мы выполняем в методе `createGame()`.

```
//*****
// метеорит
//*****
try{
    Image imageMeteorite1 = Image.createImage("/
Meteorite1.png");
    Image imageMeteorite2 = Image.createImage("/
Meteorite2.png");
    meteor[0] = new Meteorite(imageMeteorite1, 40, 35);
    meteor[1] = new Meteorite(imageMeteorite1, 40, 35);
    meteor[2] = new Meteorite(imageMeteorite2, 40, 35);
    meteor[3] = new Meteorite(imageMeteorite2, 40, 35);
    imageMeteorite1 = null;
    imageMeteorite2 = null;
}catch (Exception ex) {
    System.err.println("meteor images are not loaded");
}
```

Для создания массива объектов с одним-единственным графическим изображением можно применять цикл, но у нас два разных рисунка, поэтому цикл не используется. Создав объект класса `Meteorite`, можно переходить в метод `setGame()` для определения игровых позиций метеоритам и их дальнейшего добавления в игру.

```
//
//*****
// устанавливаем метеориты на позиции
//
```



```
*****
meteor[0].setPosition(Math.abs(random.nextInt())
%(screenWidth -
meteor[0].WIDTH)), -100);
meteor[1].setPosition(Math.abs(random.nextInt())    %
(screenWidth -
meteor[1].WIDTH)), -400);
meteor[2].setPosition(Math.abs(random.nextInt())    %
(screenWidth -
meteor[2].WIDTH)), -700);
meteor[3].setPosition(Math.abs(random.nextInt())    %
(screenWidth -
meteor[3].WIDTH)), -1000);

// добавляем метеориты поверх карты и корабля
for(int i = 4; -i >= 0;){
gameManager.append(meteor[i]);
}
```

В определении игровых позиций по оси Y активное участие принимает метод `nextInt()`. Этот метод позволяет генерировать последовательность случайных чисел в заданном диапазоне. Данный диапазон задается с помощью двух параметров метода, где первый параметр - это минимальное значение, а второй параметр - максимальное значение. У нас ширина экрана 240 пикселей, и именно в этом диапазоне определяются позиции объектов по оси X (рис. 14.2).

Чтобы спрайт не выходил за пределы экрана, с правой стороны от ширины дисплея отнимается ширина одного фрейма графического изображения метеорита. Таким образом, при каждом новом старте игры, а также после исчезновения метеоритов с экрана объекты будут устанавливаться на новые позиции. В целом такой подход в играх не дает игроку возможности запомнить позицию объекта в игровом пространстве, внося в игру постоянный элемент неожиданности.

Для движения метеоритов по экрану применяется следующая конструкция исходного кода.

```
// *****
// движение метеорита
// *****
for (int i = 4; -i >= 0; ) {
    meteor[i].moveMeteorite(15);
    if(this.screenHeight < meteor[i].getY()){
        meteor[i].setPosition(Math.abs(random.nextInt())    %
(screenWidth -
    meteor[i].WIDTH)), -200);
    }
}
```

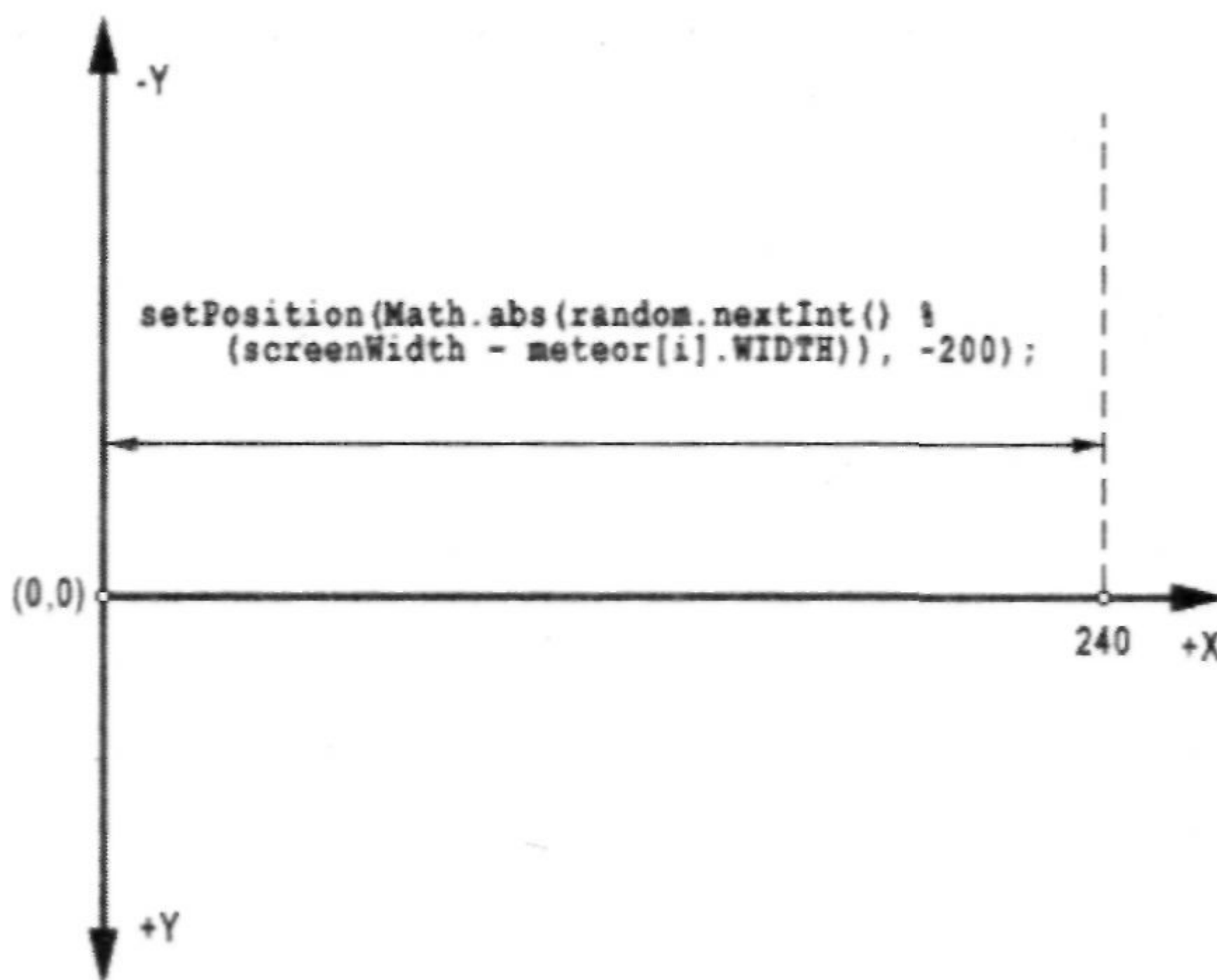


Рис. 14.2. Выбор случайной позиции на экране

В этом блоке создается простой цикл, где все элементы массива данных перемещаются сверху вниз со скоростью 15 пикселей. Дополнительно с помощью конструкции `if/else` организуется проверка выхода объекта за пределы области экрана с его нижней стороны. Как только спрайты исчезают с экрана, то они устанавливаются на новую позицию в отрицательной плоскости оси Y .

Что касается цикла `for` и, в частности, строки кода

```
for (int i = 4; -i >= 0; ) ,
```

то это один из вариантов этого цикла, который увеличивает работу цикла примерно на 10-15%. Это достоверная информация, проверенная на практике не один раз. Во всех своих мобильных, компьютерных и консольных играх я использую такую конструкцию цикла `for`. На этом мы изучили все то, что касается создания и движения метеоритов в игре, переходим к главному персонажу игры.

14.2. Реализуем стрельбу корабля

Любой космический корабль должен стрелять, и наш корабль не исключение. Для реализации выстрелов нам понадобятся один графический файл с изображением пули и новый класс `Shot`. Алгоритм действий по созданию системы уничтожения метеоритов заключается в следующем.

Создается новый класс `Shot`, в котором мы описываем свойства пули, а в частности загрузку изображения пули в игру и определение ее скорости движения на экране. Затем в классе `Ship` создается специальный метод, генерирующий с заданной периодичностью обойму пуль, поскольку графический файл у нас один, а нам необходимо создать иллюзию целой обоймы пуль. Далее уже непосредственно в классе `MainGameCanvas` формируется обойма пуль и производятся выстрелы, но обо всем по порядку.

14.2.1. Класс *Shot*

Простой код класса `Shot`, приведенный в листинге 14.2, иллюстрирует схожесть с классом метеоритов и корабля.

```
/*
 * Shot.Java
 * Листинг 14.2
 *
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/*
 * @author Stanislav Gornakov
 * Aversion 1.0
 */

public class Shot extends Sprite {

    // скорость по оси X
    int speedX = 0;
    // скорость по оси Y
    int speedY = 0 ;

    public Shot(Image image) {
        super(image);
    }

    public void shotShip() {
        this.move(speedX, -2) ;
    }
}
```

В этом классе имеется один ключевой метод, который отвечает за перемещение пуля в пространстве со скоростью два пикселя за один игровой проход. Движение пуля происходит по оси `Y` снизу вверх, поэтому в качестве значения скорости используется отрицательное значение.

14.2.2. Класс *Ship*

В классе `Ship` необходимо создать метод, генерирующий обойму пуля. Для реализации этой идеи понадобятся несколько дополнительных переменных. Посмотрите на

исходный код класса Ship, представленный в листинге 14.3. Все нововведения в коде выделены жирным шрифтом.

```
/*
 * Ship.java
 * ЛИСТИНГ 14.3
 *
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/*
 * @author Stanislav Gornakov
 * (Aversion 1.0
 */

public class Ship extends Sprite{
private MainGameCanvas gameCanvas = null;
static final int WIDTH = 46;
static final int HEIGHT = 51;
Shot shot = null;
boolean shootingShip = false;
private Image imageShot = null;
long startTime = 0;
long endTime = 0;

public Ship(MainGameCanvas gameCanvas, Image image, int
frameWidth,
int frameHeight) throws Exception {
    super(image, frameWidth, frameHeight);
this.shootingShip = shootingShip;
    this.gameCanvas = gameCanvas;
    defineCollisionRectangle(5, 5, WIDTH - 5, HEIGHT - 5);
try{
        imageShot = Image.createImage("/Shot.png");
    }catch (Exception ex) {
        System.err.println("Shot.png it is not loaded");
    }
}

public void moveShip (int direction) {...}

public Shot createShot(int updateTime) throws Exception {
```



```
        startTime = (System.currentTimeMillis() - endTime);
        if (!shootingShip && startTime > updateTime) {
            shot = new Shot(imageShot);
            shot.setPosition(this.getX() + (WIDTH/2) - 4,
this.getY());
            endTime = System.currentTimeMillis();
        } else {
            shootingShip = false;
        }
        return shot;
    }
}
```

Новый метод `createShot()` - это и есть генератор обоймы пуль. Механика работы этого метода построена по следующему принципу. В момент вызова `createShot()` в тело метода передается заданное значение времени, исчисляемое в миллисекундах (у нас это значение равно 700 миллисекундам). На основе этого значения будет производиться расчет периодичности создания и выстрела новой пули из обоймы. В дальнейшем, например на каждом новом уровне или при подборе нового оружия, это значение можно уменьшать, а соответственно, увеличивать скорость создания и выстрела пуль. В качестве позиции для выхода пули на экран выбирается точка:

```
shot.setPosition(this.getX() + (WIDTH/2) - 4, this.getY());
```

По оси X мы получаем середину одного фрейма графического изображения корабля `this.getX() + (WIDTH/2)`, а четыре дополнительных отнимаемых пикселя от приведенного значения - это середина графического изображения пули. Можно заменить целочисленное значение, равное четырем пикселям, и на такую запись `shot.WIDTH/2`, но это возможно при условии создания переменной `WIDTH` в исходном коде класса `Shot`. Теперь нам необходимо создать обойму пуль в коде класса `MainGameCanvas` и заставить их двигаться по экрану.

14.2.3. Движение пуль

В мобильных играх, в отличие от компьютерных и консольных игр, порой не всегда удобно одновременно управлять главным персонажем игры и стрелять. Хорошо, когда в телефоне система управления реализована как в смартфоне N-Gage или N-Gage QD, тогда проблем с одновременным выполнением нескольких действий не возникнет. На простом телефоне управлять кораблем с помощью джойстика и все с помощью того же джойстика стрелять (нажатие в центр джойстика) очень трудно.

Конечно, можно для стрельбы использовать клавишу за номером пять, но все равно это очень не удобно, кто играл по-настоящему и долго, тот знает, о чем я говорю. В связи с этим я настоятельно рекомендую вам производить стрельбу в автоматическом режиме либо предусмотреть выбор переключения между автоматической и ручной стрельбой.

В игре «Метеоритный дождь» применяется автоматическая стрельба. Для этого метод `firingShip()`, отвечающий за перемещение пуль в игре, постоянно вызывается в игровом цикле класса `MainGameCanvas`. Если вы хотите создать ручной режим стрельбы, то назначьте вызов метода `firingShip()` на нажатие одной из клавиш телефона. Чаще всего для этих целей используется команда **Fire**, которая автоматически назначает выстрелы на нажатие джойстика по центру и на клавишу под номером пять.

Сама схема создания и движения пуль в игре несложна, и думается, вы сможете разобраться с ней самостоятельно. Исходный код класса предложен вам в листинге 14.4. Полный исходный код изученного проекта находится на компакт-диске в папке **\Code\Chapter14**.

```
/*
 * MainGameCanvas.java
 * Листинг 14.4
 *
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import java.lang.*;
import java.util.*;
import java.io.*;

/**
 *
 * @author Stanislav Gornakov
 * (Aversion 1.0
 */

public class MainGameCanvas extends GameCanvas implements
Runnable {
private GameMidlet midlet = null;
private Graphics graphics = null;
private volatile Thread animationThread = null;
private LayerManager gameManager = null;
private Background background = null;
private Ship ship = null;
private int fps = 50;
int screenWidth = 0;
int screenHeight = 0;
int gameState = 0;
private Image imageGameContinue = null;
private Image imageComandContinue = null;
private Image imageComandMenu = null;
```

```
private Meteorite [] meteor = new Meteorite [4];
private Shot runShotShip = null;
private Shot shootingShip = null;
private Vector shotShip = null;
int updateShot = 0;
private Random random = null;

public MainGameCanvas(GameMidlet midlet) throws Exception
{
    super(true);
    this.midlet = midlet;
    setFullScreenMode(true);
    graphics = getGraphics();
    screenWidth = this.getWidth();
    screenHeight = this.getHeight();
    gameManager = new LayerManager();
    random = new Random();
    createGame();
}

public void start() {
    animationThread = new Thread(this);
    animationThread.start();
}

public void run() {
    Thread currentThread = Thread.currentThread();
    try {
        while (currentThread == animationThread) {
            long startTime = System.currentTimeMillis();
            if (isShown()) {
                updateGame();
                draw();
                flushGraphics();
            }
            long endTime = System.currentTimeMillis() -
startTime;
            if (endTime < fps) {
                synchronized (this) {
                    wait(fps - endTime);
                }
            } else {
                currentThread.yield();
            }
        }
    }
}
```

```
        }
    } catch (InterruptedException ie) {
    }
}

public void stop() {
    animationThread = null;
}

public void keyPressed(int keyCode) {
    switch(gameState) {

    case 0:

        if(keyCode == -6 || keyCode == -7) {
            this.stop();
            midlet.gameMenu();
        }
        break;

    case 1:

        if(keyCode == -6) {
            this.stop ();
            midlet.gameMenu();
        }
        if(keyCode == -7) {
            this.stop ();
            midlet.loadingGame();
        }
        break;
    }
}

private void draw() {
    graphics.setColor(250, 250, 250);
    graphics.fillRect(0, 0, screenWidth, screenHeight);
    gameManager.paint(graphics, 0, 0);

    switch(gameState) {

    case 1:

        graphics.drawImage(imageGameContinue,
```



```
screenWidth/2, screenHeight/2,
    Graphics.VCENTER |
    Graphics.HCENTER);
    graphics.drawImage(imageComandMenu, 0,
screenHeight-
    imageComandMenu.getHeight(), 0);
    graphics.drawImage(imageComandContinue,
screenWidth-
    imageComandContinue.getWidth(),
    screenHeight-imageComandContinue.getHeight(), 0);

    break;

}

}
```

```
public void createGame() throws Exception {
    //*****
    // карта
    //*****
    try{
        Image imageBackground = Image.createImage("/
Background.png");
        background = new
Background(Background.COLUMNS_WIDTH,
        Background.ROWS_HEIGHT, imageBackground,
Background.WIDTH,
        Background.HEIGHT);
        background.setVisible(false);
    }catch (Exception ex) {
        System.err.println("Background it is not
loaded");
    }

    //*****
    // изображения
    //*****
    try{
        imageGameContinue = Image.createImage("/
Continue.png");
        imageComandMenu = Image.createImage("/
comandMenu.png");
```

```

        imageComandContinue = Image.createImage("/
comandContinue.png");
    }catch (Exception ex) {
        System.err.println("Image it is not loaded");
    }

//*****
// корабль
//*****
try{
    Image imageShip = Image.createImage("/Ship.png");
    ship = new Ship(this, imageShip, Ship.WIDTH,
Ship.HEIGHT);
    ship.setVisible(false);
    imageShip = null;
}catch (Exception ex) {
    System.err.println("Ship.png it is not loaded");
}
// ===== пули для ship =====
shotShip = new Vector();

//*****
// метеорит
//*****
try{
    Image imageMeteorite1 = Image.createImage("/
Meteorite1.png");
    Image imageMeteorite2 = Image.createImage("/
Meteorite2.png");
    meteor[0] = new Meteorite(imageMeteorite1, 40, 35);
    meteor[1] = new Meteorite(imageMeteorite1, 40, 35);
    meteor[2] = new Meteorite(imageMeteorite2, 40, 35);
    meteor[3] = new Meteorite(imageMeteorite2, 40, 35);
    imageMeteorite1 = null;
    imageMeteorite2 = null;
}catch (Exception ex) {
    System.err.println("meteor images are not
loaded");
}
}

public void setGame(){
    gameState = 0 ;

```

```
//*****
// background
//*****

background.createBackground();
background.setPosition(0, -background.getHeight() +
screenHeight);
background.setVisible(true);

//*****
// ship
//*****

ship.setPosition(screenWidth/2 - ship.WIDTH/2,
screenHeight - ship.HEIGHT - 10);
ship.setVisible(true);
updateShot = 700;

//*****
// устанавливаем метеориты на позиции
//*****

meteor[0].setPosition(Math.abs(random.nextInt() %
(screenWidth -
meteor[0].WIDTH)), -100);
meteor[1].setPosition(Math.abs(random.nextInt() %
(screenWidth -
meteor[1].WIDTH)), -400);
meteor[2].setPosition(Math.abs(random.nextInt() %
(screenWidth -
meteor[2].WIDTH)), -700);
meteor[3].setPosition(Math.abs(random.nextInt() %
(screenWidth -
meteor[3].WIDTH)), -1000);

//*****
// добавляем игровые компоненты
//*****

for (int i = 4; -i >= 0; i--) {
    gameManager.append(meteor[i]);
}
gameManager.append(ship);
gameManager.append(background);

}

public void updateGame() {
```

```

switch(gameState) {

case 0:
    //*****
    //  клавиши
    //*****
    int keyStates = getKeyStates();
    //  вниз
    if ((keyStates & DOWN_PRESSED) != 0) {
        ship.moveShip(1);
    //  вверх
    } else if ((keyStates & UP_PRESSED) != 0) {
        ship.moveShip(2);
    //  влево
    } else if ((keyStates & LEFT_PRESSED) != 0) {
        ship.moveShip(3);
    //  вправо
    } else if ((keyStates & RIGHT_PRESSED) != 0) {
        ship.moveShip(4);
    //  stop
    } else {
        ship.moveShip(0);
    }

    //*****
    //  движение метеорита
    //*****
    for (int i = 4; -i >= 0;) {
        meteor[i].moveMeteorite(15);
        if(this.screenHeight < meteor[i].getY()){
        meteor[i].setPosition(Math.abs(random.nextInt() %
            (screenWidth - meteor[i].WIDTH)), -200);
        }
    }

    //*****
    //  стрельба
    //*****
    try {
        firingShip ();
    } catch (Exception exc) {
        System.err.println("Not fire");
    }
    moveShotShip();

```


<http://palata-x.narod.ru>

Глава 15. Игровые столкновения

Любая игра изобилует множеством различных пересечений игровых персонажей. Как правило, такие пересечения должны повлечь за собой назначенные действия. В некоторых случаях это ведет к потере жизни игрока, а в некоторых случаях - наоборот, к увеличению запаса жизни. Здесь все зависит от того, с каким из объектов игрок столкнулся и что за события в связи с этим должны произойти. Если игрок подбирает снаряды, жизни, провиант и т. д., то, соответственно, его личный запас энергии возрастает. Но если в главного героя попадает пуля, бомба или он коснулся нежелательного объекта, то происходят обратные действия, направленные на уменьшение его жизненной энергии, а в некоторых случаях - провианта и т. д.

Все эти описанные варианты игровых событий имеют одну общую составляющую. Это игровые столкновения между объектами, которые влекут за собой выполнение определенных действий. Вы как программист на каждое такое игровое столкновение должны задать некоторые события, которые являются основой всей логической составляющей вашей игры.

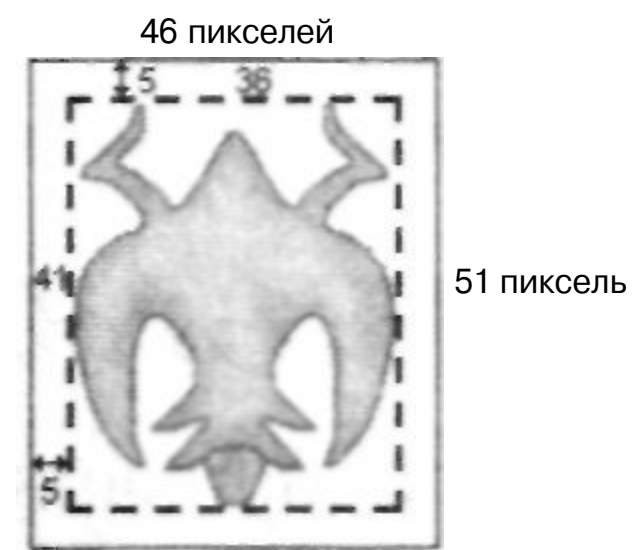
В игре «Метеоритный дождь» немного игровых персонажей, поэтому имеются всего два вида столкновений. Это столкновение между метеоритом и кораблем и столкновение метеорита с пулями. В момент столкновения метеорита и корабля в игре происходят взрыв метеорита и уменьшение жизненной энергии корабля. Если же пуля сталкивается с метеоритом, то метеорит и пуля взрываются, а игроку начисляются бонусные очки. Переходим к работе с кодом, начнем с создания методов для обработки столкновений в игре.

15.1. Пишем код для обработки игровых столкновений

В главе 8 вы получили максимум информации по игровым столкновениям. Сейчас наша задача заключается в использовании этого багажа знаний для создания эффективного механизма обработки столкновений. Но прежде, как и было обещано в предыдущей главе, мы рассмотрим назначение метода `defineCollisionRectangle()`, который вызывается в конструкторах класса `Ship`.

Этот метод позволяет переопределить размеры (в данном случае корабля) при столкновении с другими игровыми объектами. То есть фактически с помощью этого метода можно увеличить или уменьшить зону столкновения корабля с препятствиями. Посмотрите на рис. 15.1, где показано, как была переопределена зона столкновения корабля.

Переопределив размеры ограничивающего прямоугольника, используемого при столкновении объектов, мы добавляем в игровой процесс реалистичность.



определяем только изменение положения метеорита на экране и ничего более, но далее в этот метод добавим еще пару строк кода.

15.1.2. Столкновение пуль и метеоритов

Столкновение корабля с метеоритом выглядит достаточно простым, поскольку в столкновении принимают участие два объекта класса `Sprite`. Обработка столкновения метеорита и пули выглядит несколько серьезнее, так как для реализации в игре обоймы пуль был использован массив данных, представленный классом `Vector`.

Этот класс, как вы помните, имеет возможность автоматически увеличивать или уменьшать данные массива. В связи с этим обработка столкновений пуль и метеорита состоит сразу из двух методов, связанных между собой неразрывно. Посмотрите на оба этих метода.

```
boolean overlapsShipMeteor(Sprite sprite) {
    return collidesShotMeteor(sprite, shotShip.size());
}
boolean collidesShotMeteor(Sprite sprite, int count) {
    for (int i = 0; i < count; i++) {
        Shot m = (Shot)(shotShip.elementAt(i));
        if (sprite.collidesWith(m, true)) {
            m.setVisible(false);
            sprite.setPosition(Math.abs(random.nextInt() %
(screenWidth - 50)), -200);
        }
    }
    return true;
}
```

Наша главная задача заключается в определении, какая из пуль или какой из элементов массива, представленный классом `Vector`, столкнулся с метеоритом. Для этих целей создан метод `collidesShotMeteor()`, принцип работы которого состоит в следующем.

В метод передаются два параметра - игровой объект и целочисленное значение. Целочисленное значение в конкретный промежуток времени представляет количество элементов в массиве данных или количество пуль, летящих (активных в данный момент) навстречу метеориту. Цикл `for` перебирает все элементы массива, а метод `shotShip.elementAt()` выбирает текущую активную пулю. Затем при помощи вызова метода `collidesWith()` программа узнает, какая из выстрелянных пуль столкнулась с метеоритом. Если столкновение имеет место, то методом `setVisible(false)` мы скрываем пулю на экране телефона, а метеорит устанавливаем на новую игровую позицию.

Далее метод `collidesShotMeteor()` мы вызываем в методе `overlapsShipMeteor()`, и такая поэтапная конструкция методов позволяет определить столкновение метеоритов с массивом пуль, представляемых клас-

сом `Vector`. Элегантная конструкция кода, которую вы можете модернизировать под свои нужды и использовать в мобильных играх.

Рассмотренные методы обработки столкновений мы создаем непосредственно в классе `MainGameCanvas`, а затем вызываем их в игровом цикле на каждой новой итерации. Поскольку мы имеем дело с массивом данных, то для обработки столкновений применяется следующий блок кода:

```
for(int i = 4; -i >= 0;){
    overlapsShipMeteor(meteor[i]);
    checkCollisions(meteor[i]);
}
```

15.2. Рисуем на экране взрывы

На текущий момент при всех игровых столкновениях метеориты убираются с экрана на новые позиции. Это не очень красиво, да и не логично, поэтому в игру необходимо добавить визуальных эффектов, а точнее взрывов. Для представления взрыва в игре рисуется отдельное графическое изображение в виде последовательного набора фреймов, имитирующего взрыв объекта. На рис. 15.3 представлен взрыв из нашей игры. Это изображение состоит из четырех фреймов анимационной последовательности. Как мы будем использовать это изображение?

Как только в игре произойдет столкновение, в этот момент нам нужно показать на экране телефона один за другим четыре фрейма взрыва, а затем убрать изображение с экрана. То есть необходимо создать такой механизм, который в момент взрыва показывал бы на экране анимацию, состоящую из четырех фреймов, а затем убирал бы графическое изображение взрыва с экрана. Для этих целей в проекте создадим новый класс `Explosion`, в котором опишем будущий взрыв. Исходный код класса представлен в листинге 15.1.



Рис. 15.3. Изображение взрыва

```
/*
 * Explosion.java
 * Листинг 15.1
 *
 */
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**
 *
 * @author Stanislav Gornakov
 * Aversion 1.0
 */
public class Explosion extends Sprite{
```

```
private int[] sequence = {0, 1, 2, 3};

public Explosion(Image image, int frameWidth, int
frameHeight)throws Exception {
    super(image, frameWidth, frameHeight);
}

// взрыв
void updateExplosion(){
    if (this.getFrame () < 3) {
        this.nextFrame();
    }
    else{
        this.setVisible(false);
    }
}

// выбор позиции для взрыва
void explosion (Sprite sprite){
    this.setFrameSequence(sequence);
    this.setPosition(sprite.getX(), sprite.getY());
    this.setVisible(true);
}
}
```

В исходном коде класса `Explosion` создается массив `sequence`. Этот массив содержит количество фреймов анимационной последовательности изображения взрыва. Количество элементов массива, или количество фреймов, очень важно знать, потому что в дальнейшем на основе количества фреймов мы будем определять, убирать нам взрыв с экрана или нет.

Метод `updateExplosion()` как раз и нужен для определения окончания перебора фреймов анимационной последовательности. В этом методе используется конструкция `if/else`. В момент вызова этого метода происходит выполнение блока кода, следующего за оператором `if`. Метод `getFrame()` позволяет получить текущий фрейм, показанный на экране, и как только это значение будет больше, чем количество фреймов во всей анимации, то начинает выполняться блок кода, следующего за оператором `else`, где метод `this.setVisible(false)` скрывает показ анимации в игре.

Для выбора позиции показа взрыва на экране специально создан метод `explosion()`. В этот метод в качестве параметра передается взрываемый объект (в нашем случае метеорит), и на базе его текущих координат в пространстве определяется место вывода взрыва на экране.

Теперь давайте перейдем к исходному коду класса `MainGameCanvas`, где нам предстоит создать объект класса `Explosion`. В области глобальных переменных объявим новый объект этого класса.

```
private Explosion explosion = null;
```

Затем в методе `createGame()` необходимо создать объект и загрузить в него графическое изображение взрыва.

```
Image imageExplosion = Image.createImage("/
Explosion.png");
explosion = new Explosion(imageExplosion, 44, 44);
explosion.setVisible(false);
imageExplosion = null;
```

После создания объекта и загрузки изображения мы скрываем изображение методом `setVisible(false)`. Это очень важно! Сейчас нам не нужно показывать взрыв на экране, это необходимо делать только по команде, которую мы будем отдавать программе по ходу игрового процесса.

Последняя строка кода в этом блоке

```
imageExplosion = null
```

обнуляет объект класса `Image`. После того как мы воспользовались этим объектом, он нам больше не нужен. Такой подход в обнулении неиспользуемых объектов позволяет разгружать память телефона от ненужных данных. Далее с помощью менеджера слоев добавляем созданный объект `explosion` в игру.

```
gameManager.append(explosion);
```

Теперь у нас есть все, для того чтобы взрывать все подряд или часть из этого. В игровом цикле напишем следующие строки кода:

```
if(explosion.isVisible()){
    explosion.updateExplosion    ();
}
```

Алгоритм работы этого блока кода такой. Если `explosion.isVisible()`, или как только дана команда показать взрыв на экране, происходит вызов метода `explosion.updateExplosion()`, который перебирает фреймы анимации, то есть показывает взрыв на экране. А команду для показа взрывов мы даем методом `explosion.explosion(sprite)`, который вызываем в методах, отведенных для обработки столкновений на экране.

```
//=====
//  СТОЛКНОВЕНИЯ
//=====
// ship и meteorite
public void checkCollisions(Sprite sprite){
    if(ship.collidesWith(sprite, true)){
        if(!explosion.isVisible()) {
            explosion.explosion(sprite);
        }
        sprite.setPosition(Math.abs(random.nextInt()) %
```

```

(screenWidth - 50)) , -200);
    }
}
// shot ship и meteorite ,
boolean overlapsShipMeteor(Sprite sprite){
    return collidesShotMeteor(sprite, shotShip.size());
}
boolean collidesShotMeteor(Sprite sprite, int count){
    for (int i = 0; i < count; i++) {
        Shot m = (Shot)(shotShip.elementAt(i));
        if (sprite.collidesWith( m , true)) {
            m.setVisible(false);
            if(!explosion.isVisible()){
                explosion.explosion(sprite);
            }
            sprite.setPosition(Math.abs(random.nextInt())%
(screenWidth - 50)) , -200);
        }
    }
    return true;
}

```

Заметьте, что вызов метода `explosion, explosion(sprite)` происходит до смены позиции метеорита на экране. Дело в том, что работа метода `explosion, explosion(sprite)`, а точнее определение позиции вывода взрыва на экран, основана на получении текущих координат объекта. Так вот если вы будете вызывать этот метод после того, как уберете метеорит с экрана, то, соответственно, вы не увидите взрыва на экране. Взрыв будет показан, но он произойдет на новом месте метеорита, где-то в виртуальной реальности или в отрицательной плоскости оси Y.

15.3. Добавляем в игру подсчет набранных очков

В игре за каждый сбитый выстрелом метеорит начисляется определенное количество очков. Для этих целей в коде класса `MainGameCanvas` мы создаем новую переменную `score`. Объявим ее в области глобальных переменных.

```
int score = 0 ;
```

Затем в методе `setGame()` при старте каждого нового уровня будем ее обнулять.

```
score = 0;
```

После этого добавим счетчик очков в метод `collidesShotMeteor()`, где происходит столкновение пули и объекта. Этот метод - лакмусовая бумажка, определяющая попадание пули в цель.


```
boolean collidesShotMeteor(Sprite sprite,int count)
    for (int i = 0; i < count; i++) {
        Shot m = (Shot)(shotShip.elementAt(i));
        if(sprite.collidesWith( m, true)){
            score +=10;
            m.setVisible(false);
            if(!explosion.isVisible()){
                explosion.explosion(sprite);
            }
            sprite.setPosition(Math.abs(random.nextInt()%(
screenWidth - 50)),-200);
        }
    }
    return true;
}
```

Затем в конце уровня или в конце игры мы будем выводить набранные очки на экран, но об этом чуть позже в этой главе.

15.4. Механизм подсчета жизненной энергии корабля

Сейчас окончанию уровня в игре соответствует окончание игровой карты, но по пути своего полета к завершению очередной миссии корабль неизбежно будет сталкиваться с метеоритами. Поэтому в игру нужно обязательно добавить механизм подсчета жизней главного героя, и как только у корабля не останется запаса жизненной энергии, нужно досрочно остановить игровой процесс.

Для этих целей в исходном коде класса `Ship` создается новая переменная `lifeShip`. Эта переменная представляет жизненную энергию корабля.

```
int lifeShip = 0;
```

Затем в методе `setGame()` уже класса `MainGameCanvas` при старте каждого нового уровня мы будем инициализировать эту переменную заданным значением.

```
ship.lifeShip = 100;
```

Здесь мы отводим игроку запас жизненной энергии в 100 виртуальных пунктов. При этом на каждом новом уровне эти 100 пунктов будут начисляться заново. Если вы желаете, чтобы игрок продолжал играть на следующем уровне с остатком жизни, то инициализация переменной `lifeShip` должна происходить только при старте игры или при повторном запуске не пройденного игроком уровня по случаю уничтожения корабля.

Отслеживать уменьшение жизненной энергии корабля необходимо в методе, определяющем столкновение метеорита с кораблем.

```
public void checkCollisions(Sprite sprite){
```

```

    if (ship.collidesWith(sprite, true)) {
        ship.lifeShip -= 20;
        if (!explosion.isVisible()) {
            explosion.explosion (sprite);
        }
        sprite.setPosition(Math.abs(random.nextInt() %
(screenWidth - 50)), -200);
    }
}

```

В данном случае за каждое новое столкновение с метеоритом у игрока забирается 20 пунктов. Это означает, что за весь уровень игрок может столкнуться с метеоритом всего пять раз. К выбору таких и подобных значений в игре необходимо подходить очень ответственно! Прежде чем убавлять и набавлять заданное количество энергии, нужно достаточно долго поиграть в игру на реальном мобильном телефоне, а не на эмуляторе, и только на основе многократного тестирования игры решать, какие значения использовать для отбора или начисления жизненной энергии.

Что касается начисления жизней, то можно «бросить» в игру движущийся объект (например, летающую аптечку), подобрав который корабль пополнит свои запасы жизненной энергии. Аналогична ситуация и для подбора нового оружия. И в том, и в другом случае определением подбора одного из артефактов будет служить организация нового метода по обработке столкновений корабля и нового объекта.

Потом в игровом цикле класса `MainGameCanvas` мы создаем проверку следующего условия.

```

if (ship.lifeShip <= 0) gameState = 2;

```

В этом случае если жизней у корабля меньше или это число равно нулю, то присваиваем переменной `gameState` значение 2. Это новое состояние переменной, введенное нами в этой главе. Выбор данного состояния будет означать окончание игры по причине утраты жизненной энергии корабля. Соответственно, чтобы это состояние было работоспособным, необходимо создать на базе оператора `case 2`: в методах `draw()` и `keyPressed()` новые условия. Но предварительно мы загрузим в игру дополнительную табличку с уведомлением игрока об окончании игры (рис. 15.4).

```

private Image imageGameEnd = null;
imageGameEnd = Image.createImage("/GameEnd.png");

```

Затем в двух методах `draw()` и `keyPressed()` добавим новые метки.

```

// метод draw()

```

```

case 2:

```

```

    graphics.drawImage(imageGameEnd, screenWidth/2,
screenHeight/2,
Graphics.VCENTER | Graphics.HCENTER);

```

Рис. 15.4. Информационная табличка
окончания игры



```
graphics.drawImage(imageComandMenu, 0,
screenHeight-
imageComandMenu.getHeight(), 0);
graphics.drawImage(imageComandContinue, screenWidth-
imageComandContinue.getWidth(), screenHeight-
imageComandContinue.getHeight(), 0);
graphics.setFont(Font.getFont(Font.FACE_SYSTEM,
Font.STYLE_BOLD,
Font.SIZE_LARGE));
graphics.setColor(0, 0, 0);
graphics.drawString("" + score, 130, 150, 0);

break;
}

// метод keyPressed()
...
case 2:

    if(keyCode == -6){
        this.stop();
        midlet.gameMenu();
    }
    if(keyCode == -7){
        this.stop();
        midlet.loadingGame();
    }
break;
}
```

В методе `draw()` на экран выводится уже новая табличка, а также количество набранных очков. Две команды клавиш выбора остаются неизменными. В свою очередь, в методе `keyPressed()` происходит обработка нажатий клавиш выбора. На левую клавишу назначена команда перехода в меню, а на правую - продолжение игры.

15.5. Графическое представление жизненной энергии корабля на экране телефона

Антураж игрового процесса очень часто определяется наличием дополнительных графических элементов на экране. Это может быть красивая и интерактивная

табличка, где ведется подсчет очков и жизней, или спидометр, показывающий скорость машины в игре, и т. д. В игре «Метеоритный дождь» мы нарисует на экране табличку, в которой будем показывать количество оставшихся жизней у игрока. Методов реализации этой задачи очень много, поэтому давайте подробно разберемся со способом, который используется в нашей игре.

Специально для отображения жизней игрока была создана табличка в виде полоски длиной чуть более 100 пикселей. Внутренняя часть полоски изображения шириною в пять пикселей была вырезана с помощью программы Photoshop. Таким образом, если вывести табличку поверх игровой карты (а ее и нужно там выводить), то в этом вырезанном пространстве будут видны карта и другие игровые объекты, находящиеся в данный момент под табличкой.

Затем поверх игровой карты, но под табличкой, как раз в ее вырезанном отверстии размером в 100 пикселей по длине и 5 пикселей по высоте, мы рисуем средствами Java 2 ME прямоугольник, покрашенный красным цветом. Такие действия позволяют организовать нам графическую шкалу, которую будет видно через отверстие таблички, и даже если прямоугольник будет большим размером (незначительно большим), чем отверстие, то табличка все равно скроет эти огрехи, поскольку рисуется поверх красного прямоугольника (рис. 15.5).

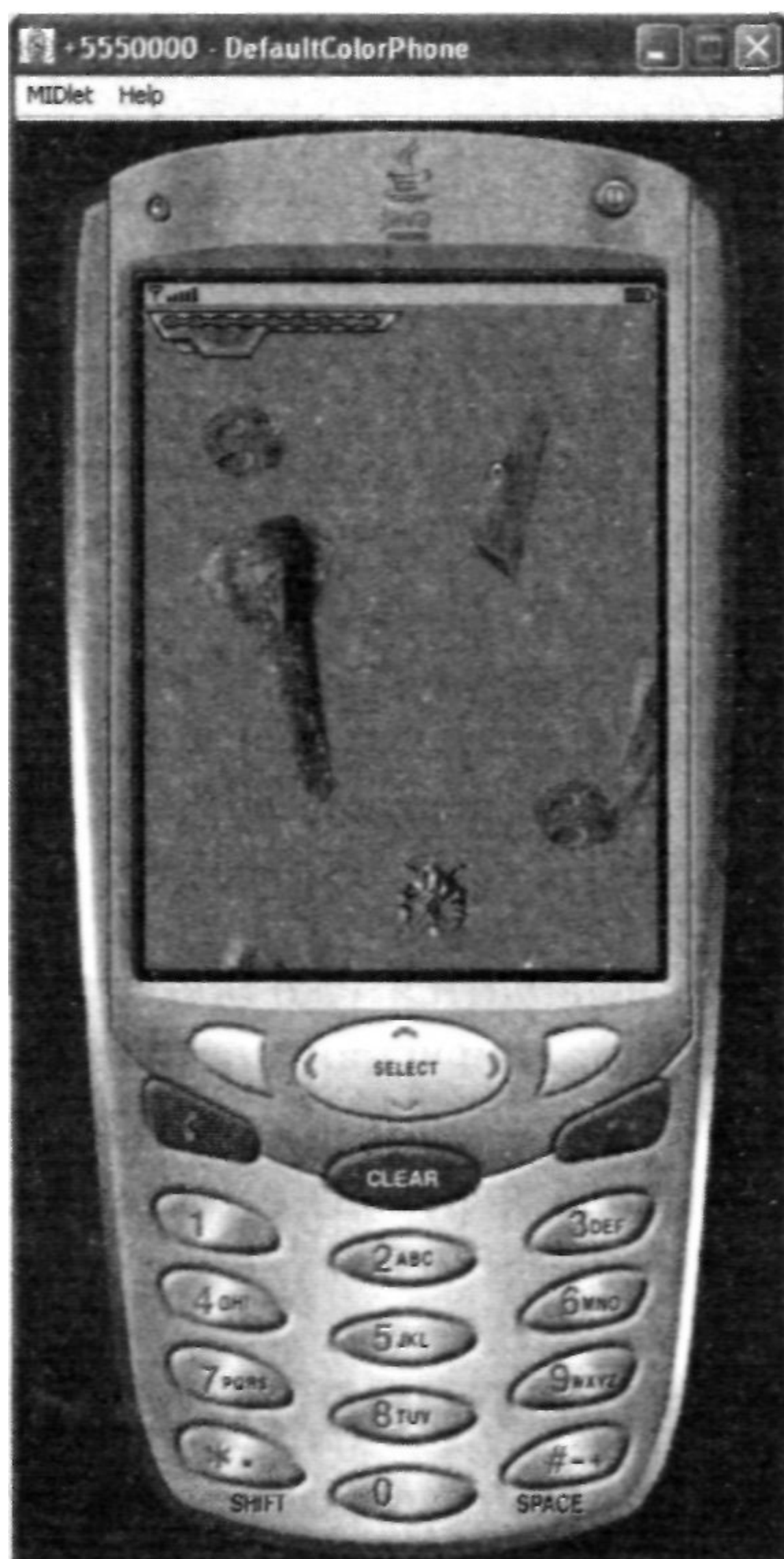


Рис. 15.5. Графическое представление жизненной энергии корабля на экране

Здесь вся трудность заключается только в подборе координат вывода для красного прямоугольника аккурат в отверстии таблички. Но если вы знаете точные размеры таблички и точное расположение отверстия, то вся сложность сводится к просчету необходимых координат. Теперь о том, как создать механизм рисования на экране красного прямоугольника и, что самое главное, как правильно его уменьшать.

Наши 100 пикселей по ширине прямоугольника - это и есть те самые 100 виртуальных пунктов жизненной энергии корабля, а значит, этот графический индикатор должен также уменьшаться при каждом столкновении корабля и метеорита на 20 пунктов.

Перейдем на время в исходный код класса `Ship` и создадим там новый метод `drawLifeShip()`, который будет рисовать на экране прямоугольник.

```
public void drawLifeShip(Graphics graphics, int x){
    graphics.setColor(250, 0, 0);
    graphics.fillRect(10, 4, x, 5);
}
```

Сначала мы устанавливаем цвет для прямоугольника (красный), а затем в методе `fillRect()` определяем место вывода прямоугольника на экран. Первые два параметра этого метода указывают программе точку вывода прямоугольника на экране, а третий параметр задает длину всего прямоугольника. Здесь используется переменная `x`, которая получает определенное значение при вызове этого метода в игровом процессе. Так вот в качестве значения в этот метод мы будем передавать значение переменной `ship.lifeShip`.

Изначально эта переменная равна 100 пунктам, но по мере попадания в корабль метеоритов значение этой переменной будет уменьшаться на 20 пунктов, значит, и сам прямоугольник тоже будет уменьшаться, а вместе с ним и наш графический индикатор жизненной энергии корабля.

Теперь перейдем в класс `MainGameCanvas`, загрузим в игру табличку и создадим индикатор жизней.

```
private Image lifeShip = null;
...
lifeShip = Image.createImage("/Life.png");
```

Затем в методе `draw()` нарисуем табличку на экране телефона и создадим красный прямоугольник, передавая в качестве параметра текущее значение жизни корабля.

```
ship.drawLifeShip(graphics, ship.lifeShip);
graphics.drawImage(lifeShip, 1, 2, 0);
```

Обратите внимание, что сначала на экране рисуется прямоугольник и поверх прямоугольника выводится табличка. Исходный код класса `MainGameCanvas` находится в листинге 15.2. Все нововведения выделены жирным шрифтом. Полный исходный код примера вы найдете на компакт-диске в папке `\Code\Chapter15`.

```
/*
 * MainGameCanvas.java
 * Листинг 15.2
 *
 */

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import java.lang.*;
import java.util.*;
import java.io.*;

/**
 *
 * @author Stanislav Gornakov
 * @version 1.0
 */

public class MainGameCanvas extends GameCanvas implements
Runnable {
private GameMidlet midlet = null;
private Graphics graphics = null;
private volatile Thread animationThread = null;
private LayerManager gameManager = null;
private Background background = null;
private Ship ship = null;
private int fps = 50;
int screenWidth = 0;
int screenHeight = 0;
int gameState = 0 ;
private Image imageGameContinue = null;
private Image imageGameEnd = null;
private Image imageComandContinue = null;
private Image imageComandMenu = null;
private Meteorite [] meteor = new Meteorite [4];
private Shot runShotShip = null;
private Shot shootingShip = null;
private Vector shotShip = null;
int updateShot = 0;
private Random random = null;
private Explosion explosion = null;
private Image lifeShip = null;
int score = 0 ;
```

```
public MainGameCanvas(GameMidlet midlet) throws Exception
{
    super(true);
    this.midlet = midlet;
    setFullScreenMode(true);
    graphics = getGraphics();
    screenWidth = this.getWidth();
    screenHeight = this.getHeight();
    gameManager = new LayerManager();
    random = new Random();
    createGame();
}

public void start() {
    animationThread = new Thread(this);
    animationThread.start();
}

public void run() {...}

public void stop() {
    animationThread = null;
}

public void keyPressed(int keyCode) {

    switch(gameState) {

    case 0:

        if(keyCode == -6 || keyCode == -7) {
            this.stop();
            midlet.gameMenu();
        }
        break;

    case 1:

        if(keyCode == -6) {
            this.stop();
            midlet.gameMenu();
        }
        if(keyCode == -7) {
            this.stop ();
        }
    }
}
```

```
        midlet.loadingGame();
    }
    break;

    case 2:

        if(keyCode == -6){
            this.stop();
            midlet.gameMenu();
        }
        if(keyCode == -7){
            this.stop();
            midlet.loadingGame();
        }
        break;
    }
}

private void draw() {
    graphics.setColor(250, 250, 250);
    graphics.fillRect(0, 0, screenWidth, screenHeight);
    gameManager.paint(graphics, 0, 0);
    ship.drawLifeShip(graphics, ship.lifeShip);
    graphics.drawImage(lifeShip, 1, 2, 0);

    switch(gameState){

        case 1:

            graphics.drawImage(imageGameContinue,
screenWidth/2, screenHeight/2,
Graphics.VCENTER | Graphics.HCENTER);
            graphics.drawImage(imageComandMenu, 0,
screenHeight-
            imageComandMenu.getHeight(), 0);
            graphics.drawImage(imageComandContinue,
screenWidth-
            imageComandContinue.getWidth(), screenHeight-
            imageComandContinue.getHeight(), 0);
            graphics.setFont(Font.getFont(Font.FACE_SYSTEM,
Font.STYLE_BOLD,
Font.SIZE_LARGE));
            graphics.setColor(0, 0, 0);
            graphics.drawString("" + score, 130 , 150, 0);
        }
    }
}
```



```
break;

case 2:

    graphics.drawImage(imageGameEnd,  screenWidth/2,
        screenHeight/2,
        Graphics.VCENTER | Graphics.HCENTER);
    graphics.drawImage(imageComandMenu,  0,
screenHeight-
    imageComandMenu.getHeight(), 0);
    graphics.drawImage(imageComandContinue,
screenWidth-
    imageComandContinue.getWidth(),  screenHeight-
    imageComandContinue.getHeight(), 0);
    graphics.setFont(Font.getFont(Font.FACE_SYSTEM,
Font.STYLE_BOLD,
    Font.SIZE_LARGE));
    graphics.setColor(0, 0, 0);
    graphics.drawString("" + score, 130 , 150, 0);

break;
}

graphics.setClip(0, 0, screenWidth, screenHeight);
}

public void createGame() throws Exception {
    //*****
    // карта
    //*****
    try{
        Image imageBackground = Image.createImage("/
Background.png");
        background = new
Background(Background.COLUMNS_WIDTH,
        Background.ROWS_HEIGHT,  imageBackground,
Background.WIDTH,
        Background.HEIGHT);
        background.setVisible(false);
    }catch (Exception ex) {
        System.err.println("Background it is not loaded");
    }

    //*****
```

```

// изображения
//*****

try{
    imageGameContinue = Image.createImage("/
Continue.png");
    imageGameEnd = Image.createImage("/GameEnd.png");
    imageComandMenu = Image.createImage("/
comandMenu.png");
    imageComandContinue = Image.createImage("/
comandContinue.png");
    Image imageExplosion = Image.createImage("/
Explosion.png");
    explosion = new Explosion(imageExplosion, 44, 44);
    explosion.setVisible(false);
    imageExplosion = null;
    lifeShip = Image.createImage("/Life.png");
}catch (Exception ex) {
    System.err.println("Image it is not loaded");
}

//*****
// корабль
//*****

try{
    Image imageShip = Image.createImage("/Ship.png");
    ship = new Ship(this, imageShip, Ship.WIDTH,
Ship.HEIGHT);
    ship.setVisible(false);
    imageShip = null;
}catch (Exception ex) {
    System.err.println("Ship.png it is not loaded");
}
// ===== shot для ship =====
shotShip = new Vector();

//*****
// метеорит
//*****

try{
    Image imageMeteorite1 = Image.createImage("/
Meteorite1.png"); ,
    Image imageMeteorite2 = Image.createImage("/
Meteorite2.png");
    meteor[0] = new Meteorite(imageMeteorite1, 40, 35);

```

```
        meteor[1] = new Meteorite(imageMeteorite1, 40, 35)
        meteor[2] = new Meteorite(imageMeteorite2, 40, 35)
        meteor[3] = new Meteorite(imageMeteorite2, 40, 35)
        imageMeteorite1 = null;
        imageMeteorite2 = null;
    }catch (Exception ex) {
        System.err.println("meteor images are not
loaded");
    }
}

public void setGame(){

    gameState = 0;
    score = 0;
    ship.lifeShip = 100;
    //*****
    // background
    //*****
    background.createBackground();
    background.setPosition(0, -background.getHeight() +
screenHeight);
    background.setVisible(true);

    //*****
    // ship
    //*****
    ship.setPosition(screenWidth/2-ship.WIDTH/
2,screenHeight-ship.HEIGHT -10);
    ship.setVisible(true);
    updateShot = 900;

    //*****
    // устанавливаем метеориты на позиции
    //*****
    meteor[0].setPosition(Math.abs(random.nextInt() %
(screenWidth -
meteor[0].WIDTH)), -100);
    meteor[1].setPosition(Math.abs(random.nextInt() %
(screenWidth -
meteor[1].WIDTH)), -400);
    meteor[2].setPosition(Math.abs(random.nextInt() %
(screenWidth -
meteor[2].WIDTH)), -700);
```

```

    meteor[3].setPosition(Math.abs(random.nextInt() %
(screenWidth -
meteor[3].WIDTH)), -1000);

//*****
// добавляем игровые компоненты
//*****

gameManager.append(explosion);
for(int i = 4; -i >= 0;){
    gameManager.append(meteor[i]);
}
gameManager.append(ship);

gameManager.append(background);

}

public void updateGame() {

//*****
// движение карты
//*****

if(background.getY() > - 1){
    background.move(0, 0);
    gameState = 1;
} else{
    background.move(0, 1);
}

if(ship.lifeShip <= 0) gameState = 2;

switch(gameState) {

    case 0:
//*****
// клавиши
//*****
        int keyStates = getKeyStates();
        // move down
        if ((keyStates & DOWN_PRESSED) != 0){
            ship.moveShip(1);
            // move up
        } else if ((keyStates & UP_PRESSED) != 0){
            ship.moveShip (2);
            // move left

```



```
}else if ((keyStates & LEFT_PRESSED) != 0) {
    ship.moveShip(3);
// move right
} else if ((keyStates & RIGHT_PRESSED) != 0) {
    ship.moveShip(4);
// stop
} else {
    ship.moveShip(0);
}

//*****
// движение метеорита
//*****
for (int i = 4; -i >= 0;) {
    meteor[i].moveMeteorite(15);
    if(this.screenHeight < meteor[i].getY()){
        meteor[i].setPosition(Math.abs(random.nextInt())
%( screenWidth - meteor[i].WIDTH), -200);
    }
}

//*****
// стрельба
//*****
try {
    firingShip ();
} catch (Exception exc) {
    System.err.println("Not fire");
}
moveShotShip();

//*****
// СТОЛКНОВЕНИЯ
//*****
for(int i = 4; -i >= 0;){
    overlapsShipMeteor(meteor[i]);
    checkCollisions(meteor[i]);
}

//*****
// взрыв
//*****
if(explosion.isVisible()) {
    explosion.updateExplosion();
}
```

```

        }
        break;
    }
}

protected void firingShip() throws Exception {...}
protected void moveShotShip() {...}

//=====
//  СТОЛКНОВЕНИЯ
//=====
// ship и meteorite
public void checkCollisions(Sprite sprite){
    if(ship.collidesWith(sprite, true)){
        ship.lifeShip -= 15;
        if(!explosion.isVisible()){
            explosion.explosion(sprite);
        }
        sprite.setPosition(Math.abs(random.nextInt() %
(screenWidth - 50)), -200);
    }
}

// shot ship и meteorite
boolean overlapsShipMeteor(Sprite sprite) {
return collidesShotMeteor(sprite, shotShip.size());
}
boolean collidesShotMeteor(Sprite sprite, int count){
    for (int i = 0; i < count; i++){
        Shot m = (Shot)(shotShip.elementAt(i));
        if (sprite.collidesWith( m , true)) {
            score +=10;
            m.setVisible(false);
            if(!explosion.isVisible()){
                explosion.explosion(sprite);
            }
        }

        sprite.setPosition(Math.abs(random.nextInt()%(screenWidth
- 50)), -200) ;
    }
}
return true;
}
}

```



<http://palata-x.narod.ru>

Глава 16. Звуковые эффекты

Прежде чем мы добавим в игру пару звуковых эффектов, давайте сначала рассмотрим два демонстрационных примера, иллюстрирующих воспроизведение тональных звуков и файлов в форматах WAV, MIDI, AU и MP3.

В профиле MIDP 1.0 возможность работы со звуком отсутствует, и все строится на использовании классов, предоставляемых производителями мобильных телефонов. В профиле MIDP 2.0 такая возможность имеется, поскольку появилась мобильная мультимедиа-библиотека (MMAPI), разработанная экспертной группой, в состав которой входят известные компании:

- Nokia (Specification Lead);
- Aplix Corporation;
- Beatnik, Inc.;
- France Telecom;
- Insignia Solutions;
- Mitsubishi Electric Corp.;
- Motorola;
- Netdecisions Holdings United;
- NTT DoCoMo, Inc.;
- Openwave Systems Inc.;
- Packet Video Corporation;
- Philips;
- Siemens AG ICM MP TI;
- Smart Fusion;
- Sun Microsystems, Inc.;
- Symbian Ltd;
- Texas Instruments Inc.;
- Vodafone;
- Yamaha Corporation;
- Zucotto Wireless.

На данный момент существуют две мобильные *мультимедиа-библиотеки*, различающиеся по своему назначению и спецификации, это:

- Mobile Media API - предназначена для работы с устройствами, имеющими более мощные системные ресурсы. Это, как правило, карманные портативные устройства;
- MIDP 2.0 Media API - эта библиотека направлена на поддержку мобильных устройств с ограниченными ресурсами.

В этой главе будет представлена мобильная мультимедиа-библиотека MIDP 2.0 Media API, которая используется при программировании звука в приложениях,

написанных под профиль MIDP 2.0. Работа со звуком строится по принципу блочной конструкции, состоящей из двух ключевых блоков:

- *Менеджер* - это основной диспетчер, при помощи которого создаются все проигрыватели. Дополнительно менеджер имеет возможность воспроизводить простые тональные звуки на телефоне. Менеджер в профиле MIDP 2.0 представлен классом `Manager`;
- *Проигрыватель* - осуществляет непосредственное воспроизведение звуков и представлен интерфейсом `Player`.

Работа со звуком построена на использовании нескольких интерфейсов и класса `Manager`, библиотеки MIDP 2.0 Media API. Данная библиотека состоит из двух следующих пакетов:

- `javax.microedition.media`;
- `javax.microedition.media.control`.

Эти пакеты содержат ряд интерфейсов и всего один класс `Manager`. Рассмотрим подробно оба пакета библиотеки MIDP 2.0 Media API, давая попутно краткую характеристику каждому компоненту. На основе полученного материала далее в главе создадим несколько примеров исходного кода, иллюстрирующих модель работы со звуком, а в конце главы добавим в игру «Метеоритный дождь» звуковые эффекты.

16.1. Пакет `javax.microedition.media`

Пакет `javax.microedition.media` необходим для работы со звуком и содержит четыре основных интерфейса и один класс, на базе которых и происходит воспроизведение звуков в телефоне.

16.1.1. Интерфейс `Control`

Интерфейс `Control` - это самый главный интерфейс, с его помощью осуществляется контроль над всеми имеющимися ресурсами, также от этого интерфейса наследуются еще два интерфейса `ToneControl` и `VolumeControl`.

16.1.2. Интерфейс `Controllable`

С помощью *интерфейса `Controllable`* можно получить управление над воспроизведением посредством использования двух методов:

- `Control getControl(String controlType)` - определяет тип управления;
- `Control[] getControls()` - получает управление.

16.1.3 Интерфейс `Player`

Интерфейс `Player` наследуется от интерфейса `Controllable` и необходим для реализации процесса воспроизведения звуковых данных на основе формирования

проигрывателей. Проигрыватели создаются методом `createPlayer()` класса `Manager`, например:

```
Player player1 = Manager.createPlayer();
```

После создания проигрывателя можно производить воспроизведение звука, для этого необходимо воспользоваться методами интерфейса `Player`.

Методы интерфейса *Player*

- `void addPlayerListener(PlayerListener PlayerListener)` - осуществляет обработку событий от определенного проигрывателя;
- `void close()` - закрывает проигрыватель;
- `void deallocate()` - освобождает ресурс, занятый проигрывателем;
- `String getContentType()` - получает тип звуковых данных, воспроизводимых проигрывателем;
- `long getDuration()` - получает размер звукового файла;
- `long getMediaTime()` - получает время воспроизведения звуковых данных;
- `int getState()` - определяет состояние проигрывателя;
- `void removePlayerListener(PlayerListener PlayerListener)` - удаляет установленный обработчик событий;
- `void setLoopCount(int count)` - устанавливает цикличное воспроизведение звуковых данных;
- `long setMediaTime(long now)` - устанавливает время воспроизведения;
- `void start()` - дает команду на воспроизведение;
- `void stop()` - останавливает воспроизведение.

Большинство методов направлены на работу со звуковыми данными, позже в разделе 16.3 мы разберем подробнее работу с методами интерфейса `Player`.

16.1.4. Интерфейс *PlayerListener*

Интерфейс `PlayerListener` позволяет осуществлять обработку событий, полученных от проигрывателя. Помните, в главе 5 мы разбирали работу интерфейса `CommandListener`? Интерфейс `PlayerListener` функционирует почти по такой же точно схеме, но ориентирован на работу с проигрывателем. В составе интерфейса `PlayerListener` имеется всего один метод:

- `void playerUpdate(Player player, String event, Object eventData)` - обновляет состояние проигрывателя.

С помощью констант интерфейса `Player` в методе `playerUpdate()` нужно задавать тип необходимых событий в параметрах `eventData` и `event`:

- `static String CLOSED` - уведомляет о закрытии проигрывателя;
- `static String DEVICE_AVAILABLE` - уведомляет о доступности проигрывателя;
- `static String DEVICE_UNAVAILABLE` - уведомляет о недоступности проигрывателя;

- `static String DURATION_UPDATED` - обновляет состояние;
- `static String END_OF_MEDIA` - уведомляет о конце воспроизведения данных проигрывателем;
- `static String ERROR` - уведомляет об ошибке;
- `static String STARTED` - уведомляет о начале работы проигрывателя;
- `static String STOPPED` - уведомляет о конце работы проигрывателя;
- `static String VOLUME_CHANGED` - уведомляет о выборе громкости для воспроизведения.

16.1.5. Класс *Manager*

Класс *Manager* создает проигрыватель для воспроизведения звуков, а также отслеживает доступные протоколы звуковых данных с помощью нескольких методов:

- `static Player createPlayer (InputStream stream, String type)` - создает проигрыватель для воспроизведения звуковых данных из потока;
- `static Player createPlayer (String locator)` - создает проигрыватель для воспроизведения звуковых данных из определенного файла;
- `static String[] getSupportedProtocols (String content_type)` - возвращает список доступных протоколов для мобильного устройства;
- `static void playTone (int note, int duration, int volume)` - воспроизводит различные тональные звуки.

16.2. Пакет `javax.microedition.media.control`

Пакет `javax.microedition.media.control` небольшой по своему составу и производит контроль над процессами, связанными с воспроизведением и регулировкой звука. В разделе 16.4 этой главы очень подробно рассматривается схема контроля.

16.2.1 Интерфейс *ToneControl*

С помощью интерфейса *ToneControl* происходят настройка и построение блока тональных звуков для воспроизведения. Это достигается путем использования метода `void setSequence (byte[] sequence)`, который устанавливает тональные звуки для воспроизведения и набора следующих констант:

- `static byte BLOCK_END` - конец блока воспроизведения;
- `static byte BLOCK_START` - стартовая позиция в блоке;
- `static byte C4` - нота До;
- `static byte PLAY_BLOCK` - воспроизвести блок;
- `static byte REPEAT` - повторить воспроизведение блока;
- `static byte SET_VOLUME` - установить громкость;
- `static byte SILENCE` - без звука;
- `static byte TEMPO` - темп или скорость воспроизведения;
- `static byte VERSION` - версия атрибута воспроизведения.

С помощью перечисленных констант производится настройка блока тональных звуков для воспроизведения, о которых мы поговорим подробно в *разделе 16.4*.

16.2.2. Интерфейс *VolumeControl*

Интерфейс VolumeControl имеет методы, на основе которых можно реализовать управление громкостью воспроизведения:

- `int getLevel()` - возвращает текущий уровень громкости;
- `boolean isMuted()` - определяет состояние сигнала;
- `int setLevel(int level)` - устанавливает уровень громкости. Значение может находиться в пределах от 0 до 100;
- `void setMute(boolean mute)` - устанавливает состояние сигнала.

Сейчас мы вкратце рассмотрели имеющиеся интерфейсы, классы, методы и константы двух пакетов `javax.microedition.media` и `javax.microedition.media.control`. Теперь давайте подытожим все полученную информацию и рассмотрим примеры, иллюстрирующие работу со звуком в мобильных телефонах.

16.3. Воспроизведение звуковых файлов

Воспроизведение звуковых файлов в телефоне - задача несложная. Звуковой файл должен быть размещен в каталоге создаваемого приложения. Если вы используете J2ME Wireless Toolkit, то расположите ваш файл в папке `\res`. Впоследствии, после компиляции и установки программы, звуковой файл будет находиться в JAR-архиве и доступен для воспроизведения.

Для того чтобы воспроизвести необходимый wav-файл, создается объект класса *InputStream* для связывания потока данных с ресурсом, допустим с wav-файлом, например:

```
InputStream input =
getClass().getResourceAsStream("файл.wav");
```

Затем создается проигрыватель:

```
Player player = Manager.createPlayer(input, "audio/X-wav");
```

Проигрыватель формируется с помощью метода *createPlayer()* класса *Manager*. Количество создаваемых проигрывателей регламентируется только системными ресурсами телефона. После чего используется метод *start()* для воспроизведения wav-файла.

В листинге 16.1 вы найдете пример исходного кода, в котором происходят загрузка и воспроизведение wav-файла из JAR-архива. В этом примере используется класс *Form*, с помощью которого создается пустой экран, и добавляются две команды: выход из приложения и воспроизведение wav-файла. Основные действия разворачиваются в методе *WavPlay()*, где создается проигрыватель и воспроизводится wav-файл. Обратите также внимание на подключаемые пакеты.

```
/**
Листинг 16.1
класс WavMIDlet
воспроизводит wav-файл
*/
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;
import java.io.*;

public class WavMIDlet extends MIDlet implements
CommandListener
{
    // команда выхода
    private Command exitMidlet = new Command("Выход",
    Command.EXIT, 0) ;
    // команда воспроизведения
    private Command pl = new Command("Играть", Command.OK,
    // объект mydisplay представляет экран телефона
    private Display mydisplay;

    public WavMIDlet()
    {
        mydisplay = Display.getDisplay(this);
    }

    public void startApp()
    {
        Form ls = new Form("Воспроизведение wav");
        // добавляем команду выхода
        ls.addCommand(exitMidlet);
        // добавляем команду воспроизведения
        ls.addCommand(pl);
        ls.setCommandListener(this);
        // отражаем текущий дисплей
        mydisplay.setCurrent(ls);
    }

    private void WawPlay()
    {
        try {
            // ищем ресурс с именем melod.wav
            InputStream input =
```



```

getClass().getResourceAsStream("melod.wav");
    // создаем проигрыватель
    Player player = Manager.createPlayer(input, "audio/X-
wav");
    // воспроизводим
    player.start();
    } catch (IOException zxz) {}
    catch (MediaException zmz) {}
}

public void pauseApp() {}

public void destroyApp(boolean unconditional){}

public void commandAction(Command c, Displayable d)
{
    if (c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
    if (c == pl)
    {
        WawPlay();
    }
}
}

```

Воспроизведение файлов в формате **MIDI** и **MP3** происходит точно так же, но при создании проигрывателя и загрузке звукового эффекта в игру применяются следующие строки кода:

```

// для MIDI
Player player = Manager.createPlayer(input, "audio/midi");
// для MP3
Player player = Manager.createPlayer(input, "audio/mpeg");
// для AU
Player player = Manager.createPlayer(input, "audio/basic");

```

16.4. Воспроизведение тональных звуков

При создании звукового сопровождения к играм и приложениям часто используются так называемые тональные звуки, генерируемые телефоном. *Воспроизведение тональных звуков* происходит примерно тем же способом, что и воспроизведение звуковых файлов, рассмотренных в предыдущем разделе. Создание

тональных звуков строится на основе секвенсора, используемого музыкантами. То есть в вашем распоряжении имеются семь нот, которые могут быть сыграны в любой тональности. Указав определенную последовательность нот в заданном массиве данных, вы сможете их потом последовательно воспроизвести. В принципе, аналогичные действия можно произвести в любом телефоне, где в музыкальном редакторе вы выстраиваете некую последовательность определенных символов, обозначающих ноты. Указав нужную последовательность нот, вы получаете готовую мелодию, созданную при помощи тональных звуков.

Перейдем к практике. Первым делом необходимо создать те самые семь нот. В классе *ToneControl* пакета `javax.microedition.media.control.*` доступна константа `C4`, которая по своему звучанию соответствует ноте «До». Для того чтобы создать, например, ноту `Re`, можно воспользоваться следующей конструкцией кода:

```
byte Re = (byte) (ToneControl.C4+1) ;
```

Для создания последующей ноты `Mi` нужно прибавить к ноте `Do` (то есть `C4`) число два и т. д. Когда закончатся все семь нот, то вы переходите к следующей октаве, что и предопределяет разные тональности звукового сопровождения. Всего можно использовать значение от 0 до 127.

Затем создается массив данных, можно назвать его *секвенсором*, в котором указывается последовательность нот. Синтаксис, используемый в секвенсоре, строго определен, и его необходимо правильно использовать. Например, имеется следующий массив данных, характеризующийся как секвенсор:

```
byte[] Nota = {...};
```

В этом массиве данных первой строкой кода должно идти указание версии используемого атрибута.

```
ToneControl.VERSION, 1,
```

Затем задается *скорость воспроизведения* с помощью целочисленного значения, которое может варьироваться от 5 до 127. Например:

```
ToneControl.TEMPO, 30,
```

Далее необходимо дать команду, указывающую начало блока последовательности нот для воспроизведения, например:

```
ToneControl.BLOCK_START, 0,
```

И только после этого идет последовательность нот. Между нотами обязательно ставится длина ноты, обычно заданная переменной, и ее диапазон может быть от 2 до 16. Например:

```
byte d = 4;
Re,d,Mi,d,Re,d,
```

Между воспроизведением нот можно использовать *паузы* для создания выразительной мелодии. Пауза задается с помощью константы `SELENCE`, например:

```
byte stop = ToneControl.SILENCE;
byte d = 4;
```

Тогда последовательность нот может быть следующей:

```
Re, d, stop, d, Mi, d, stop, d, stop, d, Re, d,
```

После того как вы задали всю последовательность нот, необходимо четко указать конец блока с помощью *константы* `BLOCK_END` следующим образом:

```
ToneControl.BLOCK_END, 0,
```

На каждую *константу* `BLOCK_START` должна присутствовать константа `BLOCK_END`. Иначе возникнет ошибка при компиляции.

В конце нужно воспользоваться *константой* `PLAY_BLOCK` для воспроизведения блока последовательности нот. После этого созданный секвенсор можно использовать для воспроизведения проигрывателем тональных звуков. Посмотрите на листинг 16.2, где показана демонстрационная программа, воспроизводящая все семь нот одной октавы.

```
/**
Листинг 16.2
класс TonMIDlet
воспроизводит тональные звуки
*/
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;
import java.io.*;

public class TonMIDlet extends MIDlet implements
CommandListener
{
// команда выхода
private Command exitMidlet = new Command("Выход",
Command.EXIT, 0);
// команда воспроизведения
private Command pl = new Command("Играть", Command.OK, 1);
// объект mydisplay представляет экран телефона
private Display mydisplay;

public TonMIDlet()
{
mydisplay = Display.getDisplay(this);
}
```

```
public void startApp()
{
    Form ls = new Form("Тональные звуки");
    // добавляем команду выхода
    ls.addCommand(exitMidlet);
    // добавляем команду воспроизведения
    ls.addCommand(pl);
    ls.setCommandListener(this);
    // отражаем текущий дисплей
    mydisplay.setCurrent(ls);
}

private void TonPlay()
{
    // нота До
    byte Do = ToneControl.C4;
    // нота Ре
    byte Re = (byte) (ToneControl.C4 + 1);
    // нота Ми
    byte Mi = (byte) (ToneControl.C4 + 2);
    // нота Фа
    byte Fa = (byte) (ToneControl.C4 + 3);
    // нота Соль
    byte So = (byte) (ToneControl.C4 + 4);
    // нота Ля
    byte Lj = (byte) (ToneControl.C4 + 5);
    // нота Си
    byte Si = (byte) (ToneControl.C4 + 6);
    // пауза
    byte stop = ToneControl.SILENCE;
    // скорость воспроизведения тональных звуков
    byte speed = 30;
    // продолжительность воспроизведения ноты
    byte pr = 4;
    // секвенсор
    byte[] Nota = {
        // атрибут, задающий номер версии
        ToneControl.VERSION, 1,
        // скорость воспроизведения
        ToneControl.TEMPO, speed,
        // начало блока
        ToneControl.BLOCK_START, 0,
        // последовательность нот для воспроизведения
```



```
Do,pr,stop,pr,Re,pr,stop,pr,Mi,pr,stop,pr,
Fa,pr, stop,pr,So,pr,stop,pr,Lj,pr,stop,pr,Si,pr,
// конец блока
ToneControl.BLOCK_END, 0,
// воспроизведение блока
ToneControl.PLAY_BLOCK, 0,
};
// воспроизводим тональные звуки из секвенсора
try{
Player player =
Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
player.realize ();
ToneControl toncontrl =
(ToneControl)player.getControl("ToneControl");
toncontrl.setSequence(Nota);
player.start ();
} catch (IOException zxz){}
catch (MediaException zmz){}
}

public void pauseApp() {}

public void destroyApp(boolean unconditional){}

public void commandAction(Command c, Displayable d)
{
    if (c == exitMidlet)
    {
        destroyApp(false);
        notifyDestroyed();
    }
    if (c == pl)
    {
        TonPlay ();
    }
}
}
```

В основе программы из листинга 16.2 лежит модель, используемая в предыдущем разделе при воспроизведении wav-файла. В классе `TonMIDlet` создается пустая форма классом `Form` и назначаются две команды: выход из программы и воспроизведение тональных звуков. При нажатии на кнопку **Играть** задействуется метод `TonPlay()`, где создаются ноты, секвенсор, после чего происходит воспроизведение последовательности нот.

16.5. Добавляем звук в игру

В игру «Метеоритный дождь» мы добавим два звуковых файла. Один звуковой файл будет воспроизводиться в меню игры, а другой звуковой файл будет проигрываться в игровом процессе в момент взрыва метеоритов. Естественно, вы в своей игре можете создать любое количество звуковых файлов и воспроизводить их в различные игровые моменты.

Все необходимые знания для воспроизведения звука в играх вы уже имеете, но я вам покажу два универсальных метода, которые написаны достаточно хорошо и которые вы можете использовать в любых играх. Первый метод создает проигрыватель и загружает в приложение необходимый звуковой файл.

```
// загрузка звукового файла
private Player createSound(String file, String format){
    Player player = null;
    try {
        InputStream input =
getClass().getResourceAsStream(file);
        player = Manager.createPlayer(input, format);
        player.prefetch();
    } catch (IOException ex) {
    } catch (MediaException ex) {
    }
    return player;
}
```

Здесь все очень просто и основано на информации, полученной в этой главе. Необходимо просто вызвать в момент создания объектов игры, например, следующим образом:

```
private Player sound = null;
...
sound = createSound("/Scroll.wav", "audio/X-wav");
```

После чего в исходном коде необходимо использовать другой метод, который занимается уже непосредственно воспроизведением звукового файла.

```
// воспроизведение
private void startSound() {
    if (scroll != null) {
        try {
            sound.stop();
            sound.setMediaTime(0L);
            sound.start();
        } catch (MediaException ex) {
        }
    }
}
```

```
}  
}
```

Вызов в игре метода `startSound()` должен производиться в том месте, где вы желаете услышать воспроизведение звука. Сам метод интересен тем, что организует в своем теле автоматический запуск и остановку воспроизведения файла, создавая тем самым безошибочный механизм проигрывания звуковых данных.

Что касается игры «Метеоритный дождь», то здесь также используются два этих метода. Давайте рассмотрим воспроизведение звука в меню игры. Идея здесь следующая. В момент создания класса `Menu` в игру загружается звуковой эффект, воспроизведение которого назначается на нажатие пользователем джойстика и клавиш выбора. Соответственно, когда пользователь начинает перемещаться по меню джойстиком, он слышит характерные щелчки. В листинге 16.3 приведен исходный код класса `Menu`, с которым, думаю, вы разберетесь самостоятельно.

Что касается использования звука в самой игре, то там также применяется аналогичная схема, но воспроизведение взрыва происходит в методе, обрабатывающем столкновения корабля и пуль с метеоритом. Полный исходный код всего проекта и окончательной версии игры вы найдете на компакт-диске в папке **\Code\Chapter16\Meteoric_rain16**.

```
/*  
 * Menu.java  
 * ЛИСТИНГ 16.3  
 *  
 */  
  
import javax.microedition.lcdui.*;  
import javax.microedition.lcdui.game.*;  
import javax.microedition.media.*;  
import java.io.*;  
  
/**  
 *  
 * @author Stanislav Gornakov  
 * Aversion 1.0  
 */  
  
public class Menu extends Canvas implements Runnable{  
    private GameMidlet midlet = null;  
    private volatile Thread thread = null;  
    private Image imageMenu = null;  
    private Image imageAbout = null;  
    private Image imageCursorGame = null;  
    private Image imageCreditAbout = null;  
    private Image imageCursorExit = null;
```

```
private boolean loop = false;
private boolean about= false;
int x45 = 0;
int x70 = 0;
int yCursorGame = 0 ;
int yCursorAbout = 0;
int yCursorExit = 0;
int state = 1;
private Player scroll = null;

public Menu(GameMidlet midlet) {
    this.midlet = midlet;
    setFullScreenMode(true);
    try {
        imageMenu = Image.createImage("/Menu.png");
        imageAbout = Image.createImage("/Splash.png");
        imageCursorGame = Image.createImage("/Game.png");
        imageCreditAbout = Image.createImage("/About.png");
        imageCursorExit = Image.createImage("/Exit.png");
    }catch (Exception ex) {
        System.err.println("In the block =menu= images
are not loaded");
    }
    // Курсоры
    x45 = 45;
    x70 = 70;
    yCursorGame = 180;
    yCursorAbout = 220;
    yCursorExit = 260;
    state = 1;
    scroll = createSound("/Scroll.wav", "audio/X-wav");
}

public void start () {
    loop = true;
    thread = new Thread(this);
    thread.start();
}

public void run() {
```



```
while (loop) {
    repaint();
    try {
        Thread.sleep(20);
    } catch (Exception ex) {
        System.err.println("Class Menu metod run()");
    }
}

}

public void stop() {
    loop = false;
    thread = null;
    System.gc();
}

public void paint (Graphics graphics) {...}

protected void keyPressed(int keyCode) {
    int action = getGameAction(keyCode);

    if (about) {
        if (keyCode == -6 || keyCode == -7)
        {
            about = false;
            startScrollSound();
        }
    }
}

switch (action) {

    // нажать Down
    case Canvas.DOWN:
        state += 1;
        if (state > 3) state = 1;
        startScrollSound();
        break;

    // нажать Up
    case Canvas.UP:
```

```
        state -= 1;
        if (state < 1) state = 3;
        startScrollSound();
break;

// нажать Fire
case Canvas.FIRE:
    startScrollSound();
    // игра
    if (state == 1) {
        this.stop();
        midlet.loadingGame();
    }
    // об игре
    if (state == 2) {
        about = true;
    }
    // ВЫХОД
    if (state == 3) {
        this.stop();
        midlet.exitGame();
    }
break;
    }
}

// загрузка звукового файла
private Player createSound(String file, String format) {
    Player player = null;
    try {
        InputStream input =
getClass().getResourceAsStream(file);
        player = Manager.createPlayer(input, format);
        player.prefetch();
    } catch (IOException ex) {
    } catch (MediaException ex) {
    }
    return player;
}

// воспроизведение
private void startScrollSound() {
    if (scroll != null) {
```

```
        try {  
            scroll.stop ();  
            scroll.setMediaTime(0L);  
            scroll.start ();  
        } catch (MediaException ex) {  
        }  
    }  
}  
}
```

<http://palata-x.narod.ru>

<http://palata-x.narod.ru>

Приложение 1

Обзор компакт-диска

- **Code** - эта папка содержит программы и исходные коды, рассмотренные в книге.
- **IDE** - папка с инструментальными средствами для программирования мобильных приложений. В папке располагаются следующие файлы:
 - **j2sdk-1_4_2_05-windows-i586-p** - Java 2 SE SDK;
 - **sun_java_wireless_toolkit-2_3** - инструментарий Wireless Toolkit;
 - **netbeans-5_0-windows** - инструментарий NetBeans **IDE**;
 - **netbeans_mobility-5_0-win** - дополнения к инструментарию NetBeans **IDE**, позволяющие работать с мобильными программами.
- **SDK** - эта папка имеет пять вложенных папок.
- **BenQ-Siemens** - Java SDK компании BenQ-Siemens:
 - **smtk** - программное средство от компании BenQ-Siemens;
 - набор эмуляторов с номерами, идентичными названиям телефонов этой компании.
- **J Motorola** - Java SDK компании Motorola:
 - **sdk_motorola_6.1.1** - комплект программных средств.
- **Nokia** - Java SDK компании Nokia;
 - **carbide_j_vl_0_1** - инструменты от Nokia.
- **Samsung** - Java SDK компании Samsung:
 - **SJSDKv3.0** - комплект Java SDK от Samsung.
- **Sony Ericsson** - Java SDK компании Sony Ericsson:
 - **semc_java_me_sdk.2-2-3** - инструментарий Wireless Toolkit от Sony Ericsson;
 - **semc_java_me_sdk_2_2_3_addon3** - дополнительные эмуляторы.
 - **semc_java_me_sdk_2_2_3_addon5** - дополнительные эмуляторы.



<http://palata-x.narod.ru>

Приложение 2

Справочник по Java 2 Micro Edition

В этом приложении содержится исчерпывающий справочный материал по всем пакетам, интерфейсам, классам, конструкторам классов и константам платформы Java 2 Micro Edition. Справочник выполнен на основе имеющейся документации к платформе Java 2 ME, находящейся на сайте компании Sun Microsystems по адресу в Интернете: <http://java.sun.com>. Аналогичная документация также поставляется с платформой Java 2 ME, которую вы найдете на компакт-диске, приложенном к книге, в составе среды программирования SUN ONE Studio 4 Mobile Edition и J2ME Wireless Toolkit 2.1. Справочник рассматривает все пакеты, имеющиеся в составе платформы Java 2 ME. Каждый пакет содержит множество интерфейсов, классов, конструкторов и констант. Для всех имеющихся компонентов каждого пакета дается краткая характеристика, на основании которой вам будет очень легко сориентироваться в создании приложений для платформы Java 2 Micro Edition.

2.1. Пакет `java.lang`

Содержит системные классы, или основы языка Java.

2.1.1. Интерфейс *Runnable*

- Использование интерфейса `Runnable` обеспечивает создание потока в программах.

Метод

- `void run()` - запускает поток в приложении.

2.1.2. Класс *Boolean*

Объектно-ориентированный класс-оболочка, или, как еще говорят, «обертка» для типа `Boolean`.

Конструктор

- `Boolean(boolean value)` - создает объект класса `Boolean`.

Методы

- `boolean booleanValue()` - возвращает значение объекта класса `Boolean`;
- `boolean equals(Object obj)` - возвращает значение `true`, если это объект класса `Boolean`;

- `int hashCode()` - возвращает специальный код (хэш-код) объекта класса `Boolean`;
- `String toString()` - возвращает объект класса `String` для булевой переменной.

2.1.3. Класс *Byte*

Объектно-ориентированный класс-оболочка для простого типа `Byte`.

Конструктор

- `Byte(byte value)` - создает объект класса `Byte`.

Методы

- `byte byteValue()` - возвращает значение объекта класса `Byte`;
- `boolean equals(Object obj)` - возвращает значение в байтах для объекта класса `Byte`;
- `int hashCode()` - возвращает специальный код (хэш-код) объекта класса `Byte`;
- `static byte parseByte(String s)` - возвращает значение в байтах для указанного объекта `String`;
- `static byte parseByte(String s, int radix)` - возвращает значение в байтах для указанного объекта `String` на основе системы исчисления;
- `String toString()` - возвращает объект класса `String`, представленный значением `Byte`.

Константы

- `static byte MAX_VALUE` - максимальное значение в байтах;
- `static byte MIN_VALUE` - минимальное значение в байтах.

2.1.4. Класс *Character*

Объектно-ориентированный класс-сболочка для простого типа `Char`.

Конструктор

- `Character(char value)` - создает объект класса `Character`.

Методы

- `char charValue()` - возвращает значение объекта класса `Character`;
- `static int digit(char ch, int radix)` - возвращает числовое значение на основе системы исчисления;
- `boolean equals(Object obj)` - сравнивает объект;
- `int hashCode()` - возвращает специальный код (хэш-код) объекта класса `Character`;
- `static boolean isDigit(char ch)` - узнает, является ли значение `ch` числовым значением;
- `static boolean isLowerCase(char ch)` - определяет, является ли символьное значение строчным;
- `static boolean isUpperCase(char ch)` - определяет, является ли символьное значение заглавным;

- `static char toLowerCase(char ch)` - переводит символ в нижний регистр;
- `String toString()` - возвращает объект класса `String`, представленный значением `Character`;
- `static char toUpperCase(char ch)` - переводит символ в верхний регистр.

Константы

- `static int MAX_RADIX` - максимально доступное преобразование;
- `static char MAX_VALUE` - максимальное значение;
- `static int MIN_RADIX` - минимально доступное преобразование;
- `static char MIN_VALUE` - минимальное значение.

2.1.5. Класс Class

Виртуальная Java-машина создает объекты этого класса, которые представляют интерфейсы и классы языка Java.

Методы

- `static Class forName(String className)` - возвращает объект `Class` по названию класса;
- `String getName()` - возвращает имя интерфейса, класса, массива классов, простых типов, представляемых классом `Class`;
- `InputStream getResourceAsStream(String name)` - берет искомый ресурс с заданным именем;
- `boolean isArray()` - определяет, является ли объект массивом классов;
- `boolean isAssignableFrom(Class cls)` - определяет, является ли интерфейс или класс суперинтерфейсом или суперклассом;
- `boolean isInstance(Object obj)` - определяет совместимость указанных объектов;
- `boolean isInterface()` - определяет, каким типом интерфейса представлен данный класс;
- `Object newInstance()` - создает новый экземпляр класса;
- `String toString()` - конвертирует объект к виду `String`.

2.1.6. Класс Integer

Объектно-ориентированный класс для простого типа `int`.

Конструктор

- `Integer(int value)` - создает объект класса `Integer`.

Методы

- `byte byteValue()` - возвращает значение в байтах;
- `boolean equals(Object obj)` - сравнивает объекты;
- `int hashCode()` - возвращает специальный код (хэш-код) объекта класса `Integer`;
- `int intValue()` - возвращает значение числа как тип `int`;

- `long longValue()` - возвращает значение числа как тип `long`;
- `static int parseInt(String s)` - извлекает целое десятичное число из заданного объекта класса `String`;
- `static int parseInt(String s, int radix)` - извлекает целое десятичное число со знаком с использованием основания системы исчисления из заданного объекта класса `String`;
- `short shortValue()` - возвращает значение числа как тип `short`;
- `static String toBinaryString(int i)` - создает строковое представление целочисленного значения в виде целого числа без знака в двоичном представлении;
- `static String toHexString(int i)` - создает строковое представление целочисленного значения в виде целого числа без знака в шестнадцатеричном представлении;
- `static String toOctalString(int i)` - создает строковое представление целочисленного значения в виде целого числа без знака в восьмеричном представлении;
- `String toString()` - возвращает объект класса `String`, представленный значением целого числа;
- `static String toString(int i)` - возвращает заданный объект класса `String` как целое число;
- `static String toString(int i, int radix)` - создает строковое представление целого числа на основании системы исчисления;
- `static Integer valueOf(String s)` - возвращает новый объект класса `Integer`, инициализированный значением `s`;
- `static Integer valueOf(String s, int radix)` - возвращает новый объект класса `Integer`, инициализированный значением `s` на основе системы исчисления.

Константы

- `static int MAX_VALUE` - максимальное значение;
- `static int MIN_VALUE` - минимальное значение.

2.1.7. Класс Long

Объектно-ориентированный класс-оболочка для простого типа `long`.

Конструктор

- `Long(long value)` - создает объект класса `Long`.

Методы

- `boolean equals(Object obj)` - сравнивает объекты;
- `int hashCode()` - возвращает специальный код (хэш-код) объекта класса `Long`;
- `long longValue()` - возвращает значение числа как тип `long`;
- `static long parseLong(String s)` - извлекает большое целое десятичное число из заданного объекта класса `String`;

- `static long parseLong(String s, int radix)` - извлекает большое целое десятичное число со знаком с использованием основания системы исчисления из заданного объекта класса `String`;
- `String toString()` - возвращает объект класса `String`, представленный значением большого целого числа;
- `static String toString(long i)` - возвращает заданный объект класса `String` как целое большое число;
- `static String toString(long i, int radix)` - создает строковое представление большого целого числа на основании системы исчисления.

Константы

- `static long MAX_VALUE` - максимальное значение;
- `static long MIN_VALUE` - минимальное значение.

2.1.8. Класс Math

Математический класс, содержащий несколько методов для различных математических операций. Очень сильно урезан, в отличие от класса `Math` из Java 2 SE.

Методы

- `static int abs(int a)` - возвращает абсолютное значение из параметра `int a`, заданное целочисленным значением;
- `static long abs(long a)` - возвращает абсолютное значение из параметра `long a`, заданное большим целочисленным значением;
- `static int max(int a, int b)` - возвращает одно большее из двух значений типа `int`;
- `static long max(long a, long b)` - возвращает одно большее из двух значений типа `long`;
- `static int min(int a, int b)` - возвращает одно меньшее из двух значений типа `int`;
- `static long min(long a, long b)` - возвращает одно меньшее из двух значений типа `long`.

2.1.9. Класс Object

Суперкласс для всех классов Java. Все классы наследуются от класса `Object` и являются его подклассами.

Методы

- `boolean equals(Object obj)` - сравнивает объекты;
- `Class getClass()` - возвращает класс объекта;
- `int hashCode()` - возвращает специальный код (хэш-код) для объекта;
- `void notify()` - пробуждает отдельно взятый поток;
- `void notifyAll()` - пробуждает все имеющиеся потоки;
- `String toString()` - возвращает строковое представление данного объекта;

- `void wait()` - приостанавливает работу потока;
- `void wait(long timeout)` - приостанавливает работу потока на время, заданное в миллисекундах;
- `void wait(long timeout, int nanos)` - приостанавливает работу потока на время, заданное в миллисекундах, учитывая также дополнительное время, заданное в наносекундах.

2.1.10. Класс *Runtime*

Класс времени исполнения приложения.

Методы

- `void exit(int status)` - осуществляет выход из работающего приложения;
- `long freeMemory()` - возвращает количество доступной памяти в мобильном устройстве;
- `void gc()` - производит сборку мусора;
- `static Runtime getRuntime()` - возвращает объект времени исполнения во время работы программы;
- `long totalMemory()` - возвращает доступный объем памяти для виртуальной Java-машины.

2.1.11. Класс *Short*

Объектно-ориентированный класс-оболочка для простого типа `Short`.

Конструктор

- `Short(short value)` - создает объект класса `Short`.

Методы

- `boolean equals(Object obj)` - сравнивает объекты;
- `int hashCode()` - возвращает специальный код (хэш-код) объекта класса `Short`;
- `static short parseShort(String s)` - извлекает короткое целое десятичное число из заданного объекта класса `String`;
- `static short parseShort(String s, int radix)` - извлекает короткое целое десятичное число с использованием основания системы исчисления из заданного объекта класса `String`;
- `short shortValue()` - возвращает значение переменной;
- `String toString()` - возвращает объект класса `String`, представляющий короткое целое значение.

Константы

- `static short MAX_VALUE` - максимальное значение;
- `static short MIN_VALUE` - минимальное значение.

2.1.12. Класс String

Создает символьные строки текста.

Конструкторы

- `String()` - создает пустой объект класса `String`, то есть объект не имеет определенной символьной последовательности;
- `String(byte[] bytes)` - создает объект класса `String` из указанного массива байт в соответствующей кодировке, поддерживаемой системой;
- `String(byte[] bytes, int off, int len)` - создает объект класса `String` из указанного массива байт в соответствующей кодировке, поддерживаемой системой. Параметр `off` - это индекс первого байта, от которого происходит конвертация, и параметр `len` указывает на количество байт для конвертации;
- `String(byte[] bytes, int off, int len, String enc)` - создает объект класса `String` из указанного массива байт в соответствующей кодировке, поддерживаемой системой. Параметр `off` - это индекс первого байта, от которого происходит конвертация, параметр `len` указывает на количество байт для конвертации, параметр `enc` - на вид кодировки;
- `String(byte[] bytes, String enc)` - создает объект класса `String` из массива байт в заданной кодировке, указанной в параметре `enc`;
- `String(char[] value)` - создает строку текста из массива символов;
- `String(char[] value, int offset, int count)` - создает строку текста из массива символов. Параметр `offset` - это начало массива, параметр `count` - длина массива;
- `String(String value)` - создает объект класса `String` со значением, определенным в параметре `value`;
- `String(StringBuffer buffer)` - создает объект класса `String` со значением из параметра `buffer`, являющегося объектом класса `StringBuffer`.

Методы

- `char charAt(int index)` - возвращает символ по заданному индексу в параметре `index`, отсчет идет от значения 0;
- `int compareTo(String anotherString)` - сравнивает две строки на основании лексографии;
- `String concat(String str)` - конкатенация двух строк;
- `boolean endsWith(String suffix)` - тестирует строку на окончание подстроки `suffix`;
- `boolean equals(Object anObject)` - сравнивает строки;
- `byte[] getBytes()` - конвертирует строку текста в массив байт в кодировке, по умолчанию заданной системой устройства;
- `byte[] getBytes(String enc)` - конвертирует строку текста в массив байт в кодировке, указанной в параметре `enc`;
- `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)` - производит копирование символов строки в массив символов;

- `int hashCode()` - возвращает специальный код (хэш-код) для строки текста;
- `int indexOf(int ch)` - возвращает положение первого символа в строке текста;
- `int indexOf(int ch, int fromIndex)` - возвращает положение первого символа в строке текста и производит поиск по заданному индексу в параметре `fromIndex`;
- `int indexOf(String str)` - возвращает положение первого символа в подстроке текста;
- `int indexOf(String str, int fromIndex)` - возвращает положение первого символа в подстроке текста и производит поиск по заданному индексу в параметре `fromIndex`;
- `int lastIndexOf(int ch)` - возвращает положение последнего символа в подстроке текста;
- `int lastIndexOf(int ch, int fromIndex)` - возвращает положение последнего символа в подстроке текста и производит поиск по заданному индексу в параметре `fromIndex`;
- `int length()` - возвращает длину определенной строки текста;
- `boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)` - проверяет заданную область текста на совпадение;
- `String replace(char oldChar, char newChar)` - производит замену строки текста на новую строку из параметра `oldChar` в параметр `newChar`;
- `boolean startsWith(String prefix)` - проверяет строку на совпадение с начальным указанием префикса;
- `boolean startsWith(String prefix, int toffset)` - проверяет строку на совпадение с начальным указанием префикса и индекса;
- `String substring(int beginIndex)` - возвращает новую строку текста, являющуюся подстрокой этого текста;
- `String substring(int beginIndex, int endIndex)` - возвращает новую строку текста, являющуюся подстрокой этого текста, заданной начальным и конечным индексами;
- `char[] toCharArray()` - конвертирует строку текста в массив символов;
- `String toLowerCase()` - приводит строку текста к строчному написанию;
- `String toString()` - возвращает строковый объект;
- `String toUpperCase()` - приводит строку текста к заглавному написанию;
- `String trim()` - удаляет имеющиеся пробелы в начале и конце строки;
- `static String valueOf(boolean b)` - возвращает строковое представление логической переменной;
- `static String valueOf(char c)` - возвращает строковое представление переменной типа `char`;

- `static String valueOf(char[] data)` - возвращает строковое представление массива значений типа `char`;
- `static String valueOf(char[] data, int offset, int count)` - возвращает строковое представление массива значений типа `char` с определенного начала по заданной длине;
- `static String valueOf(int i)` - возвращает строковое представление переменной типа `int`;
- `static String valueOf(long l)` - возвращает строковое представление переменной типа `long`;
- `static String valueOf(Object obj)` - возвращает строковое представление объекта.

2.1.13. Класс *StringBuffer*

Класс `StringBuffer` может содержать строки символов любого размера.

Конструкторы

- `StringBuffer()` - создает пустой объект класса `StringBuffer`, то есть объект не имеет определенной символьной последовательности, с длиной не более 16 символов;
- `StringBuffer(int length)` - создает объект класса `StringBuffer` с заданной длиной;
- `StringBuffer(String str)` - создает объект класса `StringBuffer` со значением из параметра `str`, являющегося объектом класса `String`.

Методы

- `StringBuffer append(boolean b)` - добавляет в конец буфера логическую переменную в строковом представлении;
- `StringBuffer append(char c)` - добавляет в конец буфера символ;
- `StringBuffer append(char[] str)` - добавляет в конец буфера массив символов;
- `StringBuffer append(char[] str, int offset, int len)` - добавляет в конец буфера массив символов по начальному индексу и длине массива символов;
- `StringBuffer append(int i)` - добавляет в конец буфера значение типа `int` в строковом представлении;
- `StringBuffer append(long l)` - добавляет в конец буфера значение типа `long` в строковом представлении;
- `StringBuffer append(Object obj)` - добавляет в конец буфера объект в строковом представлении;
- `StringBuffer append(String str)` - добавляет в конец буфера строку текста;
- `int capacity()` - возвращает имеющуюся свободную емкость буфера;
- `char charAt(int index)` - возвращает символ по заданному индексу переменной `index`;

- `StringBuffer delete(int start, int end)` - удаляет подстроку из строки по указанному начальному значению в параметре `start` и конечному в параметре `end`;
- `StringBuffer deleteCharAt(int index)` - удаляет символ из строки по указанному индексу в параметре `index`;
- `void ensureCapacity(int minimumCapacity)` - задает минимальную емкость буфера;
- `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)` - копирует в символьный массив содержимое строкового буфера;
- `StringBuffer insert(int offset, boolean b)` - вставляет в буфер строковое представление логической переменной из параметра `boolean b`;
- `StringBuffer insert(int offset, char c)` - вставляет в буфер символ из параметра `c`;
- `StringBuffer insert(int offset, char[] str)` - вставляет в буфер массив символов из параметра `char[] str`;
- `StringBuffer insert(int offset, int i)` - вставляет в буфер строковое представление переменной типа `int`;
- `StringBuffer insert(int offset, long l)` - вставляет в буфер строковое представление переменной типа `long`;
- `StringBuffer insert(int offset, Object obj)` - вставляет в буфер строковое представление объекта;
- `StringBuffer insert(int offset, String str)` - вставляет в буфер строку текста;
- `int length()` - определяет длину строки;
- `StringBuffer reverse()` - производит замену буфера новой символьной последовательностью;
- `void setCharAt(int index, char ch)` - устанавливает символ в буфер по заданному индексу;
- `void setLength(int newLength)` - устанавливает новую длину для буфера;
- `String toString()` - преобразует содержимое буфера в строку.

2.1.14. Класс *System*

Содержит ряд системных методов.

Методы

- `static void arraycopy(Object src, int src_position, Object dst, int dst_position, int length)` - копирует массив из указанного массива по заданной позиции;
- `static long currentTimeMillis()` - возвращает время, измеряемое в миллисекундах;
- `static void exit(int status)` - производит выход из программы;
- `static void gc()` - совершает сборку мусора;

- `static String getProperty(String key)` - возвращает приоритетное свойство по строковому ключу;
- `static int identityHashCode(Object x)` - возвращает специальный код (хэш-код) объекта.

Константы

- `static PrintStream err` - выходной поток, сообщающий об имеющихся ошибках;
- `static PrintStream out` - выходной поток данных.

2.1.15. Класс Thread

Создает поток в работе приложения для виртуальной Java-машины, мобильных телефонов. Доступна многопоточность.

Конструкторы

- `Thread ()` - создает новый поток;
- `Thread (Runnable target)` - создает новый поток с заданным объектом в параметре `target`, реализующем возможности интерфейса `Runnable`.

Методы

- `static int activeCount()` - возвращает количество задействованных потоков;
- `static Thread currentThread()` - возвращает выполняющийся в данный момент поток;
- `int getPriority()` - узнает приоритет определенного потока;
- `boolean isAlive()` - тестирует поток на работоспособность;
- `void join()` - ожидает окончание потока;
- `void setPriority(int newPriority)` - устанавливает приоритет для потока;
- `static void sleep(long millis)` - останавливает выполнение потока на заданное количество времени, измеряемое в миллисекундах;
- `void start()` - дает команду на выполнение потока посредством метода `run()` интерфейса `Runnable`;
- `String toString()` - возвращает строковое представление потока;
- `static void yield()` - регулирует вызовы последующих потоков, низших по приоритету.

Константы

- `static int MAX_PRIORITY` - максимальный приоритет потока;
- `static int MIN_PRIORITY` - минимальный приоритет потока;
- `static int NORM_PRIORITY` - приоритет по умолчанию.

2.1.16. Класс Throwable

Суперкласс для всех классов, предназначенных для работы с ошибками и исключениями в языке программирования Java.

Конструкторы

- `Throwable()` - создает новый объект класса `Throwable`;
- `Throwable(String message)` - создает новый объект класса `Throwable` с заданным сообщением об ошибках.

Методы

- `String getMessage()` - возвращает сообщение об ошибке;
- `void printStackTrace()` - отслеживает ошибки на выходном потоке;
- `String toString()` - возвращает описание объекта класса `Throwable`.

2.1.17. Исключения

- `Exceptions` - исключения для классов и подклассов;
- `ArithmeticException` - арифметическое исключение;
- `ArrayIndexOutOfBoundsException` - исключение, обрабатывающее неправильный индекс в массиве данных;
- `ArrayStoreException` - исключение, обрабатывающее неправильно заданный тип объекта в массиве объектов;
- `ClassCastException` - неправильно указан подкласс объекта;
- `ClassNotFoundException` - класс не найден;
- `IllegalAccessException` - нет доступа к классу;
- `IllegalArgumentException` - указан неправильный аргумент;
- `IllegalMonitorStateException` - мониторинг объектов;
- `IllegalStateException` - неправильно вызванный метод;
- `IllegalThreadStateException` - неправильные установки потока;
- `IndexOutOfBoundsException` - исключает неверно указанный индекс;
- `InstantiationException` - исключает ситуацию в создании или вызове членов абстрактного класса;
- `InterruptedException` - исключает прерывание потока, находящегося в состоянии ожидания;
- `NegativeArraySizeException` - исключает ситуацию в создании большего размера массива данных, чем было указано при инициализации;
- `NumberFormatException` - неправильное преобразование строки в целочисленный тип данных;
- `RuntimeException` - суперкласс исключений времени исполнения виртуальной машины Java;
- `SecurityException` - менеджер безопасности;
- `StringIndexOutOfBoundsException` - выход индекса за пределы строки.

2.1.18. Ошибки

- `Error` - обобщенная модель ошибок;
- `OutOfMemoryError` - ошибки, связанные с выходом за пределы памяти;
- `VirtualMachineError` - ошибки времени исполнения.

2.2. Пакет java.util

В этом пакете содержатся классы стандартных утилит для создания приложений Java 2 ME. Пакет сильно урезан по сравнению со стандартным пакетом Java 2 SE

2.2.7. Интерфейс *Enumeration*

Декларирует возможность доступа к элементам.

Методы

- `boolean hasMoreElements()` - проверяет соответствующие перечисления на наличие элементов;
- `Object nextElement()` - возвращает последующий элемент перечисления в том случае, если перечисления содержат более одного элемента.

2.2.2. Класс *Calendar*

Необходим для работы с датой и временем, выполняет функции обыкновенного календаря.

Конструктор

- `protected Calendar()` - создает календарь. Язык и часовой пояс задаются по умолчанию.

Методы

- `boolean after(Object when)` - сравнивает два объекта и возвращает значение `true` в том случае, если время, представленное объектом `when`, находится после времени, представленного другим сравниваемым объектом;
- `boolean before(Object when)` - сравнивает два объекта и возвращает значение `true` в том случае, если время, представленное объектом `when`, находится до времени, представленного другим сравниваемым объектом;
- `boolean equals(Object obj)` - сравнивает объекты;
- `int get(int field)` - получает значение определенного поля, например время, день, месяц, год;
- `static Calendar getInstance()` - получает параметры часового пояса и языка по умолчанию;
- `static Calendar getInstance(TimeZone zone)` - получает параметры часового пояса и языка данного региона;
- `Date getTime()` - получает время;
- `protected long getTimeInMillis()` - получает время по Гринвичу, производя запись в виде миллисекунд;
- `TimeZone getTimeZone()` - определяет часовой пояс региона;
- `void set(int field, int value)` - задает определенному полю значение времени;
- `void setTime(Date date)` - устанавливает необходимую дату;
- `protected void setTimeInMillis(long millis)` - устанавливает время по Гринвичу, производя запись в виде миллисекунд;

- `void setTimeZone(TimeZone value)` - устанавливает часовой пояс региона.

Константы

- `static int AM` - формат, отражающий запись времени до полудня;
- `static int AM_PM` - формат, отражающий запись времени до полудня и после полудня;
- `static int APRIL` - значение, указывающее месяц года Апрель;
- `static int AUGUST` - значение, указывающее месяц года Август;
- `static int DATE` - значение, указывающее день;
- `static int DAY_OF_MONTH` - значение, указывающее день и месяц;
- `static int DAY_OF_WEEK` - значение, указывающее день недели;
- `static int DECEMBER` - значение, указывающее месяц года Декабрь;
- `static int FEBRUARY` - значение, указывающее месяц года Февраль;
- `static int FRIDAY` - значение, указывающее день недели пятницу;
- `static int HOUR` - значение, указывающее время;
- `static int HOUR_OF_DAY` - значение, указывающее время и день недели;
- `static int JANUARY` - значение, указывающее месяц года Январь;
- `static int JULY` - значение, указывающее месяц года Июль;
- `static int JUNE` - значение, указывающее месяц года Июнь;
- `static int MARCH` - значение, указывающее месяц года Март;
- `static int MAY` - значение, указывающее месяц года Май;
- `static int MILLISECOND` - формат записи времени в миллисекундах;
- `static int MINUTE` - формат записи времени в минутах;
- `static int MONDAY` - значение, указывающее день недели пятницу;
- `static int MONTH` - месяц;
- `static int NOVEMBER` - значение, указывающее месяц года Ноябрь;
- `static int OCTOBER` - значение, указывающее месяц года Октябрь;
- `static int PM` - формат, отображающий запись времени после полудня;
- `static int SATURDAY` - значение, указывающее день недели суббота;
- `static int SECOND` - устанавливает отображение времени в секундах;
- `static int SEPTEMBER` - значение, указывающее месяц года Сентябрь;
- `static int SUNDAY` - значение, указывающее день недели воскресенье;
- `static int THURSDAY` - значение, указывающее день недели четверг;
- `static int TUESDAY` - значение, указывающее день недели вторник;
- `static int WEDNESDAY` - значение, указывающее день недели среда;
- `static int YEAR` - значение, указывающее год.

2.2.3. Класс Date

Реализует возможность работы с датой.

Конструкторы

- `Date ()` - создает объект класса `Date`;
- `Date(long date)` - создает *объект* класса `Date` с форматом записи `00:00:00`.

Методы

- `boolean equals(Object obj)` - сравнивает две даты;
- `long getTime()` - получает время в миллисекундах;
- `int hashCode()` - возвращает специальный код (хэш-код) объекта класса `Date`;
- `void setTime(long time)` - устанавливает время.

2.2.4. Класс Hashtable

Предоставляет возможность хранения объектов с доступом к ним по определенно заданному ключу.

Конструкторы

- `Hashtable()` - создает пустой объект класса `Hashtable`;
- `Hashtable(int initialCapacity)` - создает объект класса `Hashtable` с заданной вместимостью.

Методы

- `void clear()` - очищает объект класса `Hashtable` от набора имеющихся ключей;
- `boolean contains (Object value)` - определяет наличие различных ключей;
- `boolean containsKey(Object key)` - определяет наличие определенного ключа;
- `Enumeration elements()` - возвращает последовательность имеющихся элементов;
- `Object get(Object key)` - получает необходимый объект, используя при этом заданный для этого объекта ключ;
- `boolean isEmpty()` - проверяет объект класса `Hashtable` на наличие ключей;
- `Enumeration keys()` - возвращает последовательность доступных ключей;
- `Object put(Object key, Object value)` - сохраняет объект и заданный для этого объекта ключ;
- `protected void rehash()` - увеличивает вместимость объекта класса `Hashtable`;
- `Object remove(Object key)` - удаляет указанный ключ;
- `int size()` - определяет количество имеющихся ключей;
- `String toString()` - возвращает строковое представление объекта класса `Hashtable`.

2.2.5. Класс Random

Генератор случайных чисел.

Конструкторы

- `Random()` - создает генератор случайных чисел;

- `Random(long seed)` - создает объект класса `Random`, сгенерировав целое длинное число.

Методы

- `protected int next(int bits)` - генерирует следующее случайное число;
- `int nextInt()` - генерирует целое случайное число из заданной последовательности;
- `long nextLong()` - генерирует целое длинное случайное число из заданной последовательности;
- `void setSeed(long seed)` - устанавливает заданное начальное число для последующей генерации случайных чисел.

2.2.6. Класс *Stack*

Реализует функциональность стека.

Конструктор

- `Stack()` - создает пустой стек.

Методы

- `boolean empty()` - проверяет, пустой созданный стек или нет;
- `Object peek()` - просмотр стека;
- `Object pop()` - удаляет последний объект из стека;
- `Object push(Object item)` - помещает объект в стек;
- `int search(Object o)` - возвращает начальную позицию для первого объекта в стеке.

2.2.7. Класс *Timer*

Реализует возможность работы со временем по принципу таймера.

Конструктор

- `Timer()` - создает таймер.

Методы

- `void cancel()` - закрывает работу таймера;
- `void schedule(TimerTask task, Date time)` - назначает задачу на заданное время;
- `void schedule(TimerTask task, Date firstTime, long period)` - назначает задачу на заданное время с ее последующим повторным выполнением. Частота повторений задается фиксированными промежутками времени;
- `void schedule(TimerTask task, long delay)` - назначает выполнение задачи по прошествии заданного промежутка времени;
- `void schedule(TimerTask task, long delay, long period)` - назначает выполнение задачи по прошествии заданного промежутка времени с ее последующим повторным выполнением. Частота повторений задается фиксированными промежутками времени;
- `void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)` - назначает задачу на заданное время с ее последующим

повторным выполнением. Частота повторений назначается относительно абсолютного времени;

- `void scheduleAtFixedRate(TimerTask task, long delay, long period)` - назначает выполнение задачи по прошествии заданного промежутка времени с ее последующим повторным выполнением. Частота повторений назначается относительно абсолютного времени.

2.2.8. Класс *TimerTask*

Планировщик задач.

Конструктор

- `protected TimerTask()` - создает новую задачу.

Методы

- `boolean cancel()` - отменяет выполнение задачи;
- `abstract void run()` - определяет действие для выполнения планировщиком задач;
- `long scheduledExecutionTime()` - возвращает время выполнения задачи.

2.2.9. Класс *TimeZone*

Устанавливает и определяет часовой пояс.

Конструктор

- `TimeZone()` - создает объект класса `TimeZone`.

Методы

- `static String[] getAvailableIDs()` - получает доступные идентификаторы часового пояса;
- `static TimeZone getDefault()` - получает часовой пояс региона;
- `String getID()` - получает идентификатор часового пояса;
- `abstract int getRawOffset()` - получает время по Гринвичу для часового пояса местонахождения;
- `static TimeZone getTimeZone(String ID)` - получает часовой пояс;
- `abstract boolean useDaylightTime()` - необходим для определения использования часовым поясом перехода на летнее время.

2.2.10. Класс *Vector*

Создает массивы любого размера. Имеет возможность изменять размер заданного массива.

Конструкторы

- `Vector()` - создает пустой массив для содержания объектов;
- `Vector(int initialCapacity)` - создает массив объектов с указанной размерностью;
- `Vector(int initialCapacity, int capacityIncrement)` - создает массив объектов с указанной размерностью и заданным размером дополнений к этому массиву.

Методы

- `void addElement(Object obj)` - добавляет к массиву объектов еще один объект;
- `int capacity()` - узнает текущую вместимость заданного массива объектов;
- `boolean contains(Object elem)` - определяет наличие указанного элемента в массиве объектов;
- `void copyInto(Object[] anArray)` - копирует заданные элементы в массив объектов;
- `Object elementAt(int index)` - возвращает искомый компонент по указанному индексу;
- `Enumeration elements()` - возвращает число имеющихся элементов в массиве данных;
- `void ensureCapacity(int minCapacity)` - увеличивает вместимость массива данных;
- `Object firstElement()` - возвращает самый первый элемент всего массива;
- `int indexOf(Object elem)` - проверяет массив на присутствие объекта;
- `int indexOf(Object elem, int index)` - проверяет массив на присутствие объекта по его индексу в массиве данных;
- `void insertElementAt(Object obj, int index)` - вставляет объект в массив по заданному индексу;
- `boolean isEmpty()` - проверяет массив, не пустой ли он;
- `Object lastElement()` - возвращает самый последний элемент всего массива данных;
- `int lastIndexOf(Object elem)` - возвращает последний индекс данного элемента в массиве;
- `int lastIndexOf(Object elem, int index)` - проверяет последнее присутствие объекта в массиве по его индексу;
- `void removeAllElements()` - удаляет все элементы массива;
- `boolean removeElement(Object obj)` - удаляет элемент массива;
- `void removeElementAt(int index)` - удаляет элемент массива по индексу;
- `void setElementAt(Object obj, int index)` - устанавливает элемент в массив по индексу;
- `void setSize(int newSize)` - задает размер массива;
- `int size()` - определяет размер массива;
- `String toString()` - возвращает строковое представление массива данных;
- `void trimToSize()` - уменьшает размерность массива.

Константы

- `protected int capacityIncrement` - автоматическое увеличение массива на заданное число элементов, то есть шаг увеличения массива;
- `protected int elementCount` - заданное число элементов массива;
- `protected Object[] elementData` - массив данных, в котором сохранены элементы массива.

2.2.11. Исключения

- `EmptyStackException` - указывает на пустой стек;
- `NoSuchElementException` - указывает на отсутствие элементов в определенном перечислении.

2.3. Пакет java.io

Классы этого пакета отвечают за работу с входными и выходными потоками данных.

2.3.1. Интерфейс *DataInput*

Декларирует методы для чтения простых типов во входном потоке данных.

Методы

- `boolean readBoolean()` - читает входной байт данных, и если значение этого байта отлично от 0, то возвращает `true`, иначе возвращается значение `false`;
- `byte readByte()` - производит чтение и возврат одного входного байта;
- `char readChar()` - производит чтение и возврат одного входного символа;
- `void readFully(byte[] b)` - производит чтение входных байт, размещая их в массиве данных;
- `void readFully(byte[] b, int off, int len)` - производит чтение указанных входных байт параметра `len` из параметра `b`;
- `int readInt()` - производит чтение и возврат входных байт типа `int` (четыре байта);
- `long readLong()` - производит чтение и возврат входных байт типа `long` (восемь байт);
- `short readShort()` - производит чтение и возврат входных байт типа `short` (два байта);
- `int readUnsignedByte()` - производит чтение и возврат одного входного байта в диапазоне от 0 до 256;
- `int readUnsignedShort()` - производит чтение и возврат двух входных байт в диапазоне от 0 до 256;
- `String readUTF()` - читает строку текста в формате UTF-8;
- `int skipBytes(int n)` - переходит по входному потоку, минуя пропущенные байты.

2.3.2. Интерфейс *DataOutput*

Декларирует методы для записи простых типов в выходной поток данных.

Методы

- `void write(byte[] b)` - записывает в выходной поток массив байт;
- `void write(byte[] b, int off, int len)` - производит запись определенных байт, указанных в параметре `len`, из параметра `b` выходного потока;

- `void write(int b)` - записывает в выходной поток восемь младших бит;
- `void writeBoolean(boolean v)` - записывает логическую переменную в выходной поток данных;
- `void writeByte(int v)` - записывает в выходной поток восемь младших бит;
- `void writeChar(int v)` - производит запись в выходной поток данных значения типа `char` (один символ - это два байта);
- `void writeChars(String s)` - производит запись в выходной поток данных строки текста;
- `void writeInt(int v)` - производит запись в выходной поток данных значения типа `int` (четыре байта);
- `void writeLong(long v)` - производит запись в выходной поток данных значения типа `long` (восемь байт);
- `void writeShort(int v)` - производит запись в выходной поток данных значения типа `short` (два байта);
- `void writeUTF(String str)` - записывает строку текста в выходной поток данных.

2.3.3. Класс *ByteArrayInputStream*

Совершает чтение входного потока байт из массива данных для дальнейшего размещения их в памяти.

Конструкторы

- `ByteArrayInputStream(byte[] buf)` - создает объект класса `ByteArrayInputStream`, параметр `buf` будет содержать буфер данных;
- `ByteArrayInputStream(byte[] buf, int offset, int length)` — создает объект класса `ByteArrayInputStream`. Параметр `buf` будет содержать буфер данных, параметр `offset` задает смещение от первого байта, а параметр `length` определяет максимальное значения буфера.

Методы

- `int available()` - возвращает количество байт входного потока данных;
- `void close()` - закрывает входной поток, попутно освобождая все захваченные ресурсы этим потоком;
- `void mark(int readAheadLimit)` - устанавливает маркер в заданной позиции потока данных;
- `boolean markSupported()` - проверяет объект класса `ByteArrayInputStream` на поддержку установки и сброса маркера;
- `int read()` - производит чтение каждого последующего байта во входном потоке данных;
- `int read(byte[] b, int off, int len)` - читает определенный байт, указанный в параметре `len`, из параметра `b` входного потока данных;
- `void reset()` - сбрасывает значение к установленному маркеру;
- `long skip(long n)` - пропускает заданные байты входного потока.

Константы

- `protected byte[] buf` - массив байт;
- `protected int count` - последний индекс для чтения из входного потока;

- `protected int mark` - позиция или маркер во входном потоке данных;
- `protected int pos` - последующий индекс для чтения из входного потока.

2.3.4. Класс *ByteArrayOutputStream*

Производит запись потока байт из памяти в массив выходных данных.

Конструкторы

- `ByteArrayOutputStream()` - создает новый выходной поток для записи в массив байт;
- `ByteArrayOutputStream(int size)` - создает новый выходной поток для записи в массив байт с заданным размером.

Методы

- `void close()` - закрывает выходной поток, попутно освобождая все захваченные ресурсы этим потоком;
- `void reset()` - сбрасывает в нуль счетчик выходных данных;
- `int size()` - возвращает текущий размер буфера данных;
- `byte[] toByteArray()` - создает массив байт;
- `String toString()` - производит преобразование содержимого буфера в строку текста;
- `void write(byte[] b, int off, int len)` - записывает определенный байт, указанный в параметре `len`, из параметра `b` в выходной поток;
- `void write(int b)` - записывает байт в выходной поток.

Константы

- `protected byte[] buf` - заданный буфер данных;
- `protected int count` - количество байт в буфере.

2.3.5. Класс *DataInputStream*

Этот класс наследуется от интерфейса `DataInput`, реализуя при этом все его методы.

Конструктор

- `DataInputStream (InputStream in)` - создает новый входной поток данных.

Методы

- `int available()` - возвращает доступное количество байт для чтения из входного потока;
- `void close()` - закрывает входной поток;
- `void mark(int readlimit)` - маркирует заданную позицию во входном потоке;
- `boolean markSupported()` - проверяет объект класса `DataInputStream` на поддержку установки и сброса маркера;
- `int read()` - производит чтение каждого последующего байта во входном потоке данных;
- `int read(byte[] b)` - производит чтение байт из массива во входном потоке данных;

- `int read(byte[] b, int off, int len)` - читает определенный байт, указанный в параметре `len`, из параметра `b` входного потока данных;
- `boolean readBoolean()` - читает входной байт данных, и если значение этого байта отлично от 0, то возвращает `true`, иначе возвращается значение `false`;
- `byte readByte()` - производит чтение и возврат одного входного байта;
- `char readChar()` - производит чтение и возврат одного входного символа;
- `void readFully(byte[] b)` - производит чтение входных байт, размещая их в массиве данных;
- `void readFully(byte[] b, int off, int len)` - производит чтение указанных входных байт параметра `len` из параметра `b`;
- `int readInt()` - производит чтение и возврат входных байт типа `int` (четыре байта);
- `long readLong()` - производит чтение и возврат входных байт типа `long` (восемь байт);
- `short readShort()` - производит чтение и возврат входных байт типа `short` (два байта);
- `int readUnsignedShort()` - производит чтение и возврат двух входных байт в диапазоне от 0 до 256;
- `String readUTF()` - читает строку текста в формате UTF-8;
- `Static String readUTF(DataInput in)` - производит чтение из входного потока строки символов;
- `void reset()` - сбрасывает позицию маркера;
- `long skip(long n)` - пропускает заданные байты входного потока;
- `int skipBytes(int n)` - переходит по входному потоку, минуя пропущенные байты.

Константа

- `protected InputStream in` - входной поток данных.

2.3.6. Класс *DataOutputStream*

Этот класс наследуется от интерфейса `DataOutput`, реализую при этом все его методы.

Конструктор

- `DataOutputStream(OutputStream out)` - создает новый выходной поток данных.

Методы

- `void close()` - закрывает выходной поток;
- `void flush()` - производит сброс потока данных;
- `void write(byte[] b, int off, int len)` - производит запись определенных байт, указанных в параметре `len`, из параметра `b` выходного потока;

- `void write(int b)` - записывает в выходной поток восемь младших бит;
- `void writeBoolean(boolean v)` - записывает логическую переменную в выходной поток данных;
- `void writeByte(int v)` - записывает в выходной поток восемь младших бит;
- `void writeChar(int v)` - производит запись в выходной поток данных значение типа `char` (один символ - это два байта);
- `void writeChars(String s)` - производит запись в выходной поток данных строку текста;
- `void writeInt(int v)` - производит запись в выходной поток данных значения типа `int` (четыре байта);
- `void writeLong(long v)` - производит запись в выходной поток данных значения типа `long` (восемь байт);
- `void writeShort(int v)` - производит запись в выходной поток данных значения типа `short` (два байта);
- `void writeUTF(String str)` - записывает строку текста в выходной поток данных.

Константа

- `protected OutputStream out` - выходной поток данных.

2.3.7. Класс `InputStream`

Абстрактный класс, предназначенный для работы с входным потоком байт.

Конструктор

- `InputStream()` - конструктор абстрактного класса `InputStream`.

Методы

- `int available()` - возвращает доступное количество байт для чтения из входного потока;
- `void close()` - закрывает входной поток;
- `void mark(int readlimit)` - маркирует заданную позицию во входном потоке;
- `boolean markSupported()` - проверяет объекты на поддержку установки и сброса маркера;
- `abstract read()` - производит чтение каждого последующего байта во входном потоке данных;
- `int read(byte[] b)` - производит чтение байт из массива во входном потоке данных;
- `int read(byte[] b, int off, int len)` - читает определенный байт, указанный в параметре `len`, из параметра `b` входного потока данных;
- `void reset()` - сбрасывает позицию маркера;
- `long skip(long n)` - пропускает заданные байты входного потока.

2.3.8. Класс *InputStreamReader*

Наследуется от класса `Reader`, реализуя методы для чтения символьных данных входного потока с перекодировкой.

Конструкторы

- `InputStreamReader(InputStream is)` - создает объект класса `InputStreamReader`, используя кодировку по умолчанию;
- `InputStreamReader(InputStream is, String enc)` - создает объект класса `InputStreamReader`, используя кодировку, заданную в параметре `enc`.

Методы

- `void close()` - закрывает поток;
- `void mark(int readAheadLimit)` - маркирует позицию в потоке;
- `boolean markSupported()` - определяет поддержку маркировки и сброса позиции в потоке;
- `int read()` - производит чтение символа;
- `int read(char[] cbuf, int off, int len)` - производит чтение символа в массив;
- `boolean ready()` - определяет готовность потока на чтение данных из него;
- `void reset()` - сбрасывает позицию маркера;
- `long skip(long n)` - пропускает заданные символы.

2.3.9. Класс *OutputStream*

Абстрактный класс, предназначенный для работы с выходным потоком байт.

Конструктор

- `OutputStream()` - конструктор абстрактного класса `OutputStream`.

Методы

- `void close()` - закрывает выходной поток;
- `void flush()` - осуществляет сброс выходного потока;
- `void write(byte[] b)` - записывает массив байт в выходной поток;
- `void write(byte[] b, int off, int len)` - производит запись определенных байт, указанных в параметре `len`, из параметра `b` выходного потока;
- `abstract void write(int b)` - записывает определенный байт в выходной поток.

2.3.10. Класс *OutputStreamWriter*

Наследуется от класса `Writer`, реализуя методы для записи символьных данных в выходной поток с перекодировкой.

Конструкторы

- `OutputStreamWriter(OutputStream os)` - создает объект `OutputStreamWriter`, используя кодировку по умолчанию;

- `OutputStreamWriter(OutputStream os, String enc)` - создает объект `OutputStreamWriter`, используя кодировку, заданную в параметре `enc`.

Методы

- `void close()` - закрывает поток данных;
- `void flush()` - сбрасывает поток данных;
- `void write(char[] cbuf, int off, int len)` - производит запись определенных символов, указанных в параметре `len`, из параметра `b` выходного потока;
- `void write(int c)` - записывает один символ;
- `void write(String str, int off, int len)` - производит запись определенной части строки текста, указанной в параметре `len`, из параметра `b` выходного потока.

2.3.11. Класс *PrintStream*

Расширяет выходной поток способностью печати данных.

Конструктор

- `PrintStream(OutputStream out)` - формирует объект класса `PrintStream`, отвечающий за создание нового потока печати.

Методы

- `boolean checkError()` - проверяет состояние потока;
- `void close()` - закрывает поток данных;
- `void flush()` - сбрасывает поток данных;
- `void print(boolean b)` - производит печать логического значения;
- `void print(char c)` - производит печать значения типа `char`;
- `void print(char[] s)` - производит печать массива символов;
- `void print(int i)` - производит печать значения типа `int`;
- `void print(long l)` - производит печать значения типа `long`;
- `void print(Object obj)` - производит печать объекта;
- `void print(String s)` - производит печать строки текста;
- `void println()` - производит печать, заканчивая переводом на новую строку;
- `void println(boolean x)` - производит печать логического значения, заканчивая печать переводом на новую строку;
- `void println(char x)` - производит печать значения типа `char`, заканчивая печать переводом на новую строку;
- `void println(char[] x)` - производит печать массива символов, заканчивая печать переводом на новую строку;
- `void println(int x)` - производит печать значения типа `int`, заканчивая печать переводом на новую строку;
- `void println(long x)` - производит печать значения типа `long`, заканчивая печать переводом на новую строку;
- `void println(Object x)` - производит печать объекта, заканчивая печать переводом на новую строку;

- `void println(String x)` - производит печать строки текста, заканчивая печать переводом на новую строку;
- `protected void setError()` - приводит поток, содержащий некоторые ошибки, к состоянию `true`;
- `void write(byte[] buf, int off, int len)` - производит запись определенных байт, указанных в параметре `len`, из параметра `b` потока печати;
- `void write(int b)` - записывает байт в поток печати.

2.3.12. Класс *Reader*

Абстрактный класс, предназначенный для чтения символьных потоков данных.

Конструкторы

- `protected Reader()` - создает новый поток для чтения;
- `protected Reader(Object lock)` - создает новый поток для чтения, синхронизирующийся с параметром `lock`.

Методы

- `abstract void close()` - закрывает поток данных;
- `void mark(int readAheadLimit)` - маркирует определенную позицию в потоке;
- `boolean markSupported()` - проверяет поддержку маркировки и сброса позиции в потоке;
- `int read()` - производит чтение символа;
- `int read(char[] cbuf)` - производит чтение массива символов;
- `abstract int read(char[] cbuf, int off, int len)` - производит чтение в массив;
- `boolean ready()` - определяет готовность потока для чтения данных;
- `void reset()` - сбрасывает позицию маркера;
- `long skip(long n)` - пропускает заданные символы.

Константа

- `protected Object lock` - используется при синхронизации определенных действий в потоке.

2.3.13. Класс *Writer*

Абстрактный класс, предназначенный для записи символьных данных в выходной поток.

Конструкторы

- `protected Writer()` - создает новый символьный поток для записи данных;
- `protected Writer(Object lock)` - создает новый символьный поток для записи данных, синхронизирующийся с параметром `lock`.

Методы

- `abstract void close()` - закрывает поток данных;
- `abstract void flush()` - сбрасывает поток данных;
- `void write(char[] cbuf)` - производит запись в массив символов;

- `abstract void write(char[] cbuf, int off, int len)` -совершает запись заданной части массива символов;
- `void write(int c)` - записывает один-единственный символ;
- `void write(String str)` - записывает строку текста;
- `void write(String str, int off, int len)` - совершает запись заданной части строки текста.

Константа

- `protected Object lock` - используется при синхронизации определенных действий в потоке.

2.3.14. Исключения

- `EOFException` - сигнализирует о конце файла;
- `InterruptedException` - сигнализирует о прерванном действии по вводу-выводу;
- `IOException` - указывает на исключение ввода-вывода;
- `UnsupportedEncodingException` - указывает на невозможность перекодировки;
- `UTFDataFormatException` - сигнализирует о прочтении строки формата UTF-8.

2.4. Пакет `javaх.microedition.io`

Этот пакет обеспечивает мобильное устройство связью с сетью.

2.4.1. Интерфейс `CommConnection`

Находит последовательный порт.

Методы

- `int getBaudRate()` - получает скорость передачи данных в бодах для связи;
- `int setBaudRate(int baudrate)` - устанавливает скорость передачи данных в бодах для связи.

2.4.2. Интерфейс `Connection`

Общий тип всей связи с сетью.

Метод

- `void close()` - закрывает имеющуюся связь с сетью.

2.4.3. Интерфейс `ContentConnection`

Определяет связь с потоком.

Методы

- `String getEncoding()` - определяет кодировку потока;
- `long getLength()` - возвращает продолжительность соединения;
- `String getType()` - возвращает тип соединения.

2.4.4. Интерфейс *Datagram*

Общий интерфейс дейтограммы.

Методы

- `String getAddress()` - получает адрес дейтограммы;
- `byte[] getData()` - получает данные;
- `int getLength()` - получает продолжительность соединения;
- `int getOffset()` - получает смещение;
- `void reset()` - производит сброс или обнуление указателей для чтения и записи;
- `void setAddress (Datagram reference)` - устанавливает адрес дейтограммы, взятый с другой выбранной дейтограммы;
- `void setAddress (String addr)` - устанавливает адрес дейтограммы;
- `void setData (byte[] buffer, int offset, int len)` - устанавливает в буфере смещение и длину;
- `void setLength(int len)` - устанавливает длину.

2.4.5. Интерфейс *DatagramConnection*

Определяет возможность связи дейтограммы.

Методы

- `int getMaximumLength()` - получает максимальную длину дейтограммы;
- `int getNominalLength()` - получает номинальную длину дейтограммы;
- `Datagram newDatagram(byte[] buf, int size)` - создает новый объект дейтограммы с указанным размером буфера;
- `Datagram newDatagram(byte[] buf, int size, String addr)` - создает новый объект дейтограммы с указанным размером буфера и адресом ввода-вывода;
- `Datagram newDatagram(int size)` - создает новый объект дейтограммы определенного размера;
- `Datagram newDatagram(int size, String addr)` - создает новый объект дейтограммы определенного размера и с указанием адреса ввода-вывода;
- `void receive(Datagram dgram)` - принимает дейтограмму;
- `void send(Datagram dgram)` - отправляет дейтограмму.

2.4.6. Интерфейс *HttpConnection*

Декларирует методы и константы для протокола соединения HTTP.

Методы

- `long getDate()` - возвращает данные;
- `String getFile()` - возвращает часть файла по адресу URL;
- `String getHeaderField(int n)` - возвращает заголовок файла по индексу;
- `String getHeaderField(String name)` - возвращает заголовок фай-

- `long getHeaderFieldDate(String name, long def)` - возвращает значение заданного поля для даты;
- `int getHeaderFieldInt (String name, int def)` - возвращает значение заданного поля для номера;
- `String getHeaderFieldKey(int n)` - получает файл заголовка по ключу;
- `String getHost()` - возвращает информацию о соединении;
- `long getLastModified()` - возвращает значение модифицированного заголовка;
- `int getPort()` - возвращает номер порта соединения;
- `String getProtocol()` - возвращает имя протокола;
- `String getQuery()` - возвращает запрос на соединение;
- `String getRequestMethod()` - определяет текущий метод произведенного запроса;
- `String getRequestProperty(String key)` - возвращает свойства запроса для соединения;
- `int getResponseCode()` - возвращает код состояния протокола HTTP;
- `String getResponseMessage()` - возвращает сообщение о коде состояния протокола HTTP;
- `String getURL()` - возвращает адрес соединения;
- `void setRequestMethod(String method)` - задает метод для запроса адреса;
- `void setRequestProperty(String key, String value)` - устанавливает свойства производимого запроса.

Константы

- `static String GET` - метод соединения по протоколу HTTP;
- `static String HEAD` - основной метод соединения по протоколу HTTP;
- `static int HTTP_ACCEPTED` - запрос принят, но не был обработан;
- `static int HTTP_BAD_GATEWAY` - недопустимый ответ от сервера;
- `static int HTTP_BAD_METHOD` - непозволительный метод для запроса;
- `static int HTTP_BAD_REQUEST` - запрос не был принят;
- `static int HTTP_CLIENT_TIMEOUT` - запрос не произведен в момент связи с сервером;
- `static int HTTP_CONFLICT` - запрос не закончен из-за конфликта;
- `static int HTTP_CREATED` - запрос произведен;
- `static int HTTP_ENTITY_TOO_LARGE` - отказ обработки запроса из-за большого размера;
- `static int HTTP_EXPECT_FAILED` - запрос ожидания не выполнен;
- `static int HTTP_FORBIDDEN` - запрос принят, но выполнен не будет;
- `static int HTTP_GATEWAY_TIMEOUT` - сервер не получил своевременного ответа;
- `static int HTTP_GONE` - затребованный ресурс не найден;
- `static int HTTP_INTERNAL_ERROR` - неожиданная ошибка на сервере;
- `static int HTTP_LENGTH_REQUIRED` - отказ от приема запроса.

2.4.7. Интерфейс *HttpsConnection*

Декларирует методы и константы для безопасного сетевого соединения.

Методы

- `int getPort()` - возвращает сетевой номер порта для соединения;
- `SecurityInfo getSecurityInfo()` - получает информацию о безопасности связи.

2.4.8. Интерфейс *InputConnection*

Интерфейс для создания входной связи с сетью.

Методы

- `DataInputStream openDataInputStream()` - открывает и возвращает поток ввода данных для конкретного соединения;
- `InputStream openInputStream()` - открывает и возвращает входной поток для конкретного соединения.

2.4.9. Интерфейс *OutputConnection*

Интерфейс для создания выходной связи с сетью.

Методы

- `DataOutputStream openDataOutputStream()` - открывает и возвращает поток вывода данных для конкретного соединения;
- `OutputStream openOutputStream()` - открывает и возвращает выходной поток для конкретного соединения.

2.4.10. Интерфейс *SecureConnection*

Определяет безопасную связь с сетью.

Метод

- `SecurityInfo getSecurityInfo()` - получает информацию о безопасности связи.

2.4.11. Интерфейс *SecurityInfo*

Имеет в своем составе методы для получения информации сетевой связи.

Методы

- `String getCipherSuite()` - возвращает название используемого шифра связи;
- `String getProtocolName()` - получает имя используемого протокола соединения;
- `String getProtocolVersion()` - получает версию используемого протокола;
- `Certificate getServerCertificate()` - осуществляет возврат сертификата безопасности соединения.

2.4.12. Интерфейс *ServerSocketConnection*

Реализует связь с сервером.

Методы

- `String getLocalAddress()` - получает локальный адрес связи с разъемом (socket);
- `int getLocalPort()` - получает локальный адрес связи с портом.

2.4.13. Интерфейс *SocketConnection*

Находит разъем (socket) для потока связи.

Методы

- `String getAddress()` - получает адрес связи;
- `String getLocalAddress()` - получает локальный адрес связи;
- `int getLocalPort()` - получает локальный порт соединения;
- `int getPort()` - получает порт соединения;
- `int getSocketOption(byte option)` - получает необходимую опцию разъема для создания соединения;
- `void setSocketOption(byte option, int value)` - устанавливает необходимую опцию разъема для создания соединения.

Константы

- `static byte DELAY` - опция малого разъема (0);
- `static byte KEEPALIVE` - опция поддержки особенностей разъема (2);
- `static byte LINGER` - опция ждущего режима обработки вывода данных (1);
- `static byte RCVBUF` - опция для определенного буфера получения (3);
- `static byte SNDBUF` - опция для определенного буфера отправки (4).

2.4.14. Интерфейс *StreamConnection*

Этот интерфейс определяет связь с потоком и не имеет методов и констант.

2.4.15. Интерфейс *StreamConnectionNotifier*

Определяет возможность всей связи.

Метод

- `StreamConnection acceptAndOpen()` - возвращает разъем сервера, с которым произошло соединение.

2.4.16. Интерфейс *UDPDatagramConnection*

Реализует связь с дейтограммой.

Методы

- `String getLocalAddress()` - получает локальный адрес связи с дейтограммой;
- `int getLocalPort()` - получает локальный порт связи с дейтограммой.

2.4.17. Класс *Connector*

Класс для создания объектов связи.

Методы

- `static Connection open(String name)` - создает и открывает соединение;
- `static Connection open(String name, int mode)` - создает и открывает соединение по адресу и режиму соединения;
- `static Connection open(String name, int mode, boolean timeouts)` - создает и открывает соединение по адресу, режиму соединения и исключению времени ожидания связи;
- `static DataInputStream openDataInputStream(String name)` - создает и открывает входной поток данных;
- `static DataOutputStream openDataOutputStream(String name)` - создает и открывает выходной поток данных;
- `static InputStream openInputStream(String name)` - создает и открывает входной поток;
- `static OutputStream openOutputStream(String name)` - создает и открывает выходной поток.

Константы

- `static int READ` - режим доступа только для чтения данных;
- `static int READ_WRITE` - режим доступа для чтения и записи данных;
- `static int WRITE` - режим доступа только для записи данных.

2.4.18. Класс *PushRegistry*

Класс для поддержания списков связей.

Методы

- `static String getFilter(String connection)` - получает заданный фильтр соединения;
- `static String getMIDlet(String connection)` - получает заданный мидлет (`MIDlet`) для соединения;
- `static String[] listConnections(boolean available)` - возвращает весь список подключений для комплекта мидлетов (`MIDlet suite`);
- `static long registerAlarm(String midlet, long time)` - производит установку времени для запуска приложения;
- `static void registerConnection(String connection, String midlet, String filter)` - производит установку времени для запуска соединения;
- `static boolean unregisterConnection(String connection)` - удаляет регистрацию соединения.

2.4.19. Исключение

- `ConnectionNotFoundException` - указывает на отсутствие связи.

2.5. Пакет javax.microedition.lcdui

Пакет классов пользовательского интерфейса (UI) для создания полей, форм, уведомлений, текста и т. д.

2.5.1. Интерфейс *Choice*

Содержит набор методов, создающих возможность выбора заданных элементов.

Методы

- `int append(String stringPart, Image imagePart)` - добавляет элемент к набору элементов в конец всего имеющегося списка элементов;
- `void delete(int elementNum)` - удаляет элемент по заданному номеру;
- `void deleteAll()` - удаляет все элементы;
- `int getFitPolicy()` - предоставляет предпочтительную экранную позицию;
- `Font getFont (int elementNum)` - получает шрифт для элемента, заданного по номеру;
- `Image getImage(int elementNum)` - получает изображение для элемента, заданного по номеру;
- `int getSelectedFlags(boolean[] selectedArray_return)` - производит запрос на состояние элементов массива;
- `int getSelectedIndex()` - получает выбранный индекс элемента;
- `String getString(int elementNum)` - получает строку текста по заданному номеру;
- `void insert(int elementNum, String stringPart, Image imagePart)` - производит вставку элемента по заданному номеру в набор имеющихся элементов;
- `boolean isSelected(int elementNum)` - получает логическое значение, определяющее выбор того или иного элемента из набора элементов;
- `void set(int elementNum, String stringPart, Image imagePart)` - устанавливает новую строку текста с изображением по заданному номеру, заменяя предыдущую запись;
- `void setFitPolicy(int fitPolicy)` - устанавливает предпочтительную экранную позицию;
- `void setFont(int elementNum, Font font)` - устанавливает шрифт для заданного элемента;
- `void setSelectedFlags(boolean[] selectedArray)` - устанавливает состояние элементов массива;
- `void setSelectedIndex(int elementNum, boolean selected)` - устанавливает состояние элемента;
- `int size()` - определяет количество элементов в наборе элементов.

Константы

- `static int EXCLUSIVE` - эксклюзивный выбор;
- `static int IMPLICIT` - неявный выбор;
- `static int MULTIPLE` - множественный выбор;

- `static int POPUP` - всплывающий вид выбора;
- `static int TEXT_WRAP_DEFAULT` - текстовое сопровождение элемента будет находиться по умолчанию;
- `static int TEXT_WRAP_OFF` - текстовое сопровождение элемента должно находиться на одной строке;
- `static int TEXT_WRAP_ON` - текстовое сопровождение элемента находится на любом количестве строк.

2.5.2. Интерфейс *CommandListener*

Реализует возможность обработчика событий.

Метод

- `void commandAction(Command c, Displayable d)` -обработчик событий.

2.5.3. Интерфейс *ItemCommandListener*

Реализует возможность получения событий от объектов класса `Item`.

Метод

- `void commandAction(Command c, Item item)` - обработчик событий.

2.5.4. Интерфейс *ItemStateListener*

Используется при получении событий о состоянии объектов класса `Item`, встроенных в `Form`.

Метод

- `void itemStateChanged(Item item)` - определяет состояние объекта класса `Item`.

2.5.5. Класс *Alert*

Создает различные информационные сообщения.

Конструкторы

- `Alert(String title)` - создает пустое уведомление с заголовком;
- `Alert(String title, String alertText, Image alertImage, AlertType alertType)` - создает уведомление с заголовком, текстом, изображением и типом уведомления.

Методы

- `void addCommand(Command cmd)` - добавляет команду;
- `int getDefaultTimeout()` - получает время для представления уведомления;
- `Image getImage()` - получает изображение для экрана, представленного классом `Alert`;
- `Gauge getIndicator()` - этот метод позволяет воспользоваться графическим измерителем класса `Gauge`;

- `String getString()` - получает текстовую строку;
- `int getTimeout()` - получает заданное время для представления уведомления;
- `AlertType getType()` - определяет тип используемого уведомления;
- `void removeCommand(Command cmd)` - удаляет команду;
- `void setCommandListener (CommandListener l)` - устанавливает обработчик событий;
- `void setImage(Image img)` - устанавливает изображение;
- `void setIndicator(Gauge indicator)` - устанавливает индикатор измерителя для использования класса Gauge;
- `void setString(String str)` - устанавливает строку текста;
- `void setTimeout(int time)` - устанавливает время;
- `void setType(AlertType type)` - устанавливает тип уведомлений или информационных сообщений.

Константы

- `static Command DISMISS_COMMAND` - команда отклонена;
- `static int FOREVER` - определяет постоянный показ уведомления.

2.5.6. Класс *AlertType*

Отображает тип уведомления.

Конструктор

- `protected AlertType()` - закрытый конструктор подкласса.

Метод

- `boolean playSound(Display display)` - воспроизводит звук.

Константы

- `static AlertType ALARM` - тревога;
- `static AlertType CONFIRMATION` - подтверждение;
- `static AlertType ERROR` - ошибка;
- `static AlertType INFO` - информация;
- `static AlertType WARNING` - предупреждение.

2.5.7. Класс *Canvas*

Абстрактный класс, обеспечивающий графическую прорисовку различных элементов на экране телефона.

Конструктор

- `protected Canvas()` - создает новый объект класса Canvas.

Методы

- `int getGameAction (int keyCode)` - связывает игровые действия с заданным ключевым кодом;
- `int getKeyCode(int gameAction)` - получает ключевой код игровых действий;

- `String getKeyname (int keyCode)` - получает ключевой код для клавиши;
- `boolean hasPointerEvents()` - проверяет устройство на поддержку работы с указателем;
- `boolean hasPointerMotionEvents()` - проверяет поддержку устройством перемещения указателя;
- `boolean hasRepeatEvents()` - проверяет устройство на поддержку работы с повторными событиями;
- `protected void hideNotify()` - выполняет запрос после удаления объекта класса `Canvas` с дисплея;
- `boolean isDoubleBuffered()` - осуществляет двойную буферизацию;
- `protected void keyPressed(int keyCode)` - вызывается при нажатии клавиши;
- `protected void keyReleased(int keyCode)` - вызывается при отпускании нажатой клавиши;
- `protected void keyRepeated(int keyCode)` - повторное нажатие клавиши;
- `protected abstract void paint (Graphics g)` - прорисовка, или рендеринг, графики на экране телефона;
- `protected void pointerDragged(int x, int y)` - определяет перемещение курсора;
- `protected void pointerPressed(int x, int y)` - определяет позицию курсора, в которой должно производиться нажатие определенной клавиши;
- `protected void pointerReleased(int x, int y)` - определяет позицию курсора в момент отпускания определенной клавиши;
- `void repaint ()` - повторяет прорисовку;
- `void repaint (int x, int y, int width, int height)` - повторяет прорисовку заданной области;
- `void serviceRepaints ()` - повтор прорисовки дисплея;
- `void setFullScreenMode (boolean mode)` - контроль над полноэкранным режимом отображения;
- `protected void showNotify()` - выполняет запрос до вывода объекта класса `Canvas` на дисплей;
- `protected void sizeChanged(int w, int h)` - изменяет размер.

Константы

- `static int DOWN` - движение вниз;
- `static int FIRE` - обычно используется в играх и реализует стрельбу из оружия;
- `static int GAME_A` - игровая клавиша A;
- `static int GAME_B` - игровая клавиша B;
- `static int GAME_C` - игровая клавиша C;
- `static int GAME_D` - игровая клавиша D;
- `static int KEY_NUM0` - клавиша 0;

- static int KEY_NUM1 - клавиша 1
- static int KEY_NUM2 - клавиша 2
- static int KEY_NUM3 - клавиша 3
- static int KEY_NUM4 - клавиша 4
- static int KEY_NUM5 - клавиша 5
- static int KEY_NUM6 - клавиша 6
- static int KEY_NUM7 - клавиша 7
- static int KEY_NUM8 - клавиша 8
- static int KEY_NUM9 - клавиша 9
- static int KEY_POUND - клавиша #;
- static int KEY_STAR - клавиша *;
- static int LEFT - движение влево;
- static int RIGHT - движение вправо;
- static int UP - движение вверх.

2.5.8. Класс *ChoiceGroup*

Встраиваемая группа выбираемых элементов. Интегрируется в классе *Form*, наследуется от класса *Item* и реализует интерфейс *Choice*.

Конструктор

- *ChoiceGroup(String label, int choiceType)* - создает пустой список элементов группы, определяя заголовок и тип группы элементов;
- *ChoiceGroup(String label, int choiceType, String[] stringElements, Image[] imageElements)* - создает группу элементов, определяя заголовок, тип группы элементов, текст и изображение для каждого элемента группы.

Методы

- *int append(String stringPart, Image imagePart)* - добавляет элемент в группу;
- *void delete(int elementNum)* - удаляет заданный элемент из группы;
- *void deleteAll()* - удаляет все элементы;
- *int getFitPolicy()* - предоставляет предпочтительную экранную позицию;
- *Font getFont(int elementNum)* - получает используемый шрифт элемента группы;
- *Image getImage(int elementNum)* - получает изображение для элемента группы;
- *int getSelectedFlags(boolean[] selectedArray_return)* - возвращает значение *Boolean* для группы элементов;
- *int getSelectedIndex()* - возвращает индекс выбранного элемента группы;
- *String getString(int elementNum)* - получает строку текста по номеру;
- *void insert(int elementNum, String stringPart, Image imagePart)* - вставляет элемент в группу;
- *boolean isSelected(int elementNum)* - получает выбранную логическую величину;

- `void set(int elementNum, String stringPart, Image imagePart)` - устанавливает текст и изображение в заданный элемент группы, при этом удаляя предыдущую запись;
- `void setFitPolicy(int fitPolicy)` - устанавливает предпочтительную экранную позицию;
- `void setFont(int elementNum, Font font)` - устанавливает шрифт заданному элементу;
- `void setSelectedFlags(boolean[] selectedArray)` - устанавливает состояние элементов группы;
- `void setSelectedIndex(int elementNum, boolean selected)` - устанавливает особое состояние для элемента группы при использовании множественного типа;
- `int size()` - возвращает количество используемых элементов группы.

2.5.9. Класс *Command*

Инкапсулирует командные действия, при этом не определяя фактических действий на команды, а лишь содержит информацию.

Конструкторы

- `Command(String label, int commandType, int priority)` - создает команду для дальнейшей обработки. Команда содержит название, тип и приоритет выполнения;
- `Command(String shortLabel, String longLabel, int commandType, int priority)` - создает команду для дальнейшей обработки. Команда содержит короткое и длинное названия, тип и приоритет выполнения.

Методы

- `int getCommandType()` - получает тип используемой команды;
- `String getLabel()` - получает метку или название команды;
- `String getLongLabel()` - получает длинную метку или название команды;
- `int getPriority()` - получает установленный приоритет команды.

Константы

- `static int BACK` - назад;
- `static int CANCEL` - отмена;
- `static int EXIT` - выход;
- `static int HELP` - помощь;
- `static int ITEM` - новый экран, ассоциирующийся с экраном, от которого происходит переход;
- `static int OK` - хорошо;
- `static int SCREEN` - новый экран;
- `static int STOP` - стоп.

2.5.10. Класс *Custom Item*

Создает возможность в отображении новых графических элементов, встроенных в класс `Form`.

Конструктор

- `protected CustomItem(String label)` - конструктор абстрактного класса `CustomItem`.

Методы

- `int getGameAction(int keyCode)` - получает игровые действия по коду клавиши телефона;
- `protected int getInteractionModes()` - получает все доступные режимы взаимодействия;
- `protected abstract int getMinContentHeight()` - получает минимальную высоту заданной области дисплея;
- `protected abstract int getMinContentWidth()` - получает минимальную ширину заданной области дисплея;
- `protected abstract int getPrefContentHeight(int width)` - получает предпочтительную высоту заданной области дисплея;
- `protected abstract int getPrefContentWidth(int height)` - получает предпочтительную ширину заданной области дисплея;
- `protected void hideNotify()` - уведомляет о недоступности;
- `protected void invalidate()` - сигнализирует об изменении размера или местонахождения элемента;
- `protected void keyPressed(int keyCode)` - обрабатывает нажатие клавиши;
- `protected void keyReleased(int keyCode)` - обрабатывает отпускание клавиши;
- `protected void keyRepeated(int keyCode)` - обрабатывает повторное нажатие клавиши;
- `protected abstract void paint(Graphics g, int w, int h)` - рисует компоненты;
- `protected void pointerDragged(int x, int y)` - осуществляет поддержку перьевого ввода;
- `protected void pointerPressed(int x, int y)` - в месте установки указателя были произведены действия по нажатию определенной клавиши;
- `protected void pointerReleased(int x, int y)` - в месте установки указателя были произведены действия по отпусканию нажатой клавиши;
- `protected void repaint()` - перерисовывает экран;
- `protected void repaint(int x, int y, int w, int h)` - перерисовывает заданную область экрана;
- `protected void showNotify()` - уведомление о возможности получения действий;
- `protected void sizeChanged(int w, int h)` - изменяет размер.

Константы

- `protected static int KEY_PRESS` - нажатие клавиши;
- `protected static int KEY_RELEASE` - отпускание клавиши;
- `protected static int KEY_REPEAT` - повторное нажатие клавиши;

- `protected static int NONE` - нет действий;
- `protected static int POINTER_DRAG` - перетаскивание;
- `protected static int POINTER_PRESS` - указатель нажат;
- `protected static int POINTER_RELEASE` - указатель отпущен;
- `protected static int TRAVERSE_HORIZONTAL` - горизонтальный обход;
- `protected static int TRAVERSE_VERTICAL` - вертикальный обход.

2.5.11. Класс *DateField*

Класс, представляющий работу с датой и временем. Интегрируется в класс `Form`, наследуется от класса `Item`.

Конструкторы

- `DateField(String label, int mode)` - создает объект класса `DateField` с указанием метки и режима отображения объекта;
- `DateField(String label, int mode, TimeZone timeZone)` - создает объект класса `DateField` с указанием метки, режима отображения объекта и часового пояса.

Методы

- `Date getDate()` - возвращает текущую дату;
- `void setDate(Date date)` - устанавливает новую дату;
- `int getInputMode()` - получает установленные компоненты `DATE`, `TIME` или `DATE TIME`;
- `void setInputMode(int mode)` - устанавливает компоненты `DATE`, `TIME` или `DATE_TIME`.

Константы

- `static int DATE` - дата;
- `static int DATE_TIME` - дата и время;
- `static int TIME` - только время.

2.5.12. Класс *Display*

Менеджер дисплея, определяющий, какой из объектов будет представлен на дисплее.

Методы

- `void callSerially(Runnable r)` - производит запрос на вызов метода `run()` для объекта класса `Runnable`;
- `boolean flashBacklight(int duration)` - создает эффект подсветки;
- `int getBestImageHeight(int imageType)` - получает оптимальную высоту для изображения на экране;
- `int getBestImageWidth(int imageType)` - получает оптимальную ширину для изображения на экране;
- `int getBorderStyle(boolean highlighted)` - штриховой стиль бордюра;

- `int getColor(int colorSpecifier)` - возвращает цвет;
- `Displayable getCurrent()` - получает текущий объект `Displayable` для используемого мидлета;
- `static Display getDisplay(MIDlet m)` - получает уникальный объект `Display` для используемого мидлета;
- `boolean isColor()` - получает информацию о поддержке цвета в мобильном устройстве;
- `int numAlphaLevels()` - получает количество альфа-прозрачных уровней;
- `int numColors()` - получает количество цветов, поддерживаемых мобильным устройством;
- `void setCurrent(Alert alert, Displayable nextDisplayable)` - делает видимым на экране объект класса `Alert`;
- `void setCurrent(Displayable nextDisplayable)` - делает видимым на экране последующий объект класса `Displayable`;
- `void setCurrentItem(Item item)` - делает видимым на экране объект класса `Item`;
- `boolean vibrate(int duration)` - запрос на поддержку вибрации.

Константы

- `static int ALERT` - тип изображений для уведомлений;
- `static int CHOICE_GROUP_ELEMENT` - тип изображения для класса `ChoiceGroup`;
- `static int COLOR_BACKGROUND` - цветовой компонент, используется методом `getColor()`;
- `static int COLOR_BORDER` - цветовой компонент, используется методом `getColor()`;
- `static int COLOR_FOREGROUND` - цветовой компонент, используется методом `getColor()`;
- `static int COLOR_HIGHLIGHTED_BACKGROUND` - цветовой компонент, используется методом `getColor()`;
- `static int COLOR_HIGHLIGHTED_BORDER` - цветовой компонент, используется методом `getColor()`;
- `static int COLOR_HIGHLIGHTED_FOREGROUND` - цветовой компонент, используется методом `getColor()`;
- `static int LIST_ELEMENT` - тип изображения для класса `List`.

2.5.13. Класс Displayable

Абстрактный класс, содержит иерархию классов пользовательского интерфейса.

Методы

- `void addCommand(Command cmd)` - добавляет команду;
- `int getHeight()` - получает высоту доступной области экрана в пикселях;
- `Ticker getTicker()` - получает бегущую строку;
- `String getTitle()` - получает заголовок;

- `int getWidth()` - получает ширину доступной области экрана в пикселях;
- `boolean isShown()` - проверяет видимость объекта на экране;
- `void removeCommand(Command cmd)` - удаляет команду;
- `void setCommandListener(CommandListener l)` - устанавливает обработчик событий;
- `void setTicker(Ticker ticker)` - устанавливает бегущую строку;
- `void setTitle(String s)` - устанавливает заголовок;
- `protected void sizeChanged(int w, int h)` - изменяет видимую область дисплея.

2.5.14. Класс *Font*

Класс шрифтов.

Методы

- `int charsWidth(char[] ch, int offset, int length)` - применяется для правильного планирования использования шрифта на экране дисплея;
- `int charWidth(char ch)` - получает ширину шрифта;
- `int getBaselinePosition()` - вычисляет расстояние от верхней кромки текста до опорной позиции в пикселях;
- `static Font getDefaultFont()` - получает системный шрифт, используемый устройством по умолчанию;
- `int getFace()` - получает начертание шрифта, используемого устройством по умолчанию;
- `static Font getFont(int fontSpecifier)` - используется классом `CustomItem` для получения специального шрифта;
- `static Font getFont(int face, int style, int size)` - получает шрифт с указанием начертания, стиля и размера;
- `int getHeight()` - получает высоту шрифта;
- `int getSize()` - получает размер шрифта;
- `int getStyle()` - получает стиль шрифта;
- `boolean isBold()` - возвращает значение `true`, если используется `Bold`;
- `boolean isItalic()` - возвращает значение `true`, если используется `Italic`;
- `boolean isPlain()` - возвращает значение `true`, если используется `Plain`;
- `boolean isUnderlined()` - возвращает значение `true`, если используется `Underlined`;
- `int stringWidth(String str)` - устанавливает строку текста;
- `int substringWidth(String str, int offset, int len)` - устанавливает подстроку текста.

Константы

- `static int FACE_MONOSPACE` - шрифте небольшим интервалом;
- `static int FACE_PROPORTIONAL` - пропорциональный шрифт;

- `static int FACE_SYSTEM` - системный шрифт;
- `static int FONT_INPUT_TEXT` - текст ввода;
- `static int FONT_STATIC_TEXT` - заданный по умолчанию шрифт;
- `static int SIZE_LARGE` - большой шрифт;
- `static int SIZE_MEDIUM` - средний шрифт;
- `static int SIZE_SMALL` - маленький шрифт;
- `static int STYLE_BOLD` - жирный шрифт;
- `static int STYLE_ITALIC` - курсив;
- `static int STYLE_PLAIN` - обычный шрифт;
- `static int STYLE_UNDERLINED` - подчеркнутый шрифт.

2.5.15. Класс *Form*

Этот класс создает пустую форму, в которую интегрируются классы пользовательского интерфейса.

Конструкторы

- `Form(String title)` - создает новую пустую форму;
- `Form(String title, Item[] items)` - создает новую форму с заданным заголовком и установленными компонентами класса `Item`.

Методы

- `int append(Image img)` - добавляет в форму одно изображение;
- `int append(Item item)` - этот метод добавляет любой из доступных компонентов класса `Item` в созданную форму;
- `int append(String str)` - добавляет в форму строку;
- `void delete(int itemNum)` - удаляет компонент, ссылающийся на `itemNum`;
- `void deleteAll()` - удаляет все компоненты с имеющейся формы;
- `Item get(int itemNum)` - получает позицию выбранного компонента;
- `int getHeight()` - возвращает высоту экрана в пикселях, доступную для встраиваемых компонентов;
- `int getWidth()` - возвращает ширину экрана в пикселях, доступную для встраиваемых компонентов;
- `void insert(int itemNum, Item item)` - вставляет компонент в форму до определенного компонента;
- `void set(int itemNum, Item item)` - устанавливает компонент, ссылающийся на компонент `itemNum`, заменяя при этом предшествующий компонент;
- `void setItemStateListener(ItemStateListener iListener)` - устанавливает переменную `iListener` для формы, заменяя при этом предыдущую переменную `iListener`;
- `int size()` - получает количество компонентов в форме.

2.5.16. Класс *Gauge*

Представляет графическое течение процесса, своего рода датчик или счетчик.

Конструктор

- `Gauge(String label, boolean interactive, int maxValue, int initialValue)` - создает графическое течение процесса с заданной меткой, режимом и максимальным и минимальным значениями в работе.

Методы

- `void addCommand(Command cmd)` - добавляет команду;
- `int getMaxValue()` - получает значение максимального диапазона работы процесса;
- `int getValue()` - получает текущее значение в процессе работы;
- `boolean isInteractive()` - определяет возможность изменения установленного счетчика;
- `void setDefaultCommand(Command cmd)` - задает команду по умолчанию для компонентов `Item`;
- `void setItemCommandListener(ItemCommandListener l)` - устанавливает обработчик событий;
- `void setLabel(String label)` - устанавливает метку для элемента;
- `void setLayout(int layout)` - устанавливает директивы для элемента;
- `void setMaxValue(int maxValue)` - устанавливает максимальное значение течения процесса;
- `void setPreferredSize(int width, int height)` - задает ширину и высоту для графического представления всего течения процесса;
- `void setValue(int value)` - устанавливает текущее значение процесса.

Константы

- `static int CONTINUOUS_IDLE` - непрерывное и неактивное состояние с неопределенным диапазоном работы;
- `static int CONTINUOUS_RUNNING` - непрерывное активное состояние с неопределенным диапазоном работы;
- `static int INCREMENTAL_IDLE` - увеличивающееся и неактивное состояние с неопределенным диапазоном работы;
- `static int INCREMENTAL_UPDATING` - увеличивающееся и постоянно модифицируемое состояние с неопределенным диапазоном работы;
- `static int INDEFINITE` - максимальное значение с неопределенным диапазоном работы.

2.5.1.7. Класс *Graphics*

Предоставляет возможность рисования графических элементов на экране мобильного устройства.

Методы

- `void clipRect(int x, int y, int width, int height)` - отсекает заданный прямоугольник;
- `void copyArea(int x_src, int y_src, int width, int height, int x_dest, int y_dest, int anchor)` - копирует прямоугольную

область из установленных значений в параметрах `x_src`, `y_src`, `width`, `height` в новую область `x_dest`, `y_dest`;

- `void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)` - рисует контур дуги в виде эллипса;
- `void drawChar(char character, int x, int y, int anchor)` - рисует символ;
- `void drawChars(char[] data, int offset, int length, int x, int y, int anchor)` - рисует массив символов;
- `void drawImage(Image img, int x, int y, int anchor)` - рисует изображение;
- `void drawLine(int x1, int y1, int x2, int y2)` - рисует линию из точки `x1` и `y1` до точки `x2` и `y2`;
- `void drawRegion(Image src, int x_src, int y_src, int width, int height, int transform, int x_dest, int y_dest, int anchor)` - копирует изображения в заданную область на экран телефона;
- `void drawRGB(int[] rgbData, int offset, int scanlength, int x, int y, int width, int height, boolean processAlpha)` - получает цвет в представлении **ARGB** и сохраняет в массиве данных;
- `void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)` - рисует контур прямоугольника, используя закругленные углы;
- `void drawString(String str, int x, int y, int anchor)` - рисует строку текста с заданным цветом и размером шрифта;
- `void drawSubstring(String str, int offset, int len, int x, int y, int anchor)` - рисует подстроку текста с заданным цветом и размером шрифта;
- `void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)` - рисует заполненную цветом дугу;
- `void fillRect(int x, int y, int width, int height)` - рисует заполненный цветом прямоугольник;
- `void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)` - рисует заполненный прямоугольник, используя закругленные углы;
- `void fillTriangle(int x1, int y1, int x2, int y2, int x3, int y3)` - рисует заполненный цветом треугольник;
- `int getBlueComponent()` - получает синий компонент цвета;
- `int getClipHeight()` - получает высоту для текущей области отсечения;
- `int getClipWidth()` - получает ширину для текущей области отсечения;
- `int getClipX()` - получает координату по оси **X** для текущей области отсечения;
- `int getClipY()` - получает координату по оси **Y** для текущей области отсечения;
- `int getColor()` - получает текущий цвет;

- `int getDisplayColor(int color)` - получает цвет для отображения;
- `Font getFont()` - получает текущий шрифт;
- `int getGrayScale()` - получает значение полутонов;
- `int getGreenComponent()` - получает зеленый компонент цвета;
- `int getRedComponent()` - получает красный компонент цвета;
- `int getStrokeStyle()` - получает используемый штриховой стиль;
- `int getTranslateX()` - получает координату по оси X для перемещенного контекста;
- `int getTranslateY()` - получает координату по оси Y для перемещенного контекста;
- `void setClip(int x, int y, int width, int height)` - устанавливает отсечение заданной области экрана;
- `void setColor(int RGB)` - устанавливает цвет при помощи значения RGB;
- `void setColor(int red, int green, int blue)` - назначает цвет при помощи трех цветовых компонентов: red, green и blue;
- `void setFont(Font font)` - устанавливает заданный шрифт;
- `void setGrayScale(int value)` - задает значение полутонов;
- `void setStrokeStyle(int style)` - задает штриховой стиль рисуемому контексту, используя константы SOLID и DOTTED;
- `void translate(int x, int y)` - перемещает систему координат в точку (x, y).

Константы

- `static int BASELINE` – опорная линия привязки шрифта;
- `static int BOTTOM` – нижняя линия привязки шрифта;
- `static int DOTTED` – пунктирный стиль;
- `static int HCENTER` – центральная линия привязки шрифта;
- `static int LEFT` – левая сторона привязки шрифта;
- `static int RIGHT` – правая сторона привязки шрифта;
- `static int SOLID` – штриховой стиль;
- `static int TOP` – верхняя линия привязки шрифта;
- `static int VCENTER` – выравнивает по центру изображение.

2.5.18. Класс Image

Отвечает за загрузку и отображение любых видов графических изображений формата PNG.

Методы

- `static Image createImage(byte[] imageData, int imageOffset, int imageLength)` - загружает изображение, учитывая смещение и длину в байтах;
- `static Image createImage(Image source)` - загружает изображение из файла;
- `static Image createImage(Image image, int x, int y, int width,`

- `int height, int transform)` - загружает изображение в заданное место, определенное координатами, с возможностью трансформации изображения;
- `static Image createImage(InputStream stream)` - загружает изображение из потока;
- `static Image createImage(int width, int height)` - загружает изображение в заданные размеры;
- `static Image createImage(String name)` - загружает изображение из ресурса;
- `static Image createRGBImage(int[] rgb, int width, int height, boolean processAlpha)` - загружает изображение, учитывая цветовую компоненту ARGB;
- `Graphics getGraphics()` - создает графический объект для представления изображения;
- `int getHeight()` - получает высоту изображения;
- `void getRGB(int[] rgbData, int offset, int scanlength, int x, int y, int width, int height)` - получает цвет в представлении ARGB и сохраняет в массиве данных;
- `int getWidth()` - получает ширину изображения;
- `boolean isMutable()` - проверяет изображение.

2.5.19. Класс *ImageItem*

Контейнер для загружаемых в приложение сложных графических изображений.

Конструкторы

- `ImageItem(String label, Image img, int layout, String altText)` - создает объект класса `ImageItem` с заданной меткой, изображением, размещением и дополнительной строкой текста;
- `ImageItem(String label, Image image, int layout, String altText, int appearanceMode)` - создает объект класса `ImageItem` с заданной меткой, изображением, размещением, дополнительной строкой текста и режимом представления.

Методы

- `String getAltText()` - размещает текст;
- `int getAppearanceMode()` - возвращает режим представления;
- `Image getImage()` - получает изображение;
- `int getLayout()` - получает директивы для размещения изображений;
- `void setAltText(String text)` - устанавливает дополнительный текст;
- `void setImage(Image img)` - устанавливает изображение;
- `void setLayout(int layout)` - устанавливает директивы для размещения изображений.

2.5.20. Класс *Item*

Суперкласс, содержащий ряд классов для их дальнейшей интеграции в класс `Form`.

Методы

- `void addCommand(Command cmd)` - добавляет команду;
- `String getLabel()` - получает метку объекта `Item`;
- `int getLayout()` - использует директивы для размещения компонентов в форме;
- `int getMinimumHeight()` - получает минимальную высоту;
- `int getMinimumWidth()` - получает минимальную ширину;
- `int getPreferredHeight()` - получает предпочтительную высоту;
- `int getPreferredWidth()` - получает предпочтительную ширину;
- `void notifyStateChanged()` - компонент, содержащийся в форме. Уведомляет объект `ItemStateListener` о своем состоянии;
- `void removeCommand(Command cmd)` - удаляет команду из компонента.
- `void setDefaultCommand(Command cmd)` - встроенная команда по умолчанию для данного компонента;
- `void setItemCommandListener(ItemCommandListener l)` - устанавливает обработку событий для компонента;
- `void setLabel(String label)` - устанавливает назначенную метку для компонента;
- `void setLayout(int layout)` - устанавливает рассмотренные выше директивы для форматирования компонента;
- `void setPreferredSize(int width, int height)` - устанавливает оптимальную высоту и ширину компонента.

Константы

- `static int BUTTON` - отображение элемента в виде кнопки;
- `static int HYPERLINK` - отображение элемента в виде гиперссылки;
- `static int LAYOUT_2` - режим установки;
- `static int LAYOUT_BOTTOM` - выравнивание по нижней стороне экрана;
- `static int LAYOUT_CENTER` - выравнивание по центру экрана;
- `static int LAYOUT_DEFAULT` - значение по умолчанию;
- `static int LAYOUT_EXPAND` - увеличение ширины компонента;
- `static int LAYOUT_LEFT` - выравнивание по левой стороне экрана;
- `static int LAYOUT_NEWLINE_AFTER` - размещение на новой строке;
- `static int LAYOUT_NEWLINE_BEFORE` - размещение в начале строки;
- `static int LAYOUT_RIGHT` - выравнивание по правой стороне экрана;
- `static int LAYOUT_SHRINK` - уменьшение компонента по ширине;
- `static int LAYOUT_TOP` - выравнивание по верхней стороне экрана;
- `static int LAYOUT_VCENTER` - выравнивание по центру экрана;
- `static int LAYOUT_VEXPAND` - увеличение высоты для размещения компонента;
- `static int LAYOUT_VSHRINK` - уменьшение высоты для размещения компонента;
- `static int PLAIN` - установка режима появления первого плана для компонента.

2.5.21. Класс *List*

Создает список группы элементов.

Конструкторы

- `List(String title, int listType)` - формирует список с названием и типом созданного списка;
- `List(String title, int listType, String[] stringElements, Image[] imageElements)` - формирует список с названием, типом созданного списка, массивом строк и изображений для списка элементов.

Методы

- `int append(String stringPart, Image imagePart)` - добавление списка элементов;
- `void delete(int elementNum)` - удаление заданного элемента из списка;
- `void deleteAll()` - удаление всех элементов из списка;
- `int getFitPolicy()` - получает привилегированную позицию;
- `Font getFont(int elementNum)` - получает шрифт для заданного элемента в списке;
- `Image getImage(int elementNum)` - получает изображение для заданного элемента в списке;
- `int getSelectedFlags(boolean[] selectedArray_return)` - возвращает состояние всех элементов в виде массива данных;
- `int getSelectedIndex()` - получает выбранный индекс элемента в списке;
- `String getString(int elementNum)` - получает строку текста для выбранного элемента из списка;
- `void insert(int elementNum, String stringPart, Image imagePart)` - вставляет элемент в список до указанного номера элемента в списке;
- `boolean isSelected(int elementNum)` - получает выбранный элемент из списка;
- `void removeCommand(Command cmd)` - удаляет команду для списка;
- `void set(int elementNum, String stringPart, Image imagePart)` - вставляет новый элемент в список взамен предшествующего;
- `void setFitPolicy(int fitPolicy)` - устанавливает привилегированную позицию;
- `void setFont(int elementNum, Font font)` - устанавливает шрифт заданному элементу в списке;
- `void setSelectCommand(Command command)` - этот метод предназначен для работы с типом `IMPLICIT`;
- `void setSelectedFlags(boolean[] selectedArray)` - устанавливает состояние выбранных элементов;
- `void setSelectedIndex(int elementNum, boolean selected)` - устанавливает индекс выбранного элемента в списке;
- `void setTicker(Ticker ticker)` - устанавливает бегущую строку;
- `void setTitle(String s)` - добавляет название;

- `int size()` - с помощью этого метода можно узнать количество элементов в списке.

Константа

- `static Command SELECT_COMMAND` - команда по умолчанию для типа `IMPLICIT`.

2.5.22. Класс *Screen*

Суперкласс для всех высокоуровневых классов, определяющих пользовательский интерфейс приложения.

2.5.23. Класс *Spacer*

Создает заданное пространство на экране.

Конструктор

- `Spacer(int minWidth, int minHeight)` - создает пространство на экране с заданной шириной и высотой.

Методы

- `void addCommand(Command cmd)` - добавляет команду;
- `void setDefaultCommand(Command cmd)` - устанавливает команду по умолчанию;
- `void setLabel(String label)` - устанавливает метку;
- `void setMinimumSize(int minWidth, int minHeight)` - устанавливает минимальный размер для создаваемого пространства.

2.5.24. Класс *StringItem*

Формирует текстовые строки.

Конструкторы

- `StringItem(String label, String text)` - создает строку текста с заданной меткой;
- `StringItem(String label, String text, int appearanceMode)` - создает строку текста с заданной меткой и режимом отображения.

Методы

- `int getAppearanceMode()` - возвращает заданный способ отображения текста на экране;
- `Font getFont()` - получает шрифт текста;
- `String getText()` - получает текст для класса `StringItem`;
- `void setFont(Font font)` - устанавливает шрифт текста;
- `void setPreferredSize(int width, int height)` - задает ширину и высоту текста;
- `void setText(String text)` - устанавливает текст для класса `StringItem`.

2.5.25. Класс *TextBox*

Организовывает редактируемый текстовый контейнер.

Конструктор

- `TextBox(String title, String text, int maxSize, int constraints)` - создает текстовый контейнер с заданным заголовком, строкой текста, максимальным размером символов и ограничением.

Методы

- `void delete(int offset, int length)` - удаляет все символы из созданного контейнера;
- `int getCaretPosition()` - получает текущую позицию нахождения указателя на экране;
- `int getChars(char[] data)` - копирует содержимое контейнера `TextBox` - в массив данных;
- `int getConstraints()` - получает текущие ограничения для контейнера;
- `int getMaxSize()` - возвращает максимальное число символов, установленное для контейнера `TextBox`;
- `String getString()` - получает строку текста из содержимого контейнера `TextBox`;
- `void insert(char[] data, int offset, int length, int position)` - вставляет массив символов в `TextBox`;
- `void insert(String src, int position)` - вставляет строку текста в `TextBox`;
- `void setChars(char[] data, int offset, int length)` - прописывает в `TextBox` массив символов;
- `void setConstraints(int constraints)` - устанавливает ограничения;
- `void setInitialInputMode(String characterSubset)` - задает напоминание;
- `int setMaxSize(int maxSize)` - устанавливает максимальный размер для `TextBox`;
- `void setString(String text)` - прописывает в `TextBox` строку текста;
- `void setTicker(Ticker ticker)` - устанавливает бегущую строку;
- `void setTitle(String s)` - устанавливает заголовок;
- `int size()` - определяет размер `TextBox`.

2.5.26. Класс *TextField*

Создает редактируемый текстовый контейнер, который встраивается в класс `Form`.

Конструктор

- `TextField(String label, String text, int maxSize, int constraints)` - создает текстовый контейнер с заданным заголовком, строкой текста, максимальным размером символов и ограничением, с последующей интеграцией в класс `Form`.

Методы

- `void delete(int offset, int length)` - удаляет все символы из созданного контейнера;
- `int getCaretPosition()` - получает текущую позицию нахождения указателя на экране;
- `int getChars(char[] data)` - копирует содержимое контейнера `TextField` в массив данных;
- `int getConstraints()` - получает текущие ограничения для контейнера;
- `int getMaxSize()` - возвращает максимальное число символов, установленное для контейнера `TextField`;
- `String getString()` - получает строку текста из содержимого контейнера `TextField`;
- `void insert(char[] data, int offset, int length, int position)` - вставляет массив символов в `TextField`;
- `void insert(String src, int position)` - вставляет строку текста в `TextField`;
- `void setChars(char[] data, int offset, int length)` - прописывает в `TextField` массив символов;
- `void setConstraints(int constraints)` - устанавливает ограничения;
- `void setInitialInputMode(String characterSubset)` - задает напоминание;
- `int setMaxSize(int maxSize)` - устанавливает максимальный размер для `TextField`;
- `void setString(String text)` - прописывает в `TextField` строку текста;
- `int size()` - определяет размер `TextField`.

Константы

- `static int ANY` - определяет ввод любого текста;
- `static int CONSTRAINT_MASK` - режим ограничения для маски;
- `static int DECIMAL` - ввод дробных числовых значений;
- `static int EMAILADDR` - используется при вводе электронного адреса;
- `static int INITIAL_CAPS_SENTENCE` - начальный символ каждого предложения будет печататься с заглавной буквы;
- `static int INITIAL_CAPS_WORD` - начальный символ каждого слова будет печататься с заглавной буквы;
- `static int NON_PREDICTIVE` - значение слов, не используемое в словаре, найдено не будет;
- `static int NUMERIC` - для ввода только целочисленных значений;
- `static int PASSWORD` - для ввода пароля;
- `static int PHONENUMBER` - для ввода телефонного номера;
- `static int UNEDITABLE` - редактирование не доступно;
- `static int URL` - для ввода адреса сайта.

2.5.27. Класс *Ticker*

Создает на экране бегущую строку текста.

Конструктор

- `Ticker(String str)` - формирует бегущую строку текста.

Методы

- `String getString()` - получает строку текста, заданную для объекта класса `Ticker`;
- `void setString(String str)` - устанавливает строку текста для отображения ее на экране с помощью объекта класса `Ticker`, заменяя ее новой строкой.

2.6. Пакет javax.microedition.lcdui.game

Игровой пакет, благодаря которому можно достаточно легко создавать игры для мобильных устройств.

2.6.1. Класс *GameCanvas*

Абстрактный класс, содержащий основные элементы игрового интерфейса.

Конструктор

- `protected GameCanvas(boolean suppressKeyEvents)` - конструктор абстрактного класса `GameCanvas`.

Методы

- `void flushGraphics()` - копирует изображение из внеэкранного буфера на экран;
- `void flushGraphics(int x, int y, int width, int height)` - копирует изображение из внеэкранного буфера на экран в заданный по размеру прямоугольник;
- `protected Graphics getGraphics()` - получает графические элементы для представления их впоследствии классом `GameCanvas`;
- `int getKeyStates()` - определяет, какая из клавиш нажата;
- `void paint(Graphics g)` - рисует графические элементы, представленные классом `GameCanvas`.

Константы

- `static int DOWN_PRESSED` - движение вниз;
- `static int FIRE_PRESSED` - реализует стрельбу из оружия;
- `static int GAME_A_PRESSED` - игровая клавиша A;
- `static int GAME_B_PRESSED` - игровая клавиша B;
- `static int GAME_C_PRESSED` - игровая клавиша C;
- `static int GAME_D_PRESSED` - игровая клавиша D;
- `static int LEFT_PRESSED` - движение влево;
- `static int RIGHT_PRESSED` - движение вправо;
- `static int UP_PRESSED` - движение вверх.

2.6.2. Класс *Layer*

Абстрактный класс, отвечающий за уровни, представляемые в игре.

Методы

- `int getHeight()` - получает высоту экрана;
- `int getWidth()` - получает ширину экрана;
- `int getX()` - получает горизонтальную координату уровня;
- `int getY()` - получает вертикальную координату уровня;
- `boolean isVisible()` - получает видимость данного уровня;
- `void move(int dx, int dy)` - перемещает уровень на координаты `dx` и `dy`;
- `abstract void paint(Graphics g)` - рисует уровень;
- `void setPosition(int x, int y)` - устанавливает уровень в позицию, обозначенную в координатах `x` и `y`;
- `void setVisible(boolean visible)` - устанавливает видимость данного уровня.

2.6.3. Класс *LayerManager*

Менеджер имеющихся в игре уровней.

Конструктор

- `LayerManager()` - создает менеджер уровней.

Методы

- `void append(Layer l)` - добавляет уровень в менеджер уровней;
- `Layer getLayerAt(int index)` - получает уровень с заданным индексом;
- `int getSize()` - получает общее количество уровней;
- `void insert(Layer l, int index)` - подключает новый уровень в заданный индекс;
- `void paint(Graphics g, int x, int y)` - представляет текущий менеджер уровней в заданных координатах;
- `void remove(Layer l)` - удаляет уровень из менеджера уровней;
- `void setViewWindow(int x, int y, int width, int height)` - устанавливает область на экране для отображения уровня.

2.6.4. Класс *Sprite*

Создает спрайт, представляющий изображение или анимационные фреймы.

Конструкторы

- `Sprite(Image image)` - создает неанимированный спрайт;
- `Sprite(Image image, int frameWidth, int frameHeight)` - создает спрайт, представленный анимационными фреймами;
- `Sprite(Sprite s)` - создает спрайт из другого спрайта.

Методы

- `boolean collidesWith(Sprite s, boolean pixelLevel)` - определяет столкновение между спрайтами;

- `boolean collidesWith(TiledLayer t, boolean pixelLevel)` - определяет столкновение между спрайтом и препятствием, нарисованным при помощи класса `TiledLayer`;
- `public void defineReferencePixel(int x, int y)` - изменяет опорную позицию спрайта, перенося ее в точку с координатами `x` и `y`;
- `int getFrame()` - получает текущий фрейм;
- `int getFrameSequenceLength()` - получает количество элементов в анимационных фреймах;
- `int getRawFrameCount()` - получает количество неиспользованных фреймов;
- `int getRefPixelX()` - получает координату по оси `X` для спрайта;
- `int getRefPixelY()` - получает координату по оси `Y` для спрайта;
- `void nextFrame()` - осуществляет переход на следующий фрейм;
- `void paint(Graphics g)` - рисует спрайт;
- `void prevFrame()` - осуществляет переход на предыдущий фрейм;
- `void setFrame(int sequenceIndex)` - устанавливает заданный фрейм;
- `void setFrameSequence(int[] sequence)` - устанавливает определенную фреймовую последовательность;
- `void setImage(Image img, int frameWidth, int frameHeight)` - изменяет изображение спрайта на новое изображение;
- `void setRefPixelPosition(int x, int y)` - устанавливает координаты по осям `X` и `Y` для спрайта;
- `void setTransform(int transform)` - производит трансформацию спрайта.

Константы

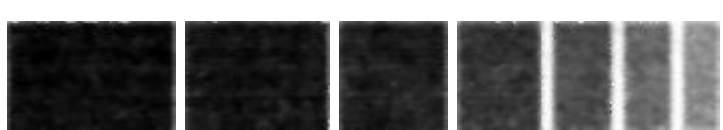
- `static int TRANS_MIRROR` - отраженный по вертикали;
- `static int TRANS_MIRROR_ROT180` - отраженный по вертикали со сдвигом на 180° по часовой стрелке;
- `static int TRANS_MIRROR_ROT270` - отраженный по вертикали со сдвигом на 270° по часовой стрелке;
- `static int TRANS_MIRROR_ROT90` - отраженный по вертикали со сдвигом на 90° по часовой стрелке;
- `static int TRANS_NONE` - без трансформации;
- `static int TRANS_ROT180` - сдвинут по часовой стрелке на 180° ;
- `static int TRANS_ROT270` - сдвинут по часовой стрелке на 270° ;
- `static int TRANS_ROT90` - сдвинут по часовой стрелке на 90° .

2.6.5. Класс *TiledLayer*

Осуществляет создание фоновых изображений.

Конструктор

- `TiledLayer(int columns, int rows, Image image, int tileWidth, int tileHeight)` - создает фоновое изображение с заданным количеством



столбцов, строк, исходным изображением и шириной и высотой одной ячейки рисунка.

Методы

- `int createAnimatedTile(int staticTileIndex)` - создает анимационный фон и возвращает следующий индекс ячейки;
- `void fillCells(int col, int row, int numCols, int numRows, int tileIndex)` - заполняет ячейки;
- `int getAnimatedTile(int animatedTileIndex)` - получает индекс анимационной последовательности;
- `int getCell(int col, int row)` - получает ячейку;
- `int getCellHeight()` - получает высоту ячейки в пикселях;
- `int getCellWidth()` - получает ширину ячейки в пикселях;
- `int getColumns()` - получает количество колонок, на которое разбито изображение фона;
- `int getRows()` - получает количество строк, на которое разбито изображение фона;
- `void paint(Graphics g)` - рисует фон;
- `void setAnimatedTile(int animatedTileIndex, int staticTileIndex)` - устанавливает анимационную последовательность;
- `void setCell(int col, int row, int tileIndex)` - рисует заданную ячейку;
- `void setStaticTileSet(Image image, int tileWidth, int tileHeight)` - заменяет набор ячеек.

2.7. Пакет `javax.microedition.media`

Пакет добавлен в профиль MIDP 2.0 и дает возможность создания звукового сопровождения в приложении.

2.7.1. Интерфейс *Control*

Осуществляет контроль над процессами.

2.7.2. Интерфейс *Controllable*

Осуществляет контроль над объектами.

Методы

- `Control getControl(String controlType)` - получает объект, осуществляющий управление;
- `Control[] getControls()` - получает совокупность объектов, осуществляющих управление.

2.7.3. Интерфейс *Player*

Реализует контроль над воспроизведением.

Методы

- `void addPlayerListener(PlayerListener PlayerListener)` - осуществляет обработку событий от определенного проигрывателя;
- `void close()` - закрывает проигрыватель;
- `void deallocate()` - освобождает ресурс, занятый проигрывателем;
- `String getContentType()` - получает тип аудиоданных, воспроизводимых проигрывателем;
- `long getDuration()` - получает размер аудиофайла;
- `long getMediaTime()` - получает время воспроизведения аудиоданных;
- `int getState()` - определяет состояние проигрывателя;
- `void prefetch()` - захватывает ресурсы для последующего воспроизведения данных;
- `void realize()` - создает проигрыватель без захвата ресурсов;
- `void removePlayerListener(PlayerListener PlayerListener)` - удаляет установленный обработчик событий;
- `void setLoopCount(int count)` - устанавливает цикличное воспроизведение аудиоданных;
- `long setMediaTime(long now)` - устанавливает время воспроизведения;
- `void start()` - дает команду на воспроизведение;
- `void stop()` - останавливает воспроизведение.

Константы

- `static int CLOSED` - закрывает проигрыватель;
- `static int PREFETCHED` - захватывает ресурсы для воспроизведения;
- `static int REALIZED` - приобретает информацию для воспроизведения;
- `static int STARTED` - воспроизведение запущенно;
- `static long TIME_UNKNOWN` - неизвестное время установки;
- `static int UNREALIZED` - не произошло захвата ресурсов и информации для воспроизведения.

2.7.4. Интерфейс *PlayerListener*

Получает асинхронные события проигрывателя.

Методы

- `void playerUpdate(Player player, String event, Object eventData)` - обновляет состояние проигрывателя.

Константы

- `static String CLOSED` - уведомляет о закрытии проигрывателя;
- `static String DEVICE_AVAILABLE` - уведомляет о доступности проигрывателя;
- `static String DEVICE_UNAVAILABLE` - уведомляет о недоступности проигрывателя;

- `static String DURATION_UPDATED` - обновляет состояние;
- `static String END_OF_MEDIA` - уведомляет о конце воспроизведения данных проигрывателем;
- `static String ERROR` - уведомляет об ошибке;
- `static String STARTED` - уведомляет о начале работы проигрывателя;
- `static String STOPPED` - уведомляет о конце работы проигрывателя;
- `static String VOLUME_CHANGED` - уведомляет о выборе громкости для воспроизведения.

2.7.5. Класс *Manager*

Менеджер системных ресурсов.

Методы

- `static Player createPlayer(InputStream stream, String type)` - создает проигрыватель для воспроизведения аудиоданных из потока;
- `static Player createPlayer(String locator)` - создает проигрыватель для воспроизведения аудиоданных из определенного файла;
- `static String[] getSupportedContentTypes(String protocol)` - возвращает список доступных контекстных типов для протоколов;
- `static String[] getSupportedProtocols(String content_type)` - возвращает список доступных протоколов для контекстных типов;
- `static void playTone(int note, int duration, int volume)` - воспроизводит различные тональные звуки.

Константа

- `static String TONE_DEVICE_LOCATOR` - необходимая для последовательного воспроизведения тонов устройства.

2.7.6. Исключения

- `MediaException` - исключает ошибки в работе методов этого пакета.

2.8. Пакет `javax.microedition.media.control`

Осуществляет контроль над процессами.

2.8.1. Интерфейс *ToneControl*

Производит воспроизведение тональных звуков на устройстве.

Метод

- `void setSequence(byte[] sequence)` - устанавливает тональные звуки.

Константы

- `static byte BLOCK_END` - конец блока воспроизведения;
- `static byte BLOCK_START` - стартовая позиция в блоке;

- static byte C4 - нота До;
- static byte PLAY_BLOCK - воспроизвести блок;
- static byte REPEAT - повторить воспроизведение блока;
- static byte RESOLUTION - событие;
- static byte SET_VOLUME - установить громкость;
- static byte SILENCE - без звука;
- static byte TEMPO - темп или скорость воспроизведения;
- static byte VERSION - версия атрибута воспроизведения.

2.8.2. Интерфейс *VolumeControl*

Регулирует громкость воспроизведения.

Методы

- int getLevel() - возвращает текущий уровень громкости;
- boolean isMuted() - определяет состояние сигнала;
- int setLevel(int level) - устанавливает уровень громкости. Значение может находиться в пределах от 0 до 100;
- void setMute(boolean mute) - устанавливает состояние сигнала.

2.9. Пакет javax.microedition.midlet

С помощью этого пакета происходит связь между приложением и мобильным информационным профилем устройства (MIDP).

2.9.1. Класс *MIDlet*

Основной класс мидлета должен наследовать класс MIDlet для управления работой приложения.

Конструктор

- protected MIDlet() - закрытый конструктор.

Методы

- int checkPermission(String permission) - получить статус;
- protected abstract void destroyApp(boolean unconditional) - заканчивает работу программы;
- String getAppProperty(String key) - получает свойства программного обеспечения;
- void notifyDestroyed() - уведомляет программное обеспечение о конце работы;
- void notifyPaused() - уведомляет программное обеспечение о паузе в работе;
- protected abstract void pauseApp() - переходит в состояние паузы;
- boolean platformRequest(String URL) - дескриптор устройства получает URL;
- void resumeRequest() - переход в активное состояние;

- `protected abstract void startApp()` - входная точка программы, осуществляет старт приложения.

2.9.2. Исключение

- `MIDletStateChangeException` - исключает неправильную работу с классом `MIDlet`.

2.10. Пакет `javax.microedition.pki`

Сертифицирует информацию для безопасной связи.

2.10.1. Интерфейс *Certificate*

Общий сертификат связи.

Методы

- `String getIssuer()` - получает используемый сертификат;
- `long getNotAfter()` - получает момент времени, после которого сертификат использовать нельзя;
- `long getNotBefore()` - получает момент времени, до которого сертификат использовать нельзя;
- `String getSerialNumber()` - получает серийный номер сертификата;
- `String getSigAlgName()` - получает имя используемого алгоритма записи данного сертификата;
- `String getSubject()` - получает название субъекта сертификата;
- `String getType()` - получает тип сертификата;
- `String getVersion()` - получает версию сертификата.

2.10.2. Исключение

- `CertificateException` - обобщенный вид ошибок, возникший при использовании данного сертификата.

2.11. Пакет `javax.microedition.rms`

Осуществляет хранение, удаление, добавление записей в системную память устройства.

2.11.1. Интерфейс *RecordComparator*

Осуществляет сортировку записей.

Метод

- `int compare(byte[] rec1, byte[] rec2)` - сортирует записи.

Константы

- `static int EQUIVALENT` - две записи одинаковы;

- `static int FOLLOWS` - первая запись больше второй записи;
- `static int PRECEDES` - вторая запись больше, чем первая.

2.11.2. Интерфейс *RecordEnumeration*

Реализует двунаправленный список записи.

Методы

- `void destroy()` - освобождает захваченные ресурсы;
- `boolean hasNextElement()` - возвращает значение `true`, если имеются последующие записи;
- `boolean hasPreviousElement()` - возвращает значение `true`, если имеются предшествующие записи;
- `boolean isKeptUpdated()` - возвращает значение `true` в том случае, если сохраняются изменения в записи;
- `void keepUpdated(boolean keepUpdated)` - устанавливает, возможно ли сохранение индексов записей при изменении, удалении или добавлении записей;
- `byte[] nextRecord()` - возвращает копию следующей записи в списке;
- `int nextRecordId()` - возвращает идентификатор следующей записи в списке;
- `int numRecords()` - возвращает число доступных записей;
- `byte[] previousRecord()` - возвращает копию предыдущей записи в списке;
- `int previousRecordId()` - возвращает идентификатор предыдущей записи в списке;
- `void rebuild()` - делает запрос для обновления списка доступных записей;
- `void reset()` - сбрасывает индекс записи к первоначальному значению.

2.11.3. Интерфейс *RecordFilter*

Определяет совпадения записей.

Метод

- `boolean matches(byte[] candidate)` - возвращает значение `true`, если кандидат соответствует заданному критерию.

2.11.4. Интерфейс *RecordListener*

Производит обработку событий, связанных с изменением, добавлением и удалением записей.

Методы

- `void recordAdded(RecordStore recordStore, int recordId)` - вызывается после добавления записи;
- `void recordChanged(RecordStore recordStore, int recordId)` - вызван после изменения записи;

- `void recordDeleted(RecordStore recordStore, int recordId)` - вызван после удаления записи.

2.11.5. Класс *RecordStore*

Производит запись данных.

Методы

- `int addRecord(byte[] data, int offset, int numBytes)` - добавляет новую запись в память мобильного устройства;
- `void addRecordListener(RecordListener listener)` - добавляет обработчик событий;
- `void closeRecordStore()` - закрывает запись;
- `void deleteRecord(int recordId)` - удаляет запись по идентификатору;
- `static void deleteRecordStore(String recordStoreName)` - удаляет запись по имени;
- `long getLastModified()` - возвращает последнее время изменения записи;
- `String getName()` - получает имя записи;
- `int getNextRecordID()` - получает идентификатор последующей записи;
- `int getNumRecords()` - получает количество доступных записей;
- `byte[] getRecord(int recordId)` - возвращает копию записи;
- `int getRecord(int recordId, byte[] buffer, int offset)` - возвращает данные записи;
- `int getRecordSize(int recordId)` - получает размер заданной записи;
- `int getSize()` - получает размер всех записей;
- `int getSizeAvailable()` - получает количество доступной памяти для записи;
- `int getVersion()` - получает версию записи;
- `static String[] listRecordStores()` - возвращает список записей;
- `static RecordStore openRecordStore(String recordStoreName, boolean createIfNecessary)` - открывает память для записи;
- `static RecordStore openRecordStore(String recordStoreName, boolean createIfNecessary, int authmode, boolean writable)` - открывает память для записи;
- `static RecordStore openRecordStore(String recordStoreName, String vendorName, String suiteName)` - открывает память для записи;
- `void removeRecordListener(RecordListener listener)` - удаляет обработчик событий;
- `void setMode(int authmode, boolean writable)` - изменяет режим доступа;
- `void setRecord(int recordId, byte[] newData, int offset, int numBytes)` - вводит данные в запись.

Константы

- `static int AUTHMODE_ANY` - позволяет осуществить доступ для любого комплекта мидлетов;
- `static int AUTHMODE_PRIVATE` - позволяет осуществить доступ только из данной программы.

<http://palata-x.narod.ru>

Алфавитный указатель

А

Анимационная
последовательность 222
Анимация 235

Б

Библиотека импорта 111

В

Виртуальная Java-
машина 46
Внеэкранный буфер 218
Воспроизведение
wav-файл 427
пауза 430
скорость 430
тональные звуки 429
Встраивание группы
элементов 134
Вывод текста 197

Д

Двойная буферизация 218
Детерминированный
алгоритм 366

З

Загрузка
изображение 174
изображение в форму 154
Задание шрифта 177

И

Игровой цикл 252, 282
простой 282
сложный 284
Изображение
анимированное 323
неанимированное 323
Инсталляция
J2ME Wireless Toolkit 68

Motorola Lanchpad
forJ2ME 91
Sony Ericsson J2ME
SDK 2.1 100
Инструментальные
средства 15
Интегрирование в форму
строки текста 147
Интерпретатор 16
Интерфейс 43
Control 424
Controllable 424
Player 424
PlayerListener 425
Runnable 199, 218
ToneControl 426
VolumeControl 427
высокоуровневый 117
низкоуровневый 118
Искусственный
интеллект 353

К

Карта 303
Класс 17
Alert 119, 161
Background 359
Canvas 183
ChoiceGroup 134
Command 113
DateField 141
DemoGraphics 199
Display 192
Explosion 405
Font 177
Form 130
GameCanvas 218
GameMidlet 258, 355
Gauge 158
Graphics 185
Image 174

ImageItem 154
InputStream 427
Item 133
Layer 219
LayerManager 221
List 165
Loading 262, 267, 358
MainClassFont 180
MainGameCanvas 271
Manager 426
Menu 290
Random 371
Ship 391
Shot 391
Spacer 152
Splash 261, 263, 357
Sprite 222, 324
StringItem 147
TextBox 114
TextField 144
Ticker 171
TiledLayer 220
ToneControl 430
Ключевое слово
class 19
extends 44
implements 43
interface 43
new 28, 37
package 44
private 26
protected 26
public 26
return 29
super 41
this 43
Ключевые коды 183
Комментарии 20
Компилятор 16
Компиляция J2ME
Wireless Toolkit 73

- java.io 55
- Java.lang 53
- java.util 54
- javax.microedition.io 56
- javax.microedition.lcdui 56

javax.microedition.lcdui.game

58

javax.microedition.media

59

javax.miaoedition.media.control

60

javax.microedition.midlet

60

javax.microedition.pki

60

javax.microedition.rms

61

Перемещение

с помощью клавиш

211

Программирование

15

объектно-

ориентированное

16

Профиль

MIDP 1.0

51

MIDP 2.0

52

Р

Размер и позиция уровня

219

Рисование

дуга

194

линия

187

окружность

194

прямоугольник

191

С

Секвенсор

430

Система координат

185

Скобки

круглые

19

фигурные

19

Скорость перемещения

329

Слой

305

Создание

бегущая строка

171

графический измеритель

процессов

158

карта

306

контейнер для

редактируемого текста

144

поток

281

проигрыватель

427

свободное

пространство

152

список элементов

165

форма

131

цикл

199

Создание проекта

J2ME Wireless Toolkit

71

Спецификатор import

44

Столкниование

двух объектов

240

с препятствием

207

Т

Типы данных

20

У

Уведомление о внештатных

ситуациях

161

Уровень

221

Установка

дата и время

141

компоненты

интерфейса

133

Ф

Файл

JAD

76

JAR

76

манифеста

75

Фон игровой сцены

220

Фоновое изображение

224

Фоновый рисунок

303

Фрейм

222

Ц

Цикл

33

do/while

36

for

36

while

34

Цикличное перемещение

204

Ш

Шаблон

373

Я

Язык программирования

15

N

NetBeans IDE

интеграция

101

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@aliants-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.aliants-kniga.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**; электронный адрес **books@aliants-kniga.ru**.

<http://palata-x.narod.ru>

Горнаков Станислав Геннадьевич

Программирование мобильных телефонов на Java 2 Micro Edition

Главный редактор *Мовчан Д. Л.*
dm@dmk-press.ru

Верстка *Старцевой Е. М.*

Корректор *Синяева Г. И.*

Дизайн обложки *Мовчан А. Г.*

Электронный адрес издательства: www.dmk-press.ru

Подписано в печать 07.12.2007. Формат 70x100 ¹/₁₆.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 31,5. Тираж 2000 экз.

Заказ № 3212.

Отпечатано с готового оригинал-макета
на ОГУП «Областная типография «Печатный двор»
432049, г. Ульяновск, ул. Пушкарева, 27.

Программирование мобильных телефонов на JAVA 2 ME

Вы держите в руках 2-е и переработанное издание одной из популярных книг по программированию мобильных телефонов на Java 2 ME. Автор написал уникальное издание, обучившее огромное количество начинающих программистов создавать свои приложения для мобильных телефонов.

Новая версия издания содержит девять дополнительных глав. Теперь читатель кроме программирования приложений для платформы Java 2 ME изучит полный процесс создания мобильной игры. Во время работы с книгой будет освоен подход в формировании полноценного мобильного игрового движка, изучена работа с графикой, анимацией, даны примеры многослойных и анимированных игровых карт. Рассматриваются основы реализации искусственного интеллекта, игровые столкновения, создание интерактивного меню игры, игровых экранов с загрузкой и заставкой компонентов приложения. Итогом книги станет изучение всех основных компонентов платформы Java 2 ME и создание полноценной мобильной игры.

Компакт-диск содержит бесплатные инструментарии от компании Sun Microsystems и большой набор телефонных эмуляторов от компаний Nokia, BenQ-Siemens, Sony Ericsson, Motorola и Samsung.

Горнаков Станислав Геннадьевич – профессиональный программист в области создания мобильных и компьютерных игр. Автор многочисленных книг по программированию мобильных телефонов, смартфонов, КПК на основе Windows Mobile и Symbian OS, а также программированию игр и игровых приставок.



Посетите Интернет-страницу автора книги по адресу: www.gornakov.ru

Internet-магазин:
www.aliants-kniga.ru

Книга – почтой:
Россия, 123242, Москва, а/я 20
e-mail: books@aliants-kniga.ru

Оптовая продажа:
«Альянс-книга»
Тел./факс: (495) 258-9195
e-mail: books@aliants-kniga.ru



ISBN 5-94074-409-5



9 785940 744092