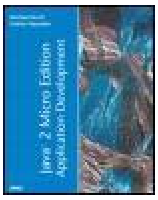


Java™ 2 Micro Edition Application Development

By Michael Kroll, Stefan Haustein

Publisher : Sams Publishing
Pub Date : June 25, 2002
ISBN : 0-672-32095-9
Pages : 504

The key to *Java 2 Micro Edition (J2ME) Application Development* is the clear, concise explanations of the J2ME technology in relation to the existing Java platform. This book assumes proficiency with Java and presents strategies for understanding and deploying J2ME applications. The book presents numerous real-world examples, including health care and financial sector examples from the authors' professional experience.

Table of Content

Copyright

Copyright © 2002 by Sams Publishing

Library of Congress Catalog Card Number: 2001086073

Printed in the United States of America

First Printing: June 2002

05 04 03 02 4 3 2 1

Trademarks

Warning and Disclaimer

Credits

Executive Editor

Michael Stephens

Acquisitions Editor

Carol Ackerman

Development Editor

Tiffany Taylor

Managing Editor

Charlotte Clapp

Project Editor

Matt Purcell

Production Editor

Rhonda Tinch-Mize

Indexer

Ginny Bess

Proofreaders

Chip Gardner

Jody Larsen

Technical Editor

Shiuh-Lin Lee

Team Coordinator

Lynne Williams

Pamalee Nelson

Media Developer

Dan Scherf

Interior Designer

Anne Jones

Cover Designer

Aren Howell

Dedication

To my parents

—Michael Kroll

To Janine

—Stefan Haustein

About the Authors

Michael Kroll studied Medical Computer Science from winter 1998 to winter 2000 at the University of Applied Sciences in Dortmund Germany. His diploma thesis with the subject "Decentral Biosignal Processing—Creating a Concept and an Implementation for Palm Connected Organizers Using J2ME" was an interdisciplinary work in cooperation with the Department of Radiology and MicroTherapy of the University of Witten/Herdecke Germany. Starting in November 2000, he became a doctoral student at the University of Applied Sciences Dortmund Germany in cooperation with the University of Witten/Herdecke Germany.

Both Michael and Stefan are the authors of the kAWT Project, an AWT subset for the KVM and members of Java PDA profile specification expert group (JSR-000075 PDA Profile for J2ME).

Stefan Haustein studied Computer Science from winter 1990 to the beginning of 1998 at the University of Dortmund. He wrote his diploma thesis in the Neuros-Project at the "Institut für Neuroinformatik" at the University of Bochum about graph-based robot navigation.

Since April 1998, he has been working as a Ph.D. student at the AI-Unit of the Computer Science department of the University of Dortmund.

Acknowledgments

We would like to thank all the professionals at Sams for their efforts helping us create this book. We would like to thank Carol Ackerman for her support and encouragement as we worked to meet our deadlines and for the nice meeting in the "beer-research-facility" at Leichlingen/Germany.

Special thanks to Tiffany Taylor for developing the book.

Special thanks to Shiuh-Lin Lee for his technical editing and for testing our demo applications provided in the book.

We would also like to thank Michael Stephens for giving us the opportunity to write this book

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an executive editor for Sams Publishing, I welcome your comments. You can fax, e-mail, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and authors' names as well as your name and phone or fax number. I will carefully review your comments and share them with the authors and editors who worked on the book.

Fax:	317-581-4770
E-mail:	feedback@samspublishing.com
Mail:	Michael Stephens Executive Editor Sams Publishing 201 West 103rd Street Indianapolis, IN 46290 USA

For more information about this book or another Sams or Que title, visit our Web site at www.samspublishing.com or www.quepublishing.com. Type the ISBN (excluding hyphens) or the title of a book in the Search field to find the page you're looking for.

Introduction

At the JavaOne conference 1999, the Java 2 Micro Edition (J2ME) was initially introduced. J2ME is a Java 2 platform, specially designed for embedded devices such as consumer electronics, cell phones, and PDAs.

This book covers the *Connected Limited Device Configuration (CLDC)* and the two profiles available for CLDC, the *Mobile Information Device Profile (MIDP)* and the *Personal Digital Assistant Profile (PDAP).*

> **Note**
>
> Note that the PDAP-related information covered in this book is based on the PDAP Public Draft Specification, which may slightly differ from the final specification; see http://www.jcp.org/jsr/detail/75.jsp.
>
> Corresponding updates, including sample applications, can be found at the Web site of this book—http://www.samspublishing.com; enter the book's ISBN (0672320959).

The intention of this book is to help you to understand the architecture the J2ME technology, especially CLDC, MIDP, and PDAP, and show you how to use it by creating real-world sample applications.

Audience

This book is intended for Java developers with general Java experience gained by creating Java 2 Standard Edition applications for the desktop computer. If you aren't yet familiar with Java syntax and semantics, we recommend that you first start by using Java tutorials available through the Web or by going to book stores focusing on the Java language. We do not assume that you have J2ME development experience at this point.

The Structure of This Book

This book is organized in ten chapters and two appendixes. The chapters and appendixes cover the topics listed as follows:

Chapter 1, "Java 2 Micro Edition Overview," gives a general overview of the *Java 2 Micro Edition (J2ME).* Starting from the Green Project, the origin of the J2ME technology, it introduces the CLDC and its profiles, tells you how to create a small hello world application, and gives an overview about currently available CLDC based development tools.

Chapter 2, "The Connected Limited Device Configuration," describes the general concepts and limitations of the CLDC and also takes a closer look at the API packages available in it. The packages and classes of the *Mobile Information Device (MID)* and *Personal Digital Assistant (PDA)* profiles, including extensions of CLDC classes, are then discussed.

Chapter 3, "MIDP Programming," handles the life cycle and user interface of MIDP applications and discusses their general design. Then, the high-level user interface API will be explained. Finally, the low-level user interface API for free graphics and games will be described, and we will give a short overview of the new MIDP 2.0.

Chapter 4, "PDAP Programming," handles the life cycle and user interface of PDA applications. First, the general design of PDA applications will be discussed. Then, the AWT subset forming the PDAP user interface API will be explained.

Chapter 5, "Data Persistency," describes the *Record Store Management System (RMS)* that is used in MIDP and PDAP to store data on a cell phone or PDA persistently. We will show the basic use of Record Stores and tell you how to use iterators in order to access stored records. Finally, we describe the new RecordStore methods, which are added by MIDP 2.0.

Chapter 6, "Networking: The Generic Connection Framework," describes a framework that is used in J2ME for network connections. We will describe the structure of this framework, show you how to establish Socket, Datagram, and HTTP connections, and show their use by implementing a client/server chat application.

Chapter 7, "PIM: Accessing the Personal Information Manager," describes the Personal Information Manager that is included in PDAP. Here, we will show you the concepts of the PIM and also show you how to create a demo application for handling contacts.

Chapter 8, "Size Does Matter: Optimizing J2ME Applications," shows you how to optimize J2ME applications by describing some essential hints for J2ME application programming.

Chapter 9, "Advanced Application: Blood Sugar Log," shows you how to create an advanced application by splitting its functionality into two parts; one basic part that can be used in a MIDP and PDAP application as well and the profile specific GUI part in which a profile specific user interface is created.

Chapter 10, "Third-Party Libraries," by creating small example applications, this chapter introduces some third-party libraries such as a substitution for the lack of floating point support, XML parsers, and a SOAP implementation for J2ME.

Appendix A, "Class Library: CLDC Packages," gives an overview of all packages included in CLDC, MIDP, and PDAP.

Appendix B, "Comparison Charts," includes a great set of tables where we compare classes of J2ME with their J2SE counterparts. Each package that is available in J2SE and J2ME is listed in detail to show you which classes and methods are available in CLDC or CLDC-NG (also known as CLDC version 1.1). If applicable, we provide workarounds to substitute a particular method of J2SE by a corresponding set of calls or using other classes to gain the same functionality as in J2SE.

Software Development Kits Used to Create the Example Applications

In order to compile and build the MIDP based example applications provided in this book, we recommend that you use Sun's J2ME Wireless Toolkit v1.0.3 in standalone mode (not the Forte integration). Follow the installation steps provided by Sun to set up the Wireless Toolkit properly.

Since a reference implementation for PDAP isn't available at the time of publishing this book, we recommend that you use the J2SE desktop AWT in order to create PDAP AWT applications. In order to get emulations of the additional J2ME packages, such as RMS or the Generic Connection Framework for J2SE, refer to the ME4SE project, available at the Web site http://me4se.org.

If you want to run a PDAP application on a real PDA such as a Palm organizer, you can use the Jbed VM that is available from the Swiss company esmertec Inc. (http://www.esmertec.com) together with a kAWT implementation supporting most of the PDAP features at this time. For PDAs where a Personal Java implementation is available, you can also use Personal Java, together with some of the ME4SE classes.

Web Site

You can download the source code for the example applications discussed in this book from www.samspublishing.com. When you reach the page, just enter the book's ISBN (0672320959) and follow the Source Code link.

Chapter 1. Java 2 Micro Edition Overview

IN THIS CHAPTER

This chapter gives a general overview of the Java 2 Micro Edition (J2ME). Starting from the Green Project, the origin of this fascinating new technology, this chapter gives a short summary of the evolutionary process leading to the actual configurations and profiles specified in the Java Community Process (JCP). Then, it discusses the Software Development Kits available from Sun, including a short programming example. It also describes the additional building steps necessary for the Connected Limited Device Configuration (CLDC), Mobile Information Device Profile (MIDP), and the PDA profile (PDAP). Finally, it gives a brief description of some third-party products relevant for J2ME application development.

> **Note**
>
> If you are already familiar with the history of Java and the Java 2 Micro Edition technology, you can skip this section and go directly to the description of the J2ME configurations and profiles.

Historical Evolution

Looking back, several projects have dealt with Java-related programming languages used on consumer devices. These projects were mostly feasibility studies and they were never standardized; but, many of the ideas were incorporated into the J2ME standards. It makes sense to look at the origins of these components to gain a better understanding of why things are as they are in J2ME.

The Green Project

The Green Project began in December 1990. Some people at Sun decided to try to figure out what the next step would be in the evolution of computing and how they could be part of it. They came to the conclusion that the next evolutionary step would be the merger of digitally controlled consumer devices and computers—the kinds of devices we know as Personal Digital Assistants (PDAs) today.

The Sun engineers developed a new SPARC-based, wireless handheld PDA called Star7 or *7. This newly created device was equipped with a 5-inch, 16-bit color LCD with touch-screen input capabilities. In addition, it was capable of communicating with other *7 devices over a built-in 900MHz wireless network. This small device required the development of an OS that would fit in only one megabyte of RAM.

The original plan was to develop the Star7 operating system using C++. However, one of the members of the Green Project, James Gosling, became fed up with C++ and decided to develop a new programming language. The result of his work was a programming language called Oak, which was especially designed to run on devices with significant resource constraints, just like Star7. Thus, Oak had to be very small, efficient, and easily portable to other hardware devices.

Oak was the original ancestor of the Java programming language, which has all the properties just mentioned. (Another byproduct of the Green Project was the Duke, which became the official Java mascot. The Java Duke is a personification of the *agent* in the user interface of Star7, similar to Microsoft's paper clip.) Star7 was finished and officially presented on September 3, 1992.

In only a few years of development, Java grew to be a leading programming language on desktop computers. However, much of the original focus—to fit a language onto portable computers—was lost during Java's evolution. The new Java development goal became fast execution on desktop machines, regardless of the size of the Java Virtual Machine (JVM). In addition, the standard libraries were extended to several megabytes for developer convenience.

But in 1998, SunLabs got back on the original track and started a new research project, the Spotless System. The goal of the new project was to create a portable JVM that was suitable for embedded systems.

The Spotless System

As mentioned, implementors of Java focused on increasing the speed of the JVM, leading to memory-consuming technologies such as HotSpot. No effort was made to keep those systems small because for desktop systems, the size was not relevant. Consequently, those Java implementations were not suitable for embedded systems offering only a small amount of memory and limited computing power.

This situation changed with the newly created virtual machine of the Spotless System, which was especially designed to fit the constraints of embedded systems. This project had the following main goals:

- Build the smallest possible complete JVM that supports the full bytecode set, including class loading and non-graphical libraries.
- Implement the new JVM in highly readable source code in order to provide the best portability to available hardware platforms.

The result of the Spotless System project was a small JVM that occupies less than 300 kilobytes of static memory on a PC system. In order to create an implementation for a real-world embedded device, the engineers first targeted the Rolodex REX personal organizer developed and distributed by Franklin Electronic Publishers. However, this device lacked a development kit, so the engineers switched to the Palm Connected Organizer as a reference platform for their JVM implementation, where excellent support for developing software solutions is available. In addition, the Palm PDA is the most popular PDA currently available in the market.

The original Spotless JVM implementation for the Palm PDA included only a small subset of the class libraries available for desktop Java. As you can see in Table 1.1, the subset is very small even when compared to the actual J2ME configurations discussed in the section "J2ME Configurations and Profiles." Although the class libraries were sufficient to show feasibility of Java development for embedded systems, they still had some major drawbacks. So the GUI components offered by the spotless package were device-specific to the Palm Connected Organizer.

Table 1.1. Packages and Classes of the Spotless System

Package Name	***Included Classes***
`java.lang`	`Class`, `Error`, `Exception`, `IndexOutOfBoundsException`,

	NullPointerException, Object, Runnable, Runtime, RuntimeException, String, StringBuffer, Thread, Throwable
java.io	InputStream, IOException, OutputStream, Serializable
java.net	InetAddress, ProtocolException, Socket, SocketException, SocketInputStream, SocketOutputStream, UnknownHostException
spotless	Beam, Bitmap, Component, Database, Event, External, ExternalException, ExternalManager, Field, Form, Graphics, IO, Label, List, Lst, NativeIO, PButton, Spotlet

The JVM developed in the Spotless System consists of a central application that acts as a class launcher, analogous to the Java command on the desktop. But in contrast to the desktop Java, it includes a complete list of all available Java classes. It gives the user the ability to run any class containing a static `main` method.

The JavaOne99 KVM Preview Version

At the JavaOne99 conference, some results of the Spotless project got their official place in the Java family. The Java Technology was split into three categories: Java 2 Enterprise Edition (J2EE), Java 2 Standard Edition, and the new Java 2 Micro Edition (J2ME). The heart of J2ME is a new virtual machine, which is specially designed for embedded systems, cellular phones, and PDAs. Because of its low memory footprint of only a few kilobytes, the new virtual machine was named Kilobyte Virtual Machine (KVM). In fact, Sun did not merely announce the new technology; it showed a preview version for PalmOS.

The new KVM was, as you may already have guessed, directly derived from the Spotless System project. However, there are some changes in the supported package names, some classes are canceled, and other classes are added (see Table 1.2). It's still possible to browse through all included Java class files, and all runnable `Spotlet`s that include a `main` method are now listed in the PalmOS application launcher. The package `spotless` has been renamed `com.sun.kjava`; it now includes an enhanced version of the Palm-specific GUI classes.

Table 1.2. Packages and Classes Included in the JavaOne99 KVM

Package Name	***Included Classes***
java.lang	Class, Error, Object, Runtime, String, StringBuffer, Thread, Throwable, Exception, IllegalAccessException, IndexOutOfBoundsException, NullPointerException, Runnable, RuntimeException
java.io	InputStream, IOExceptopn, OutputStream, Serializable
java.net	Socket, SocketException
com.sun.kjava	Bitmap, Button, Caret, CheckBox, Database, Dialog, DialogOwner, Graphics, HelpDisplay, IntVector, List, RadioButton, RadioGroup, ScrollOwner, ScrollTextBox, Slider, Spotlet, TextBox, TextField, Trigonometric, ValueSelector, VerticalScrollbar

Note

This class overview is not intended to replace an API reference. It simply describes the formation of the KVM packages and their classes compared with the Spotless System.

Beneath downsized versions of some standard Java packages, the first KVM contained some Palm-specific GUI classes, mostly derived from the Spotless project. A short look at the size of the Java standard package set shows the next problem: A small virtual machine is not really useful without small libraries.

During the months after the JavaOne conference in 1999, some Early Access (EA) versions were released for registered Java developers only. Between those releases, the APIs somehow changed—many bugs were fixed between EA version 0.1 and 0.2, including many virtual machine bugs, and some improvements were made to the `com.sun.kjava` classes, as well.

Micro Edition–Related Java Specification Requests

After KVM EA version 0.2, a great change took place. KVM EA 0.2 was developed by Sun only; but after that, the KVM technology began to be specified during the Java Community Process (JCP) by many companies that participate in a Java Specification Request related expert group.

The Java Community Process (JCP)

The JCP program was initiated by Sun on December 8, 1998, in order to create a fast and flexible formal system for the development and revision of Java technology specifications. This process lets the Java community, as well as Sun engineers, participate in the specification process of creating new Java APIs.

The main goals of JCP are to enable the wide-ranging Java community to participate in creating proposals, as well as selecting and developing new Java APIs. This process enables Java community members to advise API development efforts without needing to involve Sun engineers.

The whole process follows key milestones that enable a new specification to be drafted in a given period of time. When the Specification is approved, a Reference implementation and an additional Technology Conformance Kit follow, to enable licensees to create an implementation that is compliant with the newly specified technology.

The JCP is described in more detail on the following Web site:

http://jcp.org/

Please take a look at the documents offered by Sun if you are interested in getting more information about this topic.

Proposals for new Java Specifications are called Java Specification Requests (JSR). Those requests are not only used to create or develop new Java APIs, but also to renew or modify existing Java APIs. If developers in the Java community are interested in submitting a JSR, they must first sign a Java Specification Participation Agreement (JSPA). After they are community members, they can use a JSR template (available from Sun Microsystems Inc.), in which they specify the goals of the proposal.

The following list shows the most important J2ME-related JSRs that are currently available. At the time of this writing, the following three JSRs have been specified and are available as final releases and reference implementations:

- The Connected Limited Device Configuration (CLDC) JSR000030

URL to the specification:

http://jcp.org/jsr/detail/30.jsp

URL to download the reference implementation:

http://www.sun.com/software/communitysource/j2me/

- The Mobile Information Device Profile (MIDP) JSR000037

  URL to the specification:

  http://jcp.org/jsr/detail/37.jsp

  URL to download the reference implementation:

  http://www.sun.com/software/communitysource/midp/

- The Personal Digital Assistant Profile (PDAP) JSR000075

  URL to the specification:

  http://jcp.org/jsr/detail/075.jsp

  At the time of this writing, the reference implementation was not yet available. Refer to the book's Web site in order to get a valid URL to download the reference implementation when it becomes available.

- The Connected Limited Device Configuration 1.1 JSR000139

  URL to the specification:

  http://www.jcp.org/jsr/detail/139.jsp

- The Mobile Information Device Profile 2.0 JSR000118

  URL to the specification:

  http://www.jcp.org/jsr/detail/118.jsp

These JSRs are discussed in detail in the next section.

As of this writing, other J2ME-related JSRs belonging to the Connected Device Configuration are in the specification process. Please refer to the following URLs to obtain further information about these JSRs and their current status:

- The J2ME Platform Specification JSR000068

  http://jcp.org/jsr/detail/68.jsp

- The Connected Device Configuration (CDC) JSR000036

  http://jcp.org/jsr/detail/36.jsp

- The Foundation Profile JSR000046

  http://jcp.org/jsr/detail/36.jsp

- The Personal Profile JSR000062

  http://jcp.org/jsr/detail/62.jsp

- The Personal Basis Profile JSR000129

  http://jcp.org/jsr/detail/129.jsp

- The RMI Profile JSR000066

  http://jcp.org/jsr/detail/66.jsp

  Note

  Because this book is intended to cover CLDC-based profiles, it gives only a short overview about the JSRs that are available for CDC and focuses on covering CLDC-based applications only.

  A comparison between CDC and CLDC appears in the next section.

J2ME Configurations and Profiles

Obviously, the standard Java libraries are just too big for the Java 2 Micro Edition. Once the KVM was available, the next logical step was to define appropriate libraries. But just downsizing the standard libraries was not sufficient: The target devices have many special requirements that also must be covered by libraries.

For example, many PDAs and all cellular phones do not have a file system. Instead, data is stored persistently in simple databases in buffered RAM or flash memory. Obviously, a KVM library would need to provide access to this kind of storage. Moreover, the specific needs diverge for the potential KVM devices. A set top box (a device that decodes interactive TV signals) does not have much in common with a cellular phone except that they both normally provide a small amount of CPU power.

For these reasons, Sun decided to design several KVM profiles, one for each group of target devices. Examples of KVM profiles are the Mobile Information Device Profile (MIDP) for cellular phones and the PDA Profile for PDAs.

Like other official Java libraries, the profiles are designed in the Java Community Process. A novelty in version 2.0 of the JCP is that not just companies but also individuals can participate. For example, the authors of this book are participating in the PDAP specification process as invited experts.

The profiles are designed on top of KVM *configurations.* Whereas the profiles mainly address device-type–specific issues, the configurations summarize the available basic KVM functionality for devices with similar computing power and equipment characteristics. The following sections describe KVM configurations and profiles in more detail.

Configurations

Currently, just two configurations—CLDC and CDC—exist. The CDC profiles are still in the specification process, but the CLDC is finished and forms the basis for MIDP and PDAP.

The CLDC was designed especially for the mobile phone and PDA class of devices. It requires 128 to 512KB of memory (RAM and ROM), a battery power supply, and a network connection of at least 9600bps.

The CLDC API contains simplified versions of `java.lang`, `java.io`, `java.util`, and the new package `javax.microedition.io`, described in more detail in Chapter 6, "Networking: The Generic Connection Framework."

Because CLDC was specified for processors that may not provide floating-point support, `float` and `double` are not supported. However, CLDC 1.1, the next generation of the CLDC profile, adds support for floating point operations. Support for Java Native Interfaces (JNI) also is not included in CLDC. The reflection API is very limited; for example, user-defined class loaders are not available. Finalization is not supported.

Another restriction is that class files need to be *preverified* before execution with the KVM. The preverification step inserts hints into the class files that simplify and speed up the actual verification of the classes on the device. The preverification step includes a check for invalid data types, so class files containing `float`s will be rejected in this step for the original CLDC profile.

The second configuration available, CDC, targets more powerful devices like set top boxes, video phones, and gaming consoles with at least 512KB ROM and 256KB RAM as well as a fast network connection.

Profiles

In contrast to the configurations, which are independent of the device's purpose, the profiles cover aspects that are specific to a certain device type. For example, the profiles cover the user interface and persistent data storage. Currently, two profiles are available: MIDP and PDAP.

The Mobile Information Device Profile (MIDP)

MIDP targets cellular phones and simple pagers. It provides a very simple and abstract user interface built of simple elements. The user interface is divided into a high-level and a low-level API. The high-level API provides simple elements such as lists and forms, but it offers only very limited control over the concrete appearance on the screen. The low-level API provides full control over the screen, but no widgets; it's mainly intended for games. The UI API is not compatible with any other Java UI API, such as AWT or SWING.

The Personal Digital Assistant Profile (PDAP)

Just as the name suggests, PDAP targets PDAs. It provides a user interface that is a subset of the AWT package of the Java 2 Standard Edition and access to the Personal Information Management databases of the device. In contrast to MIDP, PDAP is based on the newer CLDC-NG configuration because the AWT classes require floating point support. In order to access existing MIDP applications, the PDA profile is a complete superset of the MID Profile. Thus, any MIDP application can run on devices supporting PDAP. However, in contrast to MIDP devices, which have always a wireless connection, existing PDAs don't necessarily provide permanent network access.

Because the PDA profile is a superset of the MID profile, applications that do not require a sophisticated user interface or PIM access should be based on the MID profile. Table 1.3 shows a short comparison of the profiles.

Table 1.3. MIDP and PDAP Comparison

	MIDP	*PDAP*
Available on phones	Yes	No[*]
Available on PDAs	Yes	Yes
Basic UI capabilities	Yes	Yes
Wireless Internet access	Yes	Yes[*]
Sophisticated UI capabilities	No	Yes
Address book access	No	Yes
Calendar access	No	Yes

[*] *Availability is device-dependent*

Sun J2ME Software Development Kits

This section gives a short overview of the J2ME development tools that are currently available. Unless explicitly stated otherwise, these CLDC and the MIDP SDKs from Sun are used as the reference SDKs for the examples in this book. The Sun SDKs are basis for several IDE add-ins and are available for the widest range of operating systems.

Caution

Although the APIs are standardized, parts of the compilation process may change with newer SDK versions. If you are in doubt or observe problems with the following instructions, please refer to the documentation provided with your SDK.

Compared to developing for the desktop, targeting the Java 2 Micro Edition requires some additional tools—a preverifier, at the least. Most SDKs do not bring in their own Java compiler; they rely on an installed JDK 1.2 or 1.3. Several SDKs contain a device emulation. Of course, it would also be possible to test the programs on the target devices directly. However, the emulations normally allow faster installation, and most of them also provide some additional debugging support.

Sun's J2ME CLDC Reference Implementation 1.0.3

The version 1.0.3 of the CLDC reference implementation (RI) does not contain any GUI classes. It is available for Windows 98/NT (Win32), Linux, and Solaris. The CLDC RI provides command-line tools only. For easier development, several integrated development environments provide CLDC plug-in support.

Note

You can download the Sun CLDC reference implementation from the following URL:

http://www.sun.com/software/communitysource/j2me/

The CLDC reference implementation ships without a Java compiler, so an additional Java compiler like the one contained in the Java Development Kit for the Java 2 Standard Edition is needed to actually build KVM programs. Also, the process for compiling J2ME applications is somewhat different from desktop development. For example, it is necessary to specify the compiler parameter `-bootclasspath`, in order to compile the application with the J2ME libraries instead of the standard desktop environment. These additional steps are described in detail in the section "Developing a Simple Application."

Sun's MIDP Reference Implementation v1.0.3

In addition to the CLDC RI, Sun offers an implementation of the MIDP as well. Similar to the CLDC RI, the MIDP implementation is distributed under the Sun Community Source License and ships without a Java compiler.

The MIDP Development Kit includes the complete source for its supported target platforms—Win32, Linux, and Solaris—available in separate files which are available for download. The MIDP SDK includes several example applications, ranging from simple demos showing how to use common MIDP widgets to complete games, such as the TitlePuzzle game (see Figure 1.1). Moreover, it includes a description of how the included development tools are used.

Figure 1.1. The MIDP Emulator on the Windows32 platform, running a Sokoban game.

Note

You can download the Sun MIDP reference implementation from the following URL:

http://www.sun.com/software/communitysource/midp/

Tools and Third-Party Products for J2ME Application Development

In addition to the Sun SDKs, several other J2ME SDKs, IDEs, and related products from other vendors are available. We'll discuss a number of these tools and products in this section.

Sun's J2ME Wireless Toolkit 1.0.3

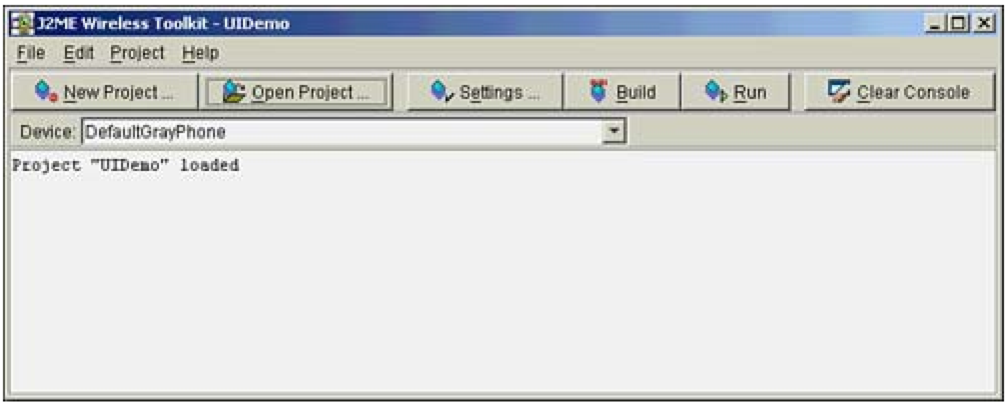
The J2ME Wireless Toolkit is a project management tool intended to simplify MIDP development. It can perform the compile and preverification steps automatically, including the generation of the JAR and JAD files. It is not a complete Integrated Development Environment (IDE), so it does not offer an editor. The MIDP SDK is included in the Wireless Toolkit, but the JDK version 1.3 is also required.

In addition to the MIDP SDK, the Wireless Toolkit contains some additional skins for the device emulation (see Figure 1.2). The additional skins are a pager with a complete keyboard and cell phone emulation with a thumbwheel, as well as support for the Palm OS Emulator, RIM pagers, and Motorola cell phones.

Figure 1.2. The Wireless Toolkit running the HelloMidp MIDlet in different device emulations.

The main window of the toolkit, shown in Figure 1.3, consists of some buttons for creating a new project, opening existing projects, and building and running the project in the specified device emulation.

Figure 1.3. The main window of the J2ME Wireless Toolkit, showing the project HelloMidp.



To edit the source code of the project files, you can use your favorite editor, such as Emacs or Windows' simple NotePad. Another option is to use Forte with the plug-in contained in the Wireless Toolkit. Together with Forte and JDK 1.3, the Wireless Toolkit forms a complete IDE for MIDP.

Note

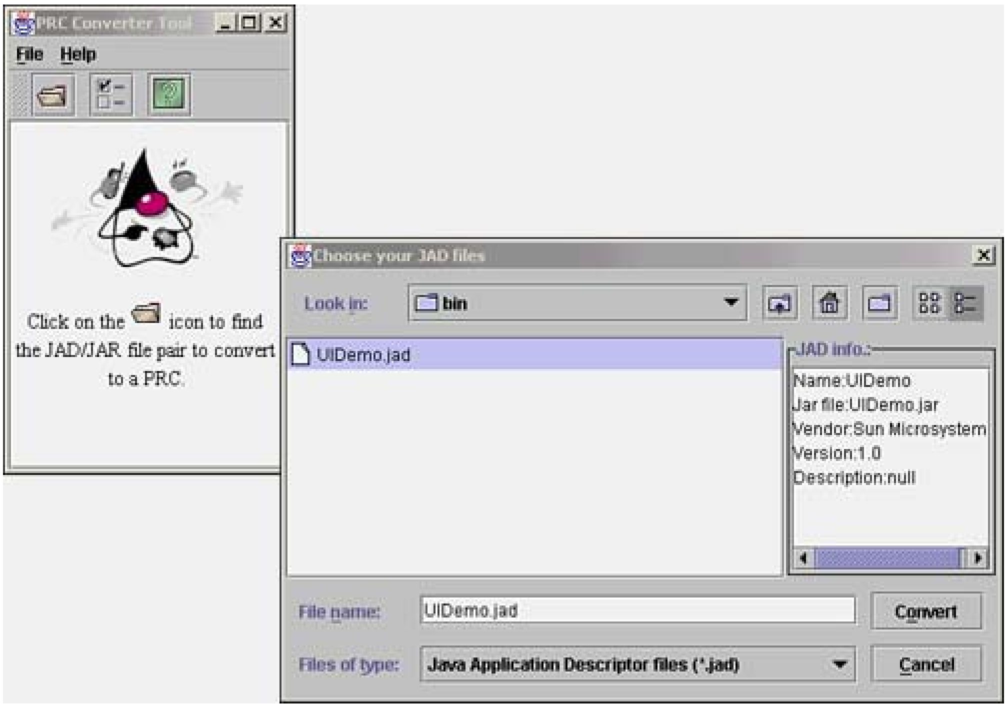
The J2ME Wireless Toolkit is available from the following URL:

http://java.sun.com/products/j2mewtoolkit/

Sun's MIDP for Palm OS

The MIDP for Palm OS is the first Mobile Information Device Profile implementation from SUN, running on a real mobile device. It consists of a set of executables in Palm prc-format and a converter capable of converting MIDP JAD/JAR file pairs (generated, for example, using the J2ME Wireless Toolkit) to the Palm MIDP format. Figure 1.4 shows the PRC Converter Tool included in the MIDP for Palm OS distribution.

Figure 1.4. The PRC Converter Tool of MIDP4Palm.



The MIDP for Palm OS implementation can be used in conjunction with the J2ME Wireless Toolkit to create and convert `MIDlet`s in order to run them on a Palm or on the Palm OS Emulator without additional tools (except an editor to create the source). Figure 1.5 shows the UIDemo application that is included in the J2ME Wireless Toolkit on a Palm Pilot.

Figure 1.5. The UIDemo MIDlet that is included in the J2ME Wireless Toolkit running on a Palm Pilot.

Note

You can download the Mobile Information Device Profile implementation for PalmOS from the following URL:

http://java.sun.com/products/midp4palm/index.html

esmertec's Jbed Micro Edition CLDC and Jbed Profile for MID

The Swiss company esmertec Inc. offers a CLDC-compatible JVM and the MID profile. The CLDC and the MID version of Jbed come with the Jbed Integrated Development Environment (IDE), shown in Figure 1.6, in order to simplify application development.

Figure 1.6. The Jbed IDE showing the development of a MIDletSuite.

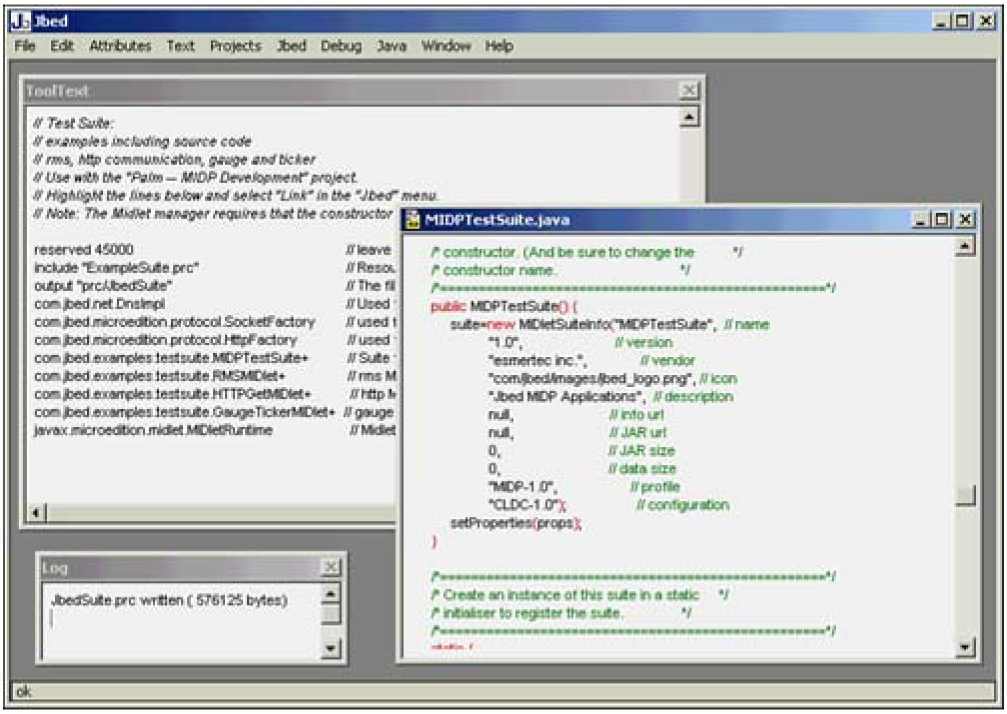


Figure 1.7 shows the sample MIDPTestSuite, which is bundled with the Jbed Profile for MID. The main advantage of Jbed is that it is the fastest JVM for embedded devices currently available on the market.

Figure 1.7. The MIDPTestSuite running on a Palm Pilot.

Jbed obtains its high speed by compiling Java programs to native code. The compilation can be performed on the device itself or in advance on the desktop. Applications running on Jbed run incredibly quickly compared to other KVMs.

It is amazing that it is possible to get a complete class-to-native compiler on a constrained device like the Palm Pilot. Moreover, Jbed provides real-time capabilities. So if you are planning to create software for embedded systems, and Jbed is available for that platform, Jbed is probably the optimal choice. However, for devices with a build in JVM, such as MIDP- powered cell phones, in most cases it will not be possible to use Jbed.

Jbed does not include a device emulation, but a Palm Operating System Emulator (POSE) is available directly from Palm Inc. without charge.

Note

Use the following link to get more information about Jbed CLDC and the Jbed Profile for MID from esmertec:

http://www.esmertec.com/

Borland's JBuilder MobileSet, Nokia Edition

The JBuilder MobileSet is a J2ME CLDC- and MIDP-compliant development environment. It is fully integrated with JBuilder 5 in order to simplify development of Java applications for mobile devices.

The MobileSet includes the following features:

- Wizards for creating MIDP Projects and single MIDlets
- Debugging MIDlets in NOKIA device emulators
- Designer for Rapid Application Development of MIDlets
- Deployment tools for creating JAR/JAD file pairs

Note

Use the following link to get more information about Borland's JBuilder MobileSet, Nokia Edition:

http://www.borland.com/jbuilder/mobileset/

Metrowerks Codewarrior for Java, Version 6.0

The Codewarrior for Java Version 6.0 offers a fully integrated development environment for Java applications and supports J2ME CLDC and MIDP application development as well. It supports a set of tools for J2ME application similar to those supported by JBuilder's MobileSet:

- Project management for J2ME applications
- MIDlet debugging in emulators
- Deployment tools for creating JAR/JAD file pairs

Note

Use the following link to get more information about Metrowerks Codewarrior for Java:

http://www.metrowerks.com/desktop/java/

Developing a Simple Application

This book would not be a true programming book if the "Hello World" example was missing. Actually, we need three different "Hello World" examples to cover the CLDC reference implementation from Sun as well as the MIDP and PDAP CLDC profiles. The following sections describe the steps necessary to compile and run a simple KVM program for each target platform.

Setting Up the System Environment Variables

If you are using one of the integrated development environments, you can probably skip the installation parts of this section. Just type the example and click Build or Compile and then Run, depending on the type of IDE you are using.

For command-line operation, it is helpful to insert the `bin/win32` directory located in your J2ME installation into the system search path. For the MIDP SDK, you need to add the `build/win32/tools` directory to the system path as well. For Windows 95/98, this is performed by adding the corresponding directory to the `path` command in the file `c:\autoexec.bat`. If you're running Windows NT 4.0 or Windows 2000, open the Properties dialog box by right-clicking on My Computer on your desktop and selecting Properties. The path information is located in the Extended tab.

The `kvm` and `preverify` utilities are also available for Solaris and Linux. However, the executables are located in different directories or download packages. Please refer to the corresponding documentation for installation on Unix systems.

Testing the Setup

In order to work with the command-line oriented SDKs, the first step is to open a command line. Windows users just need to click the Start button and choose Programs, followed by MS-DOS Command Line. For Unix, the way a command shell window is opened differs, depending on the Desktop Manager and the exact system setup. For many installations, you just need to click on the shell symbol in the start bar.

Before beginning, it makes sense to perform a short test to see whether the environment is set up correctly. Please do not skip the test: It will probably save you from spending time searching for simple and avoidable problems.

To test whether the `preverify` command is in the system search path, type in the following command:

```
C:\> preverify
```

The command should generate the following output (or similar):

```
Usage: preverify [options] classnames|dirnames ...

where options include:
   -classpath <directories separated by ';'>
                  Directories in which to look for classes
   -d <directory> Directory in which output is written (default
is ./output/)
   @<filename>    Read command line arguments from a text file
```

If you get a "command or file name not found" error message, the `PATH` environment variable probably is not set correctly. Include the `bin` directory of the KVM installation in your system search path.

Now perform the same test for the `javac` command:

```
C:\> javac
```

Again, if the system returns a "command or file name not found" error message, ensure that a JDK is installed and also that the JDK `bin` directory is in the system search path, as described in the previous section.

Now that you are sure your system is set up properly, you're ready for real programming. You can go directly to the subsection of the target profile for which you are planning to program, or you can go through all three if you want to get an overview of the profiles.

CLDC KVM Reference Implementation

Begin by setting an environment variable that points to the CLDC and Kjava classes:

Windows:

```
set CLDC_BCP=c:\j2me_cldc \bin \common \api \classes
```

Unix/Csh:

```
setenv CLDC_BCP ~/j2me_cldc/bin/common/api/classes
```

Unix/Bash:

```
export CLDC_BCP=~/j2me_cldc/bin/common/api/classes
```

Note that the actual directory containing the CLDC and Kjava classes depends on the exact installation and version number; it may differ from this example. If so, set the `CLDC_BCP` variable accordingly.

On Windows 98, you can add the previous line to the file `c:\autoexec.bat`. On Unix systems, add the line to the corresponding startup or login script. The changes will then affect all command lines automatically. If you're running Windows NT 4.0 or Windows 2000 Professional, the system environment variables are set in the same place as the system search path. Open the system properties dialog by right-clicking My Computer on your desktop and selecting Properties. Then, choose the Extended tab and select Environment Variables to set the `CLDC_BCP` variable permanently. Select New for User Variables to open a dialog box in which you can set the name (for example, `CLDC_BCP`) and the value (`c:\j2me_cldc\bin\api\classes`) of the variable. When you click OK, the new variable will be stored permanently and will be available in all new command shells. Shells already started are not affected, so they must be closed and restarted.

Finally, check whether the environment variable pointing to the CLDC and Kjava classes is set properly:

Windows:	`echo %CLDC_BCP%`
Unix:	`echo $CLDC_BCP`

Now that you have made sure the compiling environment is set up properly, you can start with the sample application. Listing 1.1 contains the complete source code of the `HelloCldc.java` file.

Listing 1.1 `Hello Cldc.java`—The `HelloCldc` Sample Source Code

```
public class HelloCldc {

    public static void main (String [] args) {
        System.out.println("Hello CLDC.");

    System.out.println("This application is running on a "
            + System.getProperty("microedition.configuration")
            + " JVM");
    }
}
```

You now can compile the sample program using the following command:

Windows:	`javac bootclasspath %CLDC_BCP% HelloCldc.java`
Unix:	`javac bootclasspath $CLDC_BCP HelloCldc.java`

The `bootclasspath` parameter is necessary because the program cannot be compiled with the standard Java desktop libraries; it must access the CLDC and Kjava libraries.

If the compile command line does not produce any error message, you can perform the preverify step:

Windows:	`preverify -classpath .;%CLDC_BCP% HelloCldc`
Unix:	`preverify -classpath .:$CLDC_BCP HelloCldc`

This step is necessary to include verification hints in the class file, simplifying the class verification on the target device. Future `javac` versions may have a switch to perform preverification at compilation time, simplifying J2ME application development.

The preverify step creates a new subdirectory named `output`, where the preverified classes are placed.

If the preverify step is successful, you can test the application with the command line kvm:

```
cd output
kvm HelloCldc
```

These commands change the current directory to the output directory containing the preverified files and start the kvm with the new verified `HelloCldc` class. The following output will be generated:

```
Hello CLDC.
This application is running on a CLDC-1.0 JVM
```

Hello MIDP

Similar to using the CLDC RI, the first step is to set up an environment variable pointing to the MIDP library classes. The environment variable simplifies the following compilation steps and helps avoid annoying problems resulting from typos in long path names:

Windows:	`set MIDP_BCP=c:\midp-fcs\classes`
Unix/Csh:	`setenv MIDP_BCP ~\midp-fcs\classes`
Unix/Bash:	`export MIDP_BCP=~\midp-fcs\classes`

You can make sure that the system path is set properly by typing **`MIDP`** on the command line. If the cell phone emulation appears, and the current directory is not the MIDP `bin` directory, the system path is set correctly. Please refer to the previous section if the path is not set or if you would like to set the variable(s) permanently.

Note

The steps to create a runnable MIDP application from compiling through preverification and so on are very confusing for J2ME beginners. We will explain MIDP development using the low-level command-line tools. We recommend that you use the Wireless Toolkit or another IDE for actual development.

Again, we will compile a simple "Hello World" program. The corresponding Java program for the MIDP API is contained in Listing 1.2. Programming with the MIDP API is described in Chapter 3, "MIDP Programming." Here, we will focus on the compilation steps.

Listing 1.2 `HelloMidp.java`—The Source Code of the MIDP Sample

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

public class HelloMidp extends MIDlet {

    public void startApp() {
        Form form = new Form ("HelloMidp");
        Display.getDisplay (this).setCurrent (form);
    }

    public void pauseApp() {
    }

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

Change to the directory containing the Java file and compile the program using this command:

Windows:	`javac bootclasspath %MIDP_BCP% HelloMidp.java`
Unix:	`javac bootclasspath $MIDP_BCP HelloMidp.java`

Now, preverify the program:

Windows:	`preverify classpath .;%MIDP_BCP% HelloMidp`
Unix:	`preverify classpath .:$MIDP_BCP HelloMidp`

The preverification command creates a new `output` directory where it stores the preverified classes. You can test the MIDlet by typing the following commands:

```
cd output
midp HelloMidp
```

These commands change the current directory to the `output` directory containing the preverified files and start the MIDP emulator with the new verified HelloMidp class, as shown in Figure 1.8.

Figure 1.8. The running HelloMidp MIDlet.

In order to deploy programs that consist of multiple classes, JAR files from the desktop are used to bundle all necessary application classes together. Fortunately, you can just `jar` the complete output directory generated by the preverifier—you do not need to specify the filenames again for `jar`:

```
jar –cf hellomidp.jar output
```

After you successfully create the JAR file, you need to create a Java Application Descriptor (JAD) file. The JAD file encapsulates the information about the contents of the JAR file. In contrast to the manifest file, the JAD file is not included in the archive. It is intended to support automatic download of MIDlets. Listing 1.3 shows the JAD file for the `HelloMidp` example MIDlet.

Listing 1.3 `HelloMidp.jad`—The JAD File for the HelloMidp MIDlet

```
MIDlet-Name: HelloMidp
MIDlet-Version: 1.0
```

```
MIDlet-Vendor: Michael Kroll & Stefan Haustein
MIDlet-Jar-Size: 908
MIDlet-Jar-URL: hellomidp.jar
MIDlet-1: HelloMidp, , HelloMidp
```

The first five entries are self-explanatory (the jar size may differ depending on compiler options and jar compression). The last entry, `MIDlet-1`, determines the first and only MIDlet that is available in the JAR file. By adding similar entries, it is possible to put more than one MIDlet into a single JAD file and its respective JAR file.

The value of the MIDlet-*<Number>* attribute consists of three parameters:

```
MIDlet-<Number>: <name>, <icon>, <class>
```

<Number> is just an enumeration of the MIDlets in the JAR file. Our example contains only a single MIDlet, so there is only a single entry named `MIDlet-1`. If more MIDlets were stored in the JAR file, the entries would be named MIDlet-2, MIDlet-3, and so on.

The parameter *<name>* describes the name of the MIDlet that is displayed on the device after the descriptor file is loaded. The *<icon>* parameter specifies an icon that can be displayed in the list. The last parameter, *<class>*, specifies the name of the actual subclass of MIDlet implemented by the application.

In order to start the `HelloMidp` sample application, place the JAD file contained in Listing 1.3 in the same directory as the JAR file. Change to that directory and type the following command:

```
midp –descriptor hellomidp.jad
```

When executing this command, the MID emulation will show a selection menu (see Figure 1.9). Select the `HelloMidp` entry. The screen will show the output of the `HelloMidp` example (shown in Figure 1.8).

Figure 1.9. The information gained from the JAD file. In this case, only one entry (`HelloMidp`) is listed.

Note

It is not necessary for the JAR and JAD files to be in the same directory. The `MIDlet-jar-url` can point to any valid HTTP URL that references the corresponding JAR file.

Hello PDAP

Unfortunately, at the time this book was printed, no official PDAP implementation was available. However, we expect the compilation steps to be similar to compiling MIDP programs, except from using a different `bootclasspath`.

Actually, because PDAP is a complete superset of MIDP, the MIDP example contained in Listing 1.2 is a valid PDAP example as well. However, a "true" PDAP application will take advantage of the more sophisticated AWT-based user interface capabilities of PDAP. In order to sketch the differences, Listing 1.4 contains a PDAP example with the corresponding modifications.

Listing 1.4 `HelloPdap.java`—The `HelloPdap` Sample Source Code

```
import java.awt.*;
import java.awt.event.*;

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

```
public class HelloPdap extends MIDlet {

    private Frame frame;

    HelloPdap() {
        frame = new Frame("HelloPdap");
        frame.pack();
    }

    public void startApp() {
        frame.show();
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
        frame.dispose();
    }
}
```

Besides the application itself, the JAD files also need a slight modification. Again, any valid MIDP JAD file is also a valid PDAP JAD file. However, for "true" `PDAP` applications, using the advanced capabilities of the PDA profile, a new entry type `PDAlet-<Number>` must be used. If a `MIDlet-<Number>` entry with the same number `<Number>` is included, the `PDAlet` entry overrides the corresponding `MIDlet` entry. This allows you to include both MIDP and PDAP versions of the same application in a single pair of JAD and JAR files. The syntax of the `PDAlet` entries is identical to the syntax of the `MIDlet` entries. For our example, a corresponding JAD file including the MIDP version of the sample is shown in Listing 1.5.

Listing 1.5 `HelloPdap.jad`—The JAD File for the HelloPdap Application, and the HelloMidp MIDlet as a Fallback Option

```
MIDlet-Name: Hello
MIDlet-Version: 1.0
MIDlet-Vendor: Michael Kroll & Stefan Haustein
MIDlet-Jar-Size: ???
MIDlet-Jar-URL: hello.jar
MIDlet-1: HelloMidp, , HelloMidp
PDAlet-1: HelloPdap, , HelloPdap
```

Summary

In this chapter, you learned the history and background of J2ME and the CLDC configuration. We covered the MID and PDA profiles and gave you an overview of some existing software development kits. You should be able to set up the Sun development kits, to compile actual MID and PDA programs, and to run them in the corresponding device emulation.

The following chapters will first describe the Connected Limited Device Configuration (CLDC). Then, we'll revisit the MIDP and PDAP "Hello World" applications from an API point of view and explain the lifecycle of a MIDlet. We will also explain the user interface of MID and PDA applications in depth.

Chapter 2. The Connected Limited Device Configuration

IN THIS CHAPTER

This chapter describes the general concepts and limitations of the Connected Limited Device Configuration (CLDC). You already saw some of the limitations in Chapter 1, "Java 2 Micro Edition Overview." Here, you'll see a complete list, and also take a closer look at the API packages available in the CLDC. This chapter also discusses the packages and classes of the Mobile Information Device (MID) and Personal Digital Assistant (PDA) profiles including extensions of CLDC classes. Finally, it describes the special steps involved in J2ME application deployment.

General CLDC Limitations

In order to make the Java feature set suitable for very limited devices such as cellular phones or PDAs, CLDC's developers had to limit the feature set in several ways. This section first describes the general language and virtual machine limitations and some consequences of the missing reflection capabilities. It then discusses the simplified security and highlights some general limitations resulting directly from the limited hardware capabilities of CLDC devices.

General Java Language Limitations

For CLDC, the Java language itself was simplified slightly. The following restrictions hold for the Java language in CLDC:

- No floating-point support (CLDC 1.0 only)
- No reflection
- No thread groups and daemon threads
- No weak references (CLDC 1.0 only)
- Error handling may be limited
- No finalization; CLDC does not support the `finalize()` method
- No Java Native Interface (JNI)
- No user-defined class loaders

The missing floating-point support is perhaps the most significant limitation because it makes development of calculation or spreadsheet programs very difficult. The MathFP API from Onno Honnes (www.jscience.net) provides fixed-point calculations as a substitute, but fixed-point arithmetic is not a complete replacement for floating-point support.

CLDC supports full exception handling, but limitations may apply to the Error exception classes. The problem is that it is very difficult to handle errors like those that arise from heap exceptions, which may be resolved only by a soft reset of the whole device. Thus, the device may handle errors differently in a manner appropriate to the device without reporting a corresponding exception to Java.

The restrictions concerning user-defined class loaders and the JNI are addressed in the upcoming section "Simplified Security Model."

Note

Due to a bug in the original CLDC specification, the .class directive (for example, `String.class`) does not work in CLDC 1.0. However, this issue is fixed in CLDC 1.1 by adding the NoClassDefFoundError class, required for the compilation process.

Consequences of the Missing Reflection Support

The reflection capabilities of the CLD configuration are very limited. `Class.forName()` and `newInstance()` are supported, but you can't work with methods, variables, or constructors at the reflection level.

As a consequence, several other APIs are not available and cannot be implemented for CLDC:

- No class loaders. CLDC supports only the built-in class loader. You can't add custom class loaders.
- No Remote Method Invocation (RMI). RMI relies on full reflection capabilities, so RMI is not possible in CLDC.
- No Jini. Jini depends on RMI, so you can't use it with the CLDC.
- No serialization. Serialization depends on reflection, so serialization is not available in CLDC.

For Jini, a solution could be the surrogate architecture, allowing simple devices to be integrated in a Jini environment. For the missing RMI and serialization capabilities, the explicit serialization of the kSOAP API, described in Chapter 10, "Third Party Libraries," can provide a replacement, even if it is limited in several ways.

Simplified Security Model

As in J2SE, the Java byte code is verified by the VM before execution in order to prevent security violations resulting from side effects of illegal byte code. In CLDC, the verification process is slightly different from that used in J2SE. CLDC introduces an additional preverification step that simplifies verification of the byte code on the device. The preverification process and its motivation are described in the next section.

For application-level security, J2SE provides *security managers* for fine-grained access control. Unfortunately, security managers consume too much memory to be included in CLDC devices. For this reason, CLDC provides the simpler *sandbox model* for application security. The sandbox model means that Java applications run in a closed environment where only APIs known to be safe can be accessed.

The sandbox model means that the following additional restrictions apply:

- The Java Native Interface (JNI) is not available, in order to prevent backdoor access to native functionality that is not exposed through the Java APIs provided with the device.

- User-defined class loaders cannot be created, in order to prevent programmers from overriding the class loading mechanism provided by the VM.

For PDAP, an additional security layer allows the user to grant applications access permissions such as network and personal information management access. By default, PDAP applications are not allowed to establish network connections or access information stored in the device address database or calendar. However, such access can be allowed via the application manager.

Off-Device Preverification

The preverification step was discussed in Chapter 1. This additional step is applied to Java class files after they are generated from the corresponding Java source files. The reason for introducing this additional step, which leads to much confusion for developers, was that the original class file verification performed by the JVM was very expensive in terms of memory and computational power. Basically, the preverification step enriches the class file with hints for the on-device verifier. Thus, the final verification can be performed more efficiently.

Please note that preverification does not mean less security. If you think of the verification process as confirming that a way exists through a labyrinth, then the preverification step marks the way. The way can still be verified in the device, and if the way is not valid, verification will detect that; but preverification lets you avoid the much greater effort involved in finding the way.

General Device Hardware Limitations

General hardware limitations of CLDC devices are

- Limited computing capabilities
- Limited memory
- Limited heap space

The computing power of processors used for mobile phones is usually very limited when compared to desktop systems. Also the memory provided by CLDC devices is very limited. Even worse, for many devices, there is a distinction between persistent (flash) memory and the heap space available, and the heap space usually is only a small fraction of the total memory (32–512KB). So the memory available at runtime may be even more restricted than the memory available in the device.

CLDC 1.1

CLDC 1.1, the "next generation" of the CLDC configuration, which is used in the PDA profile, lifts some of the original CLDC limitations. Namely, CLDC-NG no longer explicitly forbids floating point operations, and weak references are added.

CLDC Application Design

The design rules for CLDC applications are quite simple: Keep everything as small and as simple as possible. CLDC applications should be designed to consume as few resources as possible, and the user should be allowed to exit at any time without losing data. Design rules concerning the user interface will be covered in more detail in Chapters 3, "MIDP Programming," and 4, "PDAP Programming."

CLDC APIs

Packages included in CLDC are `java.io`, `java.lang`, `java.util`, and `javax.microedition.io`, where `javax.microedition.io` is mainly a replacement for the missing `java.net` package. Here, we will give only a short overview of the classes available. Please note that most classes do not provide all the methods of their J2SE counterparts. For detailed information, please consult the CLDC API documentation.

In Appendix B, "Comparison Charts," you can find a table with hints for mapping functionality of J2SE classes and methods omitted in CLDC. The following classes are available in CLDC 1.1 only: Float, Double, java.lang.ref.Reference, java.lang.ref.Reference, and `java.lang.ref. WeakReference.`

The `java.lang` Package

Supported classes from `java.lang` are `Object`, the wrapper classes for the built-in data types, `Math`, `Runtime`, `String`, `StringBuffer`, `System`, `Thread`, and `Throwable`.

Depending on the CLDC version `java.lang.Math` might not contain operations for floating point numbers. `java.lang.Class` provides only very limited support for reflection.

The following classes are available in CLDC 1.1 only: `Float`, `Double`, `java.lang.ref. Reference, java.lang.ref.Reference, and java.lang.ref.WeakReference.`

The `java.util` Package

Supported classes from `java.util` are `Calendar`, `Date`, `Hashtable`, `Random`, `Stack`, `TimeZone`, and `Vector`.

The Java 2 collection framework is not supported at all. The `Vector` class contains only the old access methods `elementAt()`, `setElementAt()`, and `addElement()`, instead of the `get(), set(),` and `add()` methods introduced with the Java 2 collection framework. The only time zone required is GMT.

The `java.io` Package

Supported classes from `java.io` are `ByteArrayInputStream`, `ByteArrayOutputStream`, `DataInputStream`, `DataOutputStream`, `InputStream`, `InputStreamReader`, `OutputStream`, `OutputStreamWriter`, `PrintStream`, `Reader`, and `Writer`. As you can infer from the list, all file-related classes are missing. The CLDC does not provide any replacement, but MIDP provides the package `javax.microedition.rms` for persistent storage, and PDAP adds a `FileConnection` interface to the `javax.microedition.io` package.

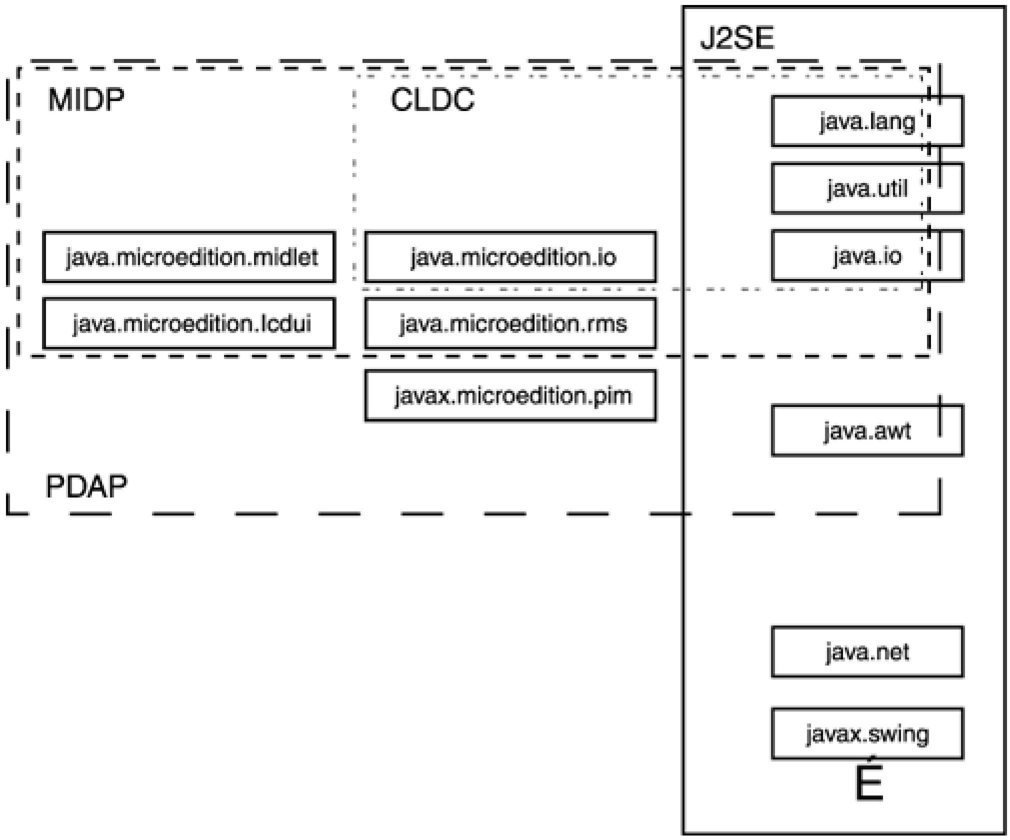
The `javax.microedition.io` Package

The package `javax.microedition.io` is mainly a compact replacement for the `java.net` package, the so-called Generic Connection Framework (described in detail in Chapter 6, "Networking: The Generic Connection Framework"). It provides a set of classes and interfaces that let you establish different kinds of network connections.

CLDC Profiles

CLDC itself provides only a basic set of classes for a J2ME profile. In Chapter 1, you were given an overview of the two profiles available for CLDC: the Mobile Information Device Profile (MIDP) and the PDA Profile (PDAP). Now, we will present the additional functionality the profiles add to CLDC. We will start with MIDP and then give an overview of PDAP. Figure 2.1 shows an overview of the J2ME CLDC packages and their intersections with J2SE.

Figure 2.1. J2ME CLDC packages including the MID and PDA profiles and intersections with J2SE.



MID Profile

As described in Chapter 1, MIDP is designed for mobile information devices such as cellular phones and two-way pagers. MIDP adds several new packages and classes to CLDC. The following sections list the additions per package.

Additions to `java.util`

MIDP adds the classes `Timer` and `TimerTask` in the package `java.lang` to the set of supported classes. `Timer` and `TimerTask` allow simplified scheduling of tasks for a point of time in the future, including repeated tasks.

Additions to `java.lang`

MIDP adds the `IllegalStateException` in the package `java.lang`. The `IllegalStateException` is thrown when a method is called in a state of the application where doing so is not allowed. For example, you can't access the display of a MIDlet before the corresponding `startApp()` method is invoked by the system.

Additions to `javax.microedition.io`

MIDP adds an `HttpConnection` for HTTP connections to the generic connection framework. The `HttpConnection` class is described in detail in Chapter 6.

Package `javax.microedition.midlet`

The `javax.microedition.midlet` package is a completely new package of MIDP. It mainly contains the class `MIDlet`, encapsulating the life cycle of an MIDP application. The `MIDlet` class can be compared to an applet to some extent, except that it is not related to the display. The life cycle of MIDP applications—including the `MIDlet` class—is described in detail in Chapter 3.

Package `javax.microedition.lcdui`

The `javax.microedition.lcdui` package contains the graphical user interface (GUI) classes for MIDP. These classes are not compatible to J2SE and provide only very basic elements adequate to the limited display of a mobile information device. The MIDP GUI classes are described in detail in Chapter 3, together with the MIDP application life cycle.

Package `javax.microedition.rms`

MIDP does not contain a file system, but instead uses a record management API for persistent storage. The record system is more adequate to the persistent memory of mobile devices, where data is usually stored persistently in random access memory instead of sequential files. The record management system is described in detail in Chapter 5, "Data Persistency."

PDA Profile

Like the MID Profile, the PDA Profile makes many additions to the CLD Configuration required for the targeted class of devices. Because PDAP is a superset of MIDP, all MIDP packages and CLDC additions are available in PDAP. Please note that PDAP 1.0 is based on CLDC 1.1, in contrast to MIDP 1.0, which is based on CLDC 1.0.

In addition to the MIDP additions to CLDC, PDAP provides a more sophisticated user interface based on a subset of the Abstract Window Toolkit (AWT) and access to the device address and calendar databases.

Package `java.awt` and Subpackages

In contrast to MIDP, PDAP does not provide a user interface API designed from scratch. The PDAP user interface is a subset of the J2SE AWT classes.

Package `javax.microedition.pim`

The personal information management classes provide access to the built-in calendar and address book databases of the PDA. The `pim` classes are described in detail in Chapter 7, "PIM: Accessing the Personal Information Manager."

Additions to `javax.microedition.io`

PDAP adds a `CommConnection` and a `FileConnection` for serial port and file access to the generic connection framework, which are described in detail in Chapter 6.

Java Application Deployment

For desktop computers, there are different ways to install an application. In order to install new software properly, the user needs to insert the medium containing the software into the computer system and start the installation. It is common practice for a wizard to guide the user through a predefined procedure for installing the software. Software or software updates can be downloaded from the Internet using a Web browser as well. When the software is downloaded, the installation usually needs to be initiated by the user or system administrator.

For limited devices, software installation is different. PDA software is usually installed through a connection to the desktop computer; for example, a serial cable or an Infrared Data Association (IrDA) connection. The installation is initiated on the desktop computer.

Using devices that provide a wireless Internet connection, it seems quite straightforward to download applications from the Internet directly to the device without going through a desktop PC.

A downloaded application must be saved in the device's storage, installed, and inspected by the platform. Applications of that kind can be launched and later deleted from the device when the user no longer needs it. A mechanism covering these issues is called Java Application Manager (JAM).

JAM Implementation

The JAM reference implementation is generally described in the CLDC reference implementation (RI) by SUN. In the RI documentation, the JAM is divided into the following steps:

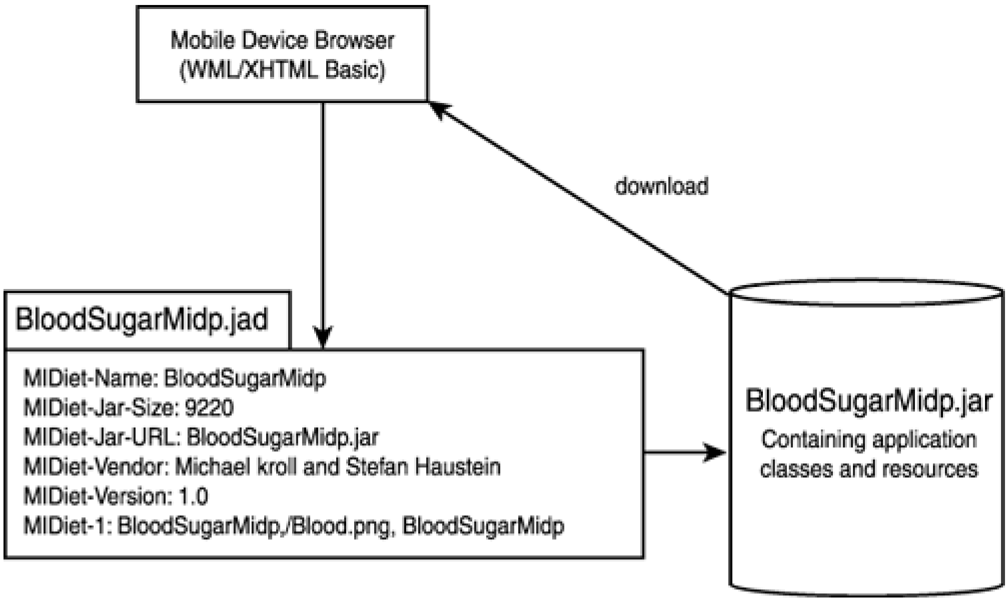
- Installing the application to the device
- Executing the application
- Updating the installed application
- Deleting the application from the target device

In order to use the JAM mechanism, the KVM version that is used should support this feature. The RI does not state how the implementation platform has to support browsing of descriptor files in the Internet, for example, but describes all the information that needs to be included in a descriptor file. A complete specification of how JAM should be implemented is given in the J2ME profiles.

The Descriptor File

The Java Application Descriptor (JAD) file is downloaded into the device and analyzed by the platform. According to the information that is stored in the descriptor file, the platform decides whether the related JAR file containing the application classes should be downloaded or rejected. The association between the descriptor file and the JAR file is shown in Figure 2.2.

Figure 2.2. The association between JAD and JAR file including a KVM application.



A JAD descriptor file consists of readable text including name-value pairs describing properties of its associated Java application. Each line of the descriptor file holds one attribute, where an attribute consists of a name and a value, separated by a colon. The following attribute describes the file size of the application JAR file:

```
Jar-File-Size: 2123
```

The application developer is responsible for maintaining the JAD and JAR files. Additionally, the descriptor file and the associated JAR file must be placed on the same Web site.

JAM on MIDP

Whereas the CLDC JAM specification is relatively vague, MIDP contains a concrete definition of the JAD files for MIDP applications. This section first takes a closer look at the contents of a JAD file, and then describes the actual transfer of an MIDP application to the target device. However, before we go into details of the JAD files, we need to explain the MIDP concept of MIDlet suites.

MIDlet Suites

A *MIDlet suite* is a set of Java applications distributed together. A group of MIDlets inside a MIDlet suite can share their persistent databases. For example, a database might consist of user logins that are used by all network-related MIDlets in one suite.

Since MIDP devices offer only a limited amount of heap space for the Java Runtime Environment, it may make sense to split application functionality into two or more MIDlets. For instance, one MIDlet of a suite could be responsible for providing a user interface for entering application data. A second MIDlet in the same suite could be responsible for synchronizing the data with a remote server over the Internet. Both MIDlets would access the same data stored in a persistent database accessible from all MIDlets of the suite.

MIDP JAD Files

JAD files have already been defined in general CLDC terms. MIDP extends the specification with respect to the special requirements of mobile information devices. The JAD file in MIDP consists of mandatory and optional attributes. The application provider must fill the mandatory attributes that are listed in Table 2.1. The syntax is the same as described in the section "Java Application Deployment." Table 2.2 shows additional optional entries.

Table 2.1. Mandatory JAD File Attributes in MIDP

Attribute Name	*Description*
`MIDlet-Name`	The name of the MIDlet suite.
`MIDlet-Version`	The version number of the MIDlet suite.
`MIDlet-Vendor`	The vendor of the given MIDlet suite.
`MIDlet-Jar-URL`	The URL from which the JAR file can be downloaded.
`MIDlet-Jar-Size`	The size of the JAR file in bytes.
`MicroEdition-Profile`	The J2ME profile that is required to run the MIDlet.
`MicroEdition-Configuration`	The J2ME configuration that is needed to run the MIDlet.
`MIDlet-<Number>`	For each midlet contained in a midlet suit, a separate `MIDlet-<number>` entry is required, where `<number>` must be replaced by a number ranging from 1 to the number of MIDlets contained in the suite. The value of the `MIDlet-<Number>` attribute consists of three parameters: `MIDlet-<Number>: <name>, <icon>, <class>` `<Number>` is an enumeration of the MIDlets in the JAR file. The first MIDlet is `MIDlet-1`. If more MIDlets are stored in the JAR file, the entries are named `MIDlet-2`, `MIDlet-3`, and so on. The parameter `<name>` describes the name of the MIDlet that is displayed on the device after the descriptor file is loaded. The `<icon>` parameter specifies an icon that can be displayed in the list. The last parameter, `<class>`, specifies the name of the actual subclass of MIDlet implemented by the application.

Table 2.2. Optional JAD File Attributes in MIDP

Attribute Name	*Description*
`MIDlet-Description`	The description of the MIDlet suite.
`MIDlet-Icon`	The name of the PNG file representing the MIDlet suite that is contained in the JAR file.
`MIDlet-Info-URL`	The URL providing further information about the MIDlet suite.
`MIDlet-Data-Size`	The minimum number of bytes of persistent data that is needed to run the MIDlet.

Specialized development tools for J2ME applications might provide support for setting the JAD file properties. For example, the SUN Wireless Toolkit provides a Settings button where the JAD options can be entered in a dialog box. Moreover, some options such as the file size of the JAR file are filled in automatically.

MIDP JAR Manifest Entries

The JAR manifest file of a MIDlet suite must contain the same attributes as the JAD file, except from the JAR URL and JAR size attributes. Note that some devices access the JAD file only for transmission, but do not store the information contained in it. Thus, application-specific attributes should be duplicated in both files for safety.

Over the Air User Initiated Provisioning for MIDP

Although the JAD file format is specified completely in the MIDP specification, questions concerning the concrete details of downloading and discovering MIDlets are left open by the specification. For that reason, the three specification leads have released a document titled "Over the Air User Initiated Provisioning Best Practice" (OTA), covering the HTTP transfer steps and application installation in detail. This can be downloaded using the following URL: http://java.sun.com/products/midp/OTAProvisioning-1.0.pdf

For MIDP developers, it is important to know that the installation is initiated when the user selects the link to a file of the MIME type `text/vnd.sun.j2me.app-descriptor` or with the suffix `.jad`.

The OTA document further specifies the recommended behavior and user interaction of the application manager. It also specifies how the device identifies itself in additional HTTP header fields. However, these technical details of the application deployment protocol are mainly important for implementers of the device application manager software and specialized Internet servers. If you are interested, the whole OTA document is available for download on the SUN Internet pages. For convenience, it is directly linked from the Web page of this book.

JAM for PDAP

Because PDAP is a superset of MIDP, a MIDP JAR file is also a valid PDAP JAR file. However, PDAP provides one extension of the JAR file format. PDA applications using PDA capabilities beyond MIDP must use `PDAlet-`<*Number*> entries. The syntax of the `PDAlet-`<*Number*> entries is identical to the `MIDlet-`<*Number*> entries. A PDAlet entry with the same number as a MIDlet entry overrides the corresponding MIDlet entry. Thus, different versions of the same applications for both profiles can be included in the same JAD and JAR files. If the device supports PDAP, the PDA applications will be loaded; otherwise the application manager automatically falls back to the MIDlet entry because PDAlet entries have no meaning for the MIDP application manager. The `HelloPdap` example (Listing 1.5) in the first chapter shows a corresponding dual-profile JAD file.

Summary

In this chapter, you have learned the limitations of the CLD configuration. You have had an overview of the CLDC API packages and the PDAP and MIDP additions. You also know about the Java Application Management mechanism for deploying J2ME applications and the JAD entries for PDAP and MIDP applications.

The next chapters will first revisit the MIDP and PDAP "Hello World" applications from an API point of view and explain the application lifecycle. Then the user interfaces of MID and PDA applications are explained in depth.

Chapter 3. MIDP Programming

IN THIS CHAPTER

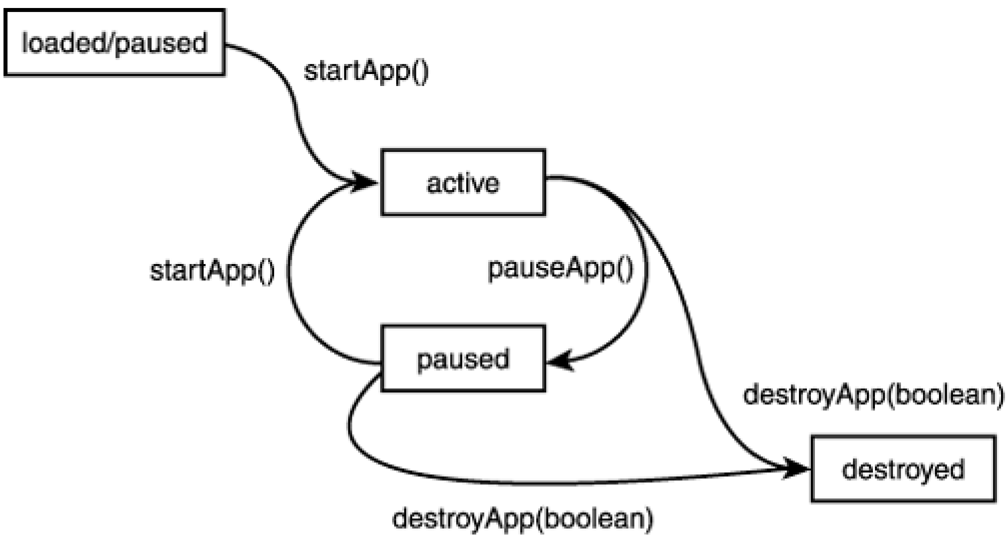
This chapter handles the life cycle and user interface of Mobile Information Device Profile (MIDP) applications. First, the general design of MIDP applications will be discussed. Then, the high-level user interface API will be explained. Finally, the low-level user interface API for free graphics and games will be described.

MIDlets

All applications for the MID Profile must be derived from a special class, `MIDlet`. The `MIDlet` class manages the life cycle of the application. It is located in the package `javax.microedition.midlet`.

MIDlets can be compared to J2SE applets, except that their state is more independent from the display state. A MIDlet can exist in four different states: loaded, active, paused, and destroyed. Figure 3.1 gives an overview of the MIDlet lifecycle. When a MIDlet is loaded into the device and the constructor is called, it is in the loaded state. This can happen at any time before the program manager starts the application by calling the `startApp`() method. After `startApp()` is called, the MIDlet is in the active state until the program manager calls `pauseApp()` or `destroyApp()`; `pauseApp`() pauses the MIDlet, and `desroyApp`() terminates the MIDlet. All state change callback methods should terminate quickly, because the state is not changed completely before the method returns.

Figure 3.1. The life cycle of a MIDlet.



In the `pauseApp()` method, applications should stop animations and release resources that are not needed while the application is paused. This behavior avoids resource conflicts with the application running in the foreground and unnecessary battery consumption. The `destroyApp()` method provides an unconditional parameter; if it is set to false, the MIDlet is allowed to refuse its termination by throwing a `MIDletStateChangeException`. MIDlets can request to resume activity by calling `resumeRequest()`. If a MIDlet decides to go to the paused state, it should notify the application manager by calling `notifyPaused()`. In order to terminate, a MIDlet can call `notifyDestroyed()`. Note that `System.exit()` is not supported in MIDP and will throw an exception instead of terminating the application.

Note

Some devices might terminate a MIDlet under some circumstances without calling `destroyApp`(), for example on incoming phone calls or when the batteries are exhausted. Thus, it might be dangerous to rely on `destroyApp()` for saving data entered or modified by the user.

Display and Displayable

MIDlets can be pure background applications or applications interacting with the user. Interactive applications can get access to the display by obtaining an instance of the `Display` class. A MIDlet can get its `Display` instance by calling `Display.getDisplay(MIDlet midlet)`, where the MIDlet itself is given as parameter.

The `Display` class and all other user interface classes of MIDP are located in the package `javax.microedition.lcdui`. The `Display` class provides a `setCurrent()` method that sets the current display content of the MIDlet. The actual device screen is not required to reflect the MIDlet display immediately—the `setCurrent()` method just influences the internal state of the MIDlet display and notifies the application manager that the MIDlet would like to have the given `Displayable` object displayed. The difference between `Display` and `Displayable` is that the `Display` class represents the display hardware, whereas `Displayable` is something that can be shown on the display. The MIDlet can call the `isShown()` method of `Displayable` in order to determine whether the content is really shown on the screen.

HelloMidp Revisited

The `HelloMidp` example from Chapter 1, "Java 2 Micro Edition Overview," is already a complete MIDlet. Now that you have the necessary foundation, you can revisit `HelloMidp` from an API point of view.

First, you import the necessary `midlet` and `lcdui` packages:

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

Like all MIDP applications, the `HelloMidp` example is required to extend the `MIDlet` class:

```
public class HelloMidp extends MIDlet {
```

In the constructor, you obtain the `Display` and create a `Form`:

```
Display display;
Form mainForm;

public HelloMidp() {
```

```
   mainForm = new Form ("HelloMidp");
}
```

A Form is a specialized Displayable class. The Form has a title that is given in the constructor. You do not add content to the form yet, so only the title will be displayed. (A detailed description of the Form class is contained in the next section.)

When your MIDlet is started the first time, or when the MIDlet resumes from a paused state, the startApp() method is called by the program manager. Here, you set the display to your form, thus requesting the form to be displayed:

```
public void startApp() {
    display = Displayable.getDisplay (this);
    display.setCurrent (mainForm);
}
```

When the application is paused, you do nothing because you do not have any allocated resources to free. However, you need to provide an empty implementation because implementation of pauseApp() is mandatory:

```
public void pauseApp() {
}
```

Like pauseApp(), implementation of destroyApp() is mandatory. Again, you don't need to do anything here for this simple application:

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

Note

The HelloMidp Midlet does not provide a command to exit the MIDlet, assuming that the device provides a general method of terminating MIDlets. For real MIDP applications, we recommend that you add a command to terminate the MIDlet because the MIDP specification does not explicitly support this assumption. More information about commands can be found in the section "Using Commands for User Interaction."

MIDP User Interface APIs

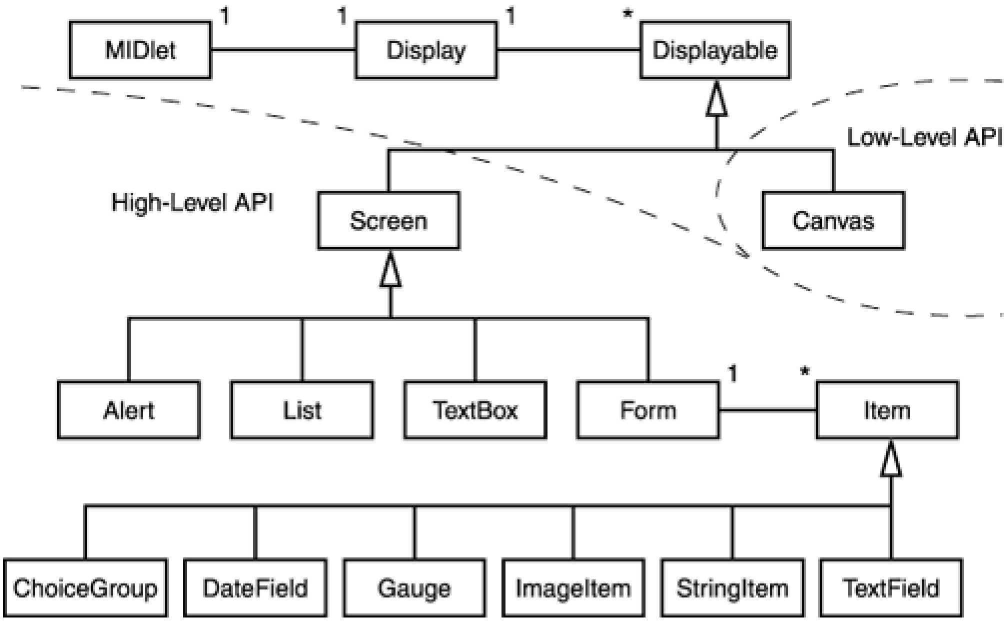
The MIDP user interface API is divided into a high- and low-level API. The high-level API provides input elements such as text fields, choices, and gauges. In contrast to the Abstract Window Toolkit (AWT), the high-level components cannot be positioned or nested freely. There are only two fixed levels: Screens and Items. The Items can be placed in a Form, which is a specialized Screen.

The high-level Screens and the low-level class Canvas have the common base class Displayable. All subclasses of Displayable fill the whole screen of the device. Subclasses of Displayable can be shown on the device using the setCurrent() method of the Display object. The display hardware of a MIDlet can be accessed by calling the static method getDisplay(), where the MIDlet itself is given as parameter. In the HelloMidp example, this step is performed in the following two lines:

```
Display display = Display.getDisplay (this);
...
display.setCurrent (mainForm);
```

Figure 3.2 shows an overview of the MIDP GUI classes and their inheritance structure.

Figure 3.2. The MIDP GUI classes.



The following sections first describe the high-level API and then the low-level API. A more complex sample application that uses both levels of the `lcdui` package together is shown in Chapter 9, "Advanced Application: Blood Sugar Log."

High-Level API

Now that you know the basics of the MIDlet's life cycle and general display model, we can start to look deeper into the `lcdui` package. We will start with another subclass of `Screen`: `Alert`. Then we will discuss some simple `Item`s like `StringItem` and `ImageItem`. We will explain the use of more advanced `Item`s such as `TextField` and `ChoiceGroup` by creating a simple `TeleTransfer` example application. As we introduce new MIDP high-level UI capabilities like other `Screen` subclasses, we will extend the `TeleTransfer` sample step by step.

Alerts

You already know the `Form` class from the first example. The simplest subclass of `Screen` is `Alert`. `Alert` provides a mechanism to show a dialog for a limited period of time. It consists of a label, text, and an optional `Image`. Furthermore, it is possible to set a period of time the `Alert` will be displayed before another `Screen` is shown. Alternatively, an `Alert` can be shown until the user confirms it. If the `Alert` does not fit on the screen and scrolling is necessary to view it entire contents, the time limit is disabled automatically.

The following code snippet creates an `Alert` with the title "HelloAlert" and displays it until it is confirmed by the user:

```
Alert alert = new Alert ("HelloAlert");
alert.setTimeout (Alert.FOREVER);
display.setCurrent (alert);
```

Forms and Items

The most important subclass of Screen is the class Form. A Form can hold any number of Items such as StringItems, TextFields, and ChoiceGroups. Items can be added to the Form using the append() method.

The Item class itself is an abstract base class that cannot be instantiated. It provides a label that is a common property of all subclasses. The label can be set and queried using the setLabel()and getLabel() methods, respectively. The label is optional, and a null value indicates that the item does not have a label. However, several widgets switch to separate screens for user interaction, where the label is used as the title of the screen. In order to allow the user to keep track of the program state, it is recommended that you provide a label at least for interactive items.

Items can neither be placed freely nor can their size be set explicitly. Unfortunately, it is not possible to implement Item subclasses with a custom appearance. The Form handles layout and scrolling automatically. Table 3.1 provides an overview of all Items available in MIDP.

Table 3.1. All Subclasses of Item

Item	*Description*
ChoiceGroup	Enables the selection of elements in group.
DateField	Used for editing date and time information.
Gauge	Displays a bar graph for integer values.
ImageItem	Used to control the layout of an Image.
StringItem	Used for read-only text elements.
TextField	Holds a single-line input field.

StringItem

StringItems are simple read-only text elements that are initialized with the label and a text String parameter only. The following code snippet shows the creation of a simple version label. After creation, the label is added to the main form in the constructor of the HelloMidp application:

```
public HelloMidp() {
    mainForm = new Form ("HelloMidp");
    StringItem versionItem = new StringItem ("Version: ", "1.0");
    mainForm.append (versionItem);
}
```

The label of the StringItem can be accessed using the setLabel() and getLabel() methods inherited from Item. To access the text, you can use the methods setText() and getText().

ImageItem

Similar to the StringItem, the ImageItem is a plain non-interactive Item. In addition to the label, the ImageItem constructor takes an Image object, a layout parameter, and an alternative text string that is displayed when the device is not able to display the image. The image given to the constructor must be non-mutable. All images loaded from the MIDlet suite's JAR file are not mutable.

The difference between mutable and non-mutable Images is described in more detail in the section about Images in this chapter. For now, we will treat the Image class as a "black box" that has a string constructor that denotes the location of the image in the JAR file. Please note that Image construction from a JAR file throws an IOException if the image cannot be loaded for some reason. The layout parameter is one of the integer constants listed in Table 3.2, where the newline constants can be combined with the horizontal alignment constants.

Table 3.2. ImageItem Layout Constants	
Constant	***Value***
LAYOUT_CENTER	The image is centered horizontally.
LAYOUT_DEFAULT	A device-dependent default formatting is applied to the image.
LAYOUT_LEFT	The image is left-aligned.
LAYOUT_NEWLINE_AFTER	A new line will be started after the image is drawn.
LAYOUT_NEWLINE_BEFORE	A new line will be started before the image is drawn.
LAYOUT_RIGHT	The image is aligned to the right.

The following code snippet shows how a center aligned `ImageItem` is added to the `HelloMidp` sample MIDlet:

```
public HelloMidp() {
    display = Display.getDisplay (this);
    mainForm = new Form ("HelloMidp");
    try {
        ImageItem logo = new ImageItem
            ("Copyright: ", Image.createImage ("/mcp.png"),
             ImageItem.LAYOUT_CENTER | mageItem.LAYOUT_NEWLINE_BEFORE
             | ImageItem.LAYOUT_NEWLINE_AFTER, "Macmillian USA");

          mainForm.append (logo);
    } catch (IOException e) {
        mainForm.append (new StringItem ("Copyright", "Sams
Publishing; Image not available:" + e));
    }
}
```

By forcing a new line before and after the image, you ensure that the image is centered in its own line. Figure 3.3 shows the corresponding display on the device. If the image cannot be loaded and an exception is thrown, a simple `StringItem` is appended to the form instead of the image.

Figure 3.3. The `HelloMidp` application showing an `ImageItem`.

Handling Textual Input in `TextFields`

As shown in Table 3.1, textual input is handled by the class `TextField`. The constructor of `TextField` takes four values: a label, initial text, a maximum text size, and constraints that indicate the type of input allowed. In addition to avoiding input of illegal characters, the constraints may also influence the keyboard mode. Several MIDP devices have a numeric keyboard only, and the constraints allow the application manager to switch the key assignments accordingly. The constants listed in Table 3.3, declared in the class `TextField`, are valid constraint values.

Table 3.3. TextField Constraint Constant Values	
Constant	***Value***

ANY	Allows any text to be added.
EMAILADDR	Adds a valid e-mail address, for instance myemail@mydomain.com.
NUMERIC	Allows integer values.
PASSWORD	Lets the user enter a password, where the entered text is masked.
PHONENUMBER	Lets the user enter a phone number.
URL	Allows a valid URL.

We will now show the usage of `TextField`s by creating a simple example `Form` for bank transfers. A bank transfer form contains at least the amount of money to be transferred and the name of the receiver.

To start the implementation of the `TeleTransfer` MIDlet, you first need to import the two packages containing the `midlet` and `lcdui` classes:

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

Every MID application is derived from `MIDlet`, so you need to extend the `MIDlet` class, too:

```
public class TeleTransfer extends MIDlet {
```

Because you want to create a `Form` that contains `Item`s for entering the transfer information, you need a corresponding member variable `mainForm`. You can already initialize the variable at its declaration because it has no dependencies from constructor parameters:

```
Form mainForm = new Form ("TeleTransfer");
```

In order to let the user enter the transfer information, add `TextField`s for the name of the receiver for entering the amount to be transferred. Because of the lack of floating-point values in the CLDC, the numeric `TextField`s in MIDP can hold integer values only. So you need to split the amount into separate fields for dollars and cents. An alternative would be to use an alphanumeric field and parse the string into two separate values. However, this may result in switching the keyboard to alpha mode on cell phones, making numeric input unnecessarily complicated. In this case, you'll limit the size of possible values to five digits for the whole dollar part and two digits for the fractional cent part. Again, you initialize the variables where they are declared:

```
TextField receiverName = new TextField
    ("Receiver Name", "", 20, TextField.ANY);
TextField receiverAccount = new TextField
    ("Receiver Account#", "", 12, TextField.NUMERIC);
TextField amountWhole = new TextField ("Dollar", "", 6,
TextField.NUMERIC);
TextField amountFraction = new TextField ("Cent", "", 2,
TextField.NUMERIC);
```

Finally, you add a variable storing the `Display` instance for your application:

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

Now you can add the constructor to your application where you added the previous `TextField`s to the main form:

```
public TeleTransfer() {
    mainForm.append (receiverName);
    mainForm.append (receiverAccount);
```

```
    mainForm.append (amountWhole);
    mainForm.append (amountFraction);
}
```

When the application is started, you request the display to show your money transfer form by calling `setCurrent()`. As explained in the "MIDlets" section, the application manager notifies you about the application start by calling the `startApp()` method. So you implement this method accordingly:

```
public void startApp() {
    display.setCurrent (mainForm);
}
```

Please note that `startApp()` is called also when the MIDlet resumes from the paused state, so you cannot move the initialization code from the constructor to this method.

Both `pauseApp()` and `destroyApp()` are declared as abstract in the `MIDlet` class, so you need to implement these methods in your application, even if you do not have real content for them. You just provide empty implementations, like in the `HelloMidp` example in the first section:

```
public void pauseApp() {
}

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

Selecting Elements Using ChoiceGroups

In the previous section, you created a simple to enter information for transferring money between two accounts. Now you will extend the application to allow the user to select different currencies. For this purpose, you will now add a `ChoiceGroup` to your application.

The `ChoiceGroup` is an MIDP UI widget enabling the user to choose between different elements in a `Form`. These elements consist of simple `String`s, but can display an optional image per element as well. `ChoiceGroup`s can be of two different types. Corresponding type constants are defined in the `Choice` interface. These constants are used in the `List` class as well; the `List` class allows an additional third type. The three type constants are listed in Table 3.4.

Table 3.4. Choice Type Constants

Constant	*Value*
`EXCLUSIVE`	Specifies a `ChoiceGroup` or `List` having only one element selected at the same time.
`IMPLICIT`	Valid for `List`s only. It lets the `List` send `Command`s to indicate state changes.
`MULTIPLE`	In contrast to `EXPLICIT`, `MULTIPLE` allows the selection of multiple elements.

The `ChoiceGroup` constructor requires at least a label and a type value. Additionally, a `String` array and an `Image` array containing the elements can be passed to the constructor. Elements can also be added dynamically using the `append()` method. The `append()` method has two parameters, a `String` for the label and an `Image`. In both cases, the image parameter may be `null` if no images are desired.

In order to add a `ChoiceGroup` to the `TeleTransfer` application, you introduce a new variable `currency` of type `ChoiceGroup`. By setting the type to `EXCLUSIVE`, you get a `ChoiceGroup` where only one item can be selected at a time. You directly add elements for the United States (USD), the European Union (EUR), and Japan (JPY) by passing a `String` array created inline. The `ChoiceGroup` enables the user to choose between three currencies that are represented textually by the abbreviations specified in the `String` array. The last parameter of the constructor is set to `null` because you do not want `Image`s to be displayed at this time:

```
ChoiceGroup currency = new ChoiceGroup
    ("Currency", Choice.EXCLUSIVE,
new String[] {"USD", "EUR", "JPY"} , null);
```

You still need to add the `currency` `ChoiceGroup` to your main `Form`. As for the text fields, this is done via the `append()` method of `Form`:

```
mainForm.append (currency);
```

Figure 3.4 shows the `TeleTransfer` application extended to choose a currency using a `ChoiceGroup`.

Figure 3.4. The `TeleTransfer` MIDlet extended to enable the user to choose a currency.

Receiving Changes from Interactive UI Items

If you run the new version of the `TeleTransfer` MIDlet, you can change the currency using the `ChoiceGroup`, but the `TextField` labels for Dollar and Cent are not changed accordingly. You need a way to notify the application if a selection is made in the currency `ChoiceGroup`.

Receiving changes of interactive high-level UI items in MIDP is based on a listener model similar to AWT. Classes implementing the `ItemStateListener` interface are able to receive notifications for the following events:

- State changes of a `ChoiceGroup`
- Value adjustments of an interactive `Gauge`
- `TextField` value changes
- `DateField` changes

The events are sent to the method `itemStateChanged()` of the `ItemStateListener`, where the item that has changed is given as a parameter. In order to actually receive these events, the `ItemStateChangeListener` must be registered using the `setItemStateListener()` method of the corresponding `Form`.

Now that you know about item state change events, you can add the desired functionality to your `TeleTransfer` MIDlet. First, you need to add the `ItemStateListener` interface to the class declaration:

```
public class TeleTransfer extends MIDlet implements ItemStateListener
{
```

You also need to implement a corresponding `itemStateChanged()` method. Since the `itemStateChanged()` method is called for changes of all `Items` in the `Form`, you need to check the item parameter indicating the event source first. If the source of the event is the currency `ChoiceGroup`, you set the labels of the amount and fraction `TextFields` correspondingly:

```
public void itemStateChanged (Item item) {
    if (item == currency) {
        int index = currency.getSelectedIndex();
        switch (index) {
        case 0: amountWhole.setLabel ("Dollar");
                 amountFraction.setLabel ("Cent");
                 break;
        case 1: amountWhole.setLabel ("Euro");
                amountFraction.setLabel ("Cent");
                 break;
        case 2: amountWhole.setLabel ("Yen");
                amountFraction.setLabel ("Sen");
            }
        }
    }
```

Just adding the interface and implementing the corresponding methods is not sufficient to enable the MIDlet to receive state changes. Additionally, you need to register your `ItemStateListener` at the `Form` containing the currency item. You do so by calling the `setItemStateListener()` method in the `TeleTransfer` constructor:

```
public TeleTransfer() {
    mainForm.append (senderAccount);
    ...
    mainForm.append (currency);
    mainForm.setItemStateListener(this);
}
```

Figure 3.5 shows the new version of the `TeleTransfer` example, where the labels are changed depending on the state of the currency `ChoiceGroup`.

Figure 3.5. The `TeleTransfer` MIDlet extended to change the labels depending on the state of the currency `ChoiceGroup`.

Using Commands for User Interaction

Now you can enter all the information required for a telegraphic transfer, but you have no means to initiate the actual transfer.

In contrast to desktop computers, which have plenty of screen space for displaying buttons or menus, a different approach is necessary for mobile devices. Some devices provide so-called *soft buttons*, which are buttons without fixed functionality that are assigned dynamically depending on the application context. The number of soft buttons may vary if they are available. Other mobile devices do not even have space for soft buttons, but provide scrolling menus. MIDP needs to abstract from the concrete device and to provide a mechanism that is suitable for all devices, independent of the availability and number of soft buttons. Thus, the `lcdui` package does not provide buttons or menus, but an abstraction called `Command`.

`Command`s can be added to all classes derived from the `Displayable` class. These classes are `Screen` and its subclasses such as `Form`, `List`, and `TextBox` for the high-level API and `Canvas` for the low-level API.

No positioning or layout information is passed to the `Command`—the `Displayable` class itself is completely responsible for arranging the visible components corresponding to `Command`s on a concrete device. The only layout and display information that can be assigned to a `Command` except from the command label is semantic information. The semantic information consists of a type and a priority. The priority allows the device to decide which commands are displayed as soft buttons if the number of commands exceeds the number of soft buttons available. For additional commands not displayed as soft buttons, a separate menu is created automatically. The type information is an additional hint for the device about how to display the command. For example, if the Exit command is always assigned to the leftmost soft button in native applications of a certain device type, the MIDP implementation is able to make the same assignment. Thus, a consistent look and feel can be accomplished for a device.

The available command type constants are listed in Table 3.5.

Table 3.5. Command Type Constants

Constant	***Value***
`Command.BACK`	Used for navigation commands that are used to return the user to the previous `Screen`.
`Command.CANCEL`	Needed to notify the screen that a negative answer occurred.
`Command.EXIT`	Used to specify a `Command` for exiting the application.
`Command.HELP`	Passed when the application requests a help screen.
`Command.ITEM`	A command type to tell the application that it is appended to an explicit item on the screen.
`Command.OK`	Needed to notify the screen that a positive answer occurred.
`Command.SCREEN`	A type that specifies a screen-specific `Command` of the application.
`Command.STOP`	Interrupts a procedure that is currently running.

The `Command` constructor takes the label, the command type and the priority as input. The `Command` class provides `read()` methods for all these fields, but it is not possible to change the parameters after creation. Using the `addCommand()` method, commands can be added to a `Form` or any other subclass of `Displayable`.

As in the case of receiving state changes of UI widgets, the MIDP uses a listener model for detecting command actions. For this purpose, the `lcdui` package contains the interface `CommandListener`. A `CommandListener` can be registered to any `Displayable` class using the `setCommandListener` method. After registration, the method `commandAction()` of the `Commandlistener` is invoked whenever the user issues a `Command`. In contrast to AWT, only one listener is allowed for each `Displayable` class. The `commandAction()` callback method provides the `Displayable` class where the command was issued and the corresponding `Command` object as parameters.

With this information, you can extend your `TeleTransfer` application with the desired `Command`s. But before going into actual command implementation, you need to add some corresponding functionality. You'll add three commands: a Send command, a Clear command, and an Exit command. For Clear, you just add a method setting the content of the fields of your form to empty strings:

```
public void clear() {
  receiverName.setString ("");
  receiverAccount.setString ("");
  amountWhole.setString ("");
  amountFraction.setString ("");
}
```

The Send command is a bit more difficult since you do not yet have the background to really submit information over the network. (Network connections will be handled in Chapter 6, "Networking: The Generic Connection Framework.") So you just display the content to be transmitted in an alert screen as a temporary replacement:

```
public void send() {
    Alert alert = new Alert ("Send");
    alert.setString ("transfer " + amountWhole.getString()
                     + "." + amountFraction.getString() + " "
                     + amountWhole.getLabel()
                     + "\nto Acc#" + receiverAccount.getString()
                     + "\nof " + receiverName.getString());
    alert.setTimeout (2000);
    display.setCurrent (alert);
    clear();
}
```

For leaving the application, the MIDlet already provides the `notifyDestroyed()` method, so you do not need to add anything here.

Now that you have implemented the corresponding functionality, the next step is to add the actual `Command` objects to your application class:

```
static final Command sendCommand = new Command ("Send",
Command.SCREEN, 1);
static final Command clearCommand = new Command ("Clear",
Command.SCREEN, 2);
static final Command exitCommand = new Command ("Exit", Command.EXIT,
2);
```

In order to enable the MIDlet to receive command actions, you need to implement the `CommandListener` interface, and the corresponding `commandAction()` method. Depending on the command received, you call `send()`, `clear()`, or `notifyDestroyed()`:

```
public class TeleTransfer extends MIDlet
    implements ItemStateListener, CommandListener {

public void commandAction (Command c, Displayable d) {
    if (c == exitCommand) {
        notifyDestroyed();
    }
    else if (c == sendCommand) {
        send();
    }
    else if (c == clearCommand) {
        clear();
    }
}
```

With these modifications, your `TeleTransfer` MIDlet is able to handle the desired commands. You still need to add the `Command`s to the `Form`, and register the `TeleTransfer` MIDlet as a `CommandListener` in order to actually receive the commands:

```
public TeleTransfer() {
    ...
    mainForm.addCommand (sendCommand);
    mainForm.addCommand (clearCommand);
    mainForm.addCommand (exitCommand);
    mainForm.setCommandListener(this);
}
```

Figure 3.6 shows the Send `Alert` of the new version of your `TeleTransfer` application.

Figure 3.6. The `TeleTransfer` MIDlet showing an alert that displays the transfer information as a summary before sending.

Further Item Classes: Gauge and DateField

Now you have used all the `Item` subclasses except `Gauge` and `DateField`. Both classes are specialized input elements, where the `Gauge` may also make sense as a pure read-only information item.

The `Gauge` item visualizes an integer value by displaying a horizontal bar. It is initialized with a label, a flag indicating whether it is interactive, and a maximum and an initial value. If a `Gauge` is interactive, the user is allowed to change the value using a device-dependent input method. Changes to the gauge value will cause `ItemEvents` if a corresponding listener is registered to the form.

The following code snippet shows the construction of a non-interactive `Gauge` labeled Progress that is initialized with a value of 0 and a maximum of 100:

```
Gauge gauge = new Gauge ("Progress", false, 0, 100);
```

If a `Gauge` is used to display progress of a process that takes a longer amount of time, you should also add a corresponding Stop command to the form to abort the progress.

The current value of the `Gauge` can be set using the method `setValue()` and read using the method `getValue()`. Analogous `setMaxValue()` and `getMaxValue()` methods let you access the maximum value of the `Gauge`.

The `DateField` is a specialized widget for entering date and time information in a simple way. It can be used to enter a date, a time, or both types of information at once. The appearance of the `DateField` is specified using three possible input mode constants in the constructor. Possible `DateField` mode constants are listed in Table 3.6.

Table 3.6. `DateField` Mode Constants

Constant	Value
DATE	Passed if the DateField should be used for entering a date only.
DATE_TIME	Used for creating a DateField to enter both date and time information.
TIME	Used to enter time information only.

The DateField has two constructors in which a label and the mode can be specified. Using the second constructor, an additional TimeZone can be passed. The following code snippet shows how a DateField for entering the date of birth can be initialized:

```
DateField dateOfBirth = new DateField ("Date of birth:",
DateField.DATE);
```

After you enter the date into the DateField, it can be accessed using the getDate() method. The DateField offers some additional methods for getting information about the input mode and methods for setting the date and the input mode as well. The concrete usage of the DateField is shown in Chapter 9 in the Blood Sugar Log application.

Further Screen Classes: List and TextBox

The current version of the TeleTransfer MIDlet shows how to use the Form and the corresponding items available in the lcdui package. The application consists of one main form that holds all application widgets. However, your main form is rather long now, so the question arises how to improve the usability of the application. This section shows how to structure the user interface by using multiple screens and introduces the List and TextBox classes.

The List Class

One possibility to clean up the user interface is to move the currency selection to a separate screen. It takes a lot of space and may need even more room if additional options are added. Also, you can assume that the currency is not changed very often.

You could create a new Form and just move the ChoiceGroup there. However, lcdui provides a special List class inherited from Screen for this purpose. The advantage of the List class is that it provides the IMPLICIT mode that was already mentioned in the section "Selecting Elements Using ChoiceGroups." Using the IMPLICIT mode, the application gets immediate notification when an item is selected. Whenever an element in the List is selected, a Command of the type List.SELECT_COMMAND is issued. As in the ChoiceGroup, the elements consist of Strings and optional Images.

For initializing the List, the lcdui packages offers constructors. The constructors work like the ChoiceGroup constructors. The first one creates an empty List with a given title and type only. The second one takes the title, the type, an array of Strings as initial amount of List elements, and an optional array of Images for each List element. In the implementation of the TeleTransfer application, you implement a new class CurrencyList extending List that will be used as your new currency selector. Since you will use the IMPLICIT mode, you need to implement a command listener, so you can already add the corresponding declaration:

```
public class CurrencyList extends List implements CommandListener {
```

To set the labels of the main form TextFields according to the index of the selected element in the CurrencyList, you create two String arrays, CURRENCY_NAMES and CURRENCY_FRACTIONS:

```
static final String [] CURRENCY_NAMES = {"Dollar", "Euro", "Yen"} ;
```

```
static final String [] CURRENCY_FRACTIONS = {"Cent", "Cent", "Sen"} ;
```

In order to set the labels of the main forms `TextField`s for the whole and the fractional amount according to the selected currency in the `CurrencyList`, you need a reference back to the main `TeleTransfer` MIDlet. For this reason, you store the `TeleTransfer` reference in a variable called `teleTransfer`. The reference is set in the constructor of your `CurrencyList`:

```
TeleTransfer teleTransfer;
```

In the constructor, you also add currency symbol images to the list. You need to load them, but the call to the super constructor must be the first statement in a constructor. So you call the constructor of the super class by specifying the title and type only. Then you create the `Image`s needed for each list element, which are stored in the MIDlet suite's JAR file. You also call `setCommandListener()` to register the currency list for handling commands that are issued:

```
public CurrencyList (TeleTransfer teletransfer) {
    super ("Select Currency", Choice.IMPLICIT);

    this.teleTransfer = teletransfer;

    try {
        append ("USD", Image.createImage ("/Dollar.png"));
        append ("EUR", Image.createImage ("/Euro.png"));
        append ("JPY",  Image.createImage ("/Yen.png"));
    }
    catch (java.io.IOException x) {
        throw new RuntimeException ("Images not found");
    }
    setCommandListener(this);
}
```

The final step in creating the `CurrencyList` is to implement the `commandAction()` method of the `CommandListener` interface. As you already know, a `List` of `IMPLICIT` type issues a `List.SELECT_COMMAND` to the registered `CommandListener` whenever a new element is selected to indicate the selection change. In case of a selection change, you modify the labels of the main form `TextField`s. The actual labels are obtained from the `String` arrays `CURRENCY_NAMES` and `CURRENCY_FRACTIONS`. Using the `teleTransfer` reference, you can access the `TextField`s. Finally, you call the new method `teleTransfer.back()`, which sets the screen back to the main form (the `back()` method will be given at the end of this section):

```
    public void commandAction (Command c, Displayable d) {
        if (c == List.SELECT_COMMAND) {
            teleTransfer.amountWhole.setLabel
                (CURRENCY_NAMES [getSelectedIndex()]);
            teleTransfer.amountFraction.setLabel
                (CURRENCY_FRACTIONS [getSelectedIndex()]);
            teleTransfer.back();
        }
    }
}
```

Figure 3.7 shows currency `Image`s and abbreviations in the `CurrencyList`.

Figure 3.7. The new `CurrencyList`.

The `TextBox` Class

Beneath `Alert`, `List`, and `Form`, there is only one further subclass of `Screen`: the `TextBox`. The `TextBox` allows the user to enter multi-line text on a separate screen. The constructor parameters and the constraint constants are identical to those of `TextField`.

As for the currency list, you can also add a new screen enabling the user to enter a transfer reason if desired. Similar to the `CurrencyList`, you implement a new class handling the commands related to the new screen. However, this time it is derived from the `TextBox`. Again, you implement the `CommandListener` interface:

```
public class TransferReason extends TextBox implements
CommandListener {
```

In the `TextBox`, you provide two commands, `okCommand` for applying the entered text and `clearCommand` for clearing the text:

```
static final Command okCommand = new Command ("OK", Command.BACK, 1);
static final Command clearCommand = new Command ("Clear",
Command.SCREEN, 2);
```

Again, you store a reference back to the `TeleTransfer` MIDlet in the `TransferReason` `TextBox`:

```
TeleTransfer teleTransfer;
```

The constructor gets the reference back to `TeleTransfer` MIDlet and stores it in the variable declared previously. You also add the commands to the `TextBox`, and register it as `CommandListener`:

```
public TransferReason (TeleTransfer teleTransfer) {
    super ("Transfer Reason", "", 50, TextField.ANY);
    this.teleTransfer = teleTransfer;

    addCommand (okCommand);
    addCommand (clearCommand);
    setCommandListener(this);
}
```

Your `commandAction()` implementation clears the text or returns to the main screen, depending on the `Command` selected:

```
    public void commandAction (Command c, Displayable d) {
        if (c == clearCommand) {
            setString ("");
        }
        else if (c == okCommand) {
```

```
            teleTransfer.back();
        }
    }
}
```

Figure 3.8 shows the `TransferReason` `TextBox`.

Figure 3.8. The `TransferReason` `TextBox` showing a sample transfer reason text.

`TeleTransfer` with Multiple Screens

Now you have created two additional screens, but you still need to integrate them in your main application. To do so, you need to change the `TeleTransfer` implementation somewhat. Since the `TeleTransfer`'s `ChoiceGroup` for selecting the currency is replaced by the `CurrencyList`, you do not need the `ItemStateListener` for detecting item changes any more. So you remove the listener and also the corresponding callback method `itemStateChanged()`. To display the two new `Screen`s `CurrencyList` and `TransferReason`, you implement the two commands `currencyCommand` and `reasonCommand`. The new commands are added to the MIDlet in the constructor using the `addCommand()` method. In the `clear()` method, the new `TextBox` is also cleared by calling the corresponding `setString()` method. Finally you add the `back()` method to the `TeleTransfer` application; this method is called from the new `Screen`s to return to the main form. The `commandAction()` method is extended to handle the new commands, displaying the new `Screen`s. Listing 3.1 shows the complete source code of the final version of the `TeleTransfer` application.

Listing 3.1 `TeleTransfer.java`—The Complete `TeleTransfer` Sample Source Code

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

class CurrencyList extends List implements CommandListener {

    TeleTransfer teleTransfer;
    static final String [] CURRENCY_NAMES = {"Dollar", "Euro",
"Yen"} ;
    static final String [] CURRENCY_FRACTIONS = {"Cent", "Cent",
"Sen"} ;

    public CurrencyList (TeleTransfer teletransfer) {
        super ("Select Currency", Choice.IMPLICIT);

        this.teleTransfer = teletransfer;

        try {
            append ("USD", Image.createImage ("/Dollar.png"));
            append ("EUR", Image.createImage ("/Euro.png"));
            append ("JPY",  Image.createImage ("/Yen.png"));
        }
```

```
        catch (java.io.IOException x) {
            throw new RuntimeException ("Images not found");
        }
        setCommandListener(this);
    }


    public void commandAction (Command c, Displayable d) {
        if (c == List.SELECT_COMMAND) {
            teleTransfer.amountWhole.setLabel
                (CURRENCY_NAMES [getSelectedIndex()]);
            teleTransfer.amountFraction.setLabel
                (CURRENCY_FRACTIONS [getSelectedIndex()]);
            teleTransfer.back();
        }
    }
}


class TransferReason extends TextBox implements CommandListener {

    static final Command okCommand = new Command ("OK", Command.BACK,
1);
    static final Command clearCommand = new Command
        ("Clear", Command.SCREEN, 2);

    TeleTransfer teleTransfer;

    public TransferReason (TeleTransfer teleTransfer) {
        super ("Transfer Reason", "", 50, TextField.ANY);
        this.teleTransfer = teleTransfer;

        addCommand (okCommand);
        addCommand (clearCommand);
        setCommandListener(this);
    }

    public void commandAction (Command c, Displayable d) {
        if (c == clearCommand) {
            setString ("");
        }
        else if (c == okCommand) {
            teleTransfer.back();
        }
    }
}

public class TeleTransfer extends MIDlet implements CommandListener {

    static final Command sendCommand = new Command ("Send",
Command.SCREEN, 2);
    static final Command clearCommand = new Command
        ("Clear", Command.SCREEN, 2);
    static final Command exitCommand = new Command ("Exit",
Command.SCREEN, 1);
    static final Command currencyCommand = new Command
        ("Currency", Command.SCREEN, 2);
    static final Command reasonCommand = new Command
        ("Reason", Command.SCREEN, 2);
    Form mainForm = new Form ("TeleTransfer");
```

```
    TextField receiverName = new TextField
        ("Receiver Name", "", 20, TextField.ANY);
    TextField receiverAccount = new TextField
        ("Receiver Account#", "", 8, TextField.NUMERIC);
    TextField amountWhole = new TextField ("Dollar", "", 6,
TextField.NUMERIC);
    TextField amountFraction = new TextField
                                       ("Cent", "", 2,
TextField.NUMERIC);

    CurrencyList currencyList = new CurrencyList (this);
    TransferReason transferReason = new TransferReason (this);
    Display display;

    public TeleTransfer() {
        mainForm.append (receiverName);
        mainForm.append (receiverAccount);
        mainForm.append (amountWhole);
        mainForm.append (amountFraction);

        mainForm.addCommand (currencyCommand);
        mainForm.addCommand (reasonCommand);
        mainForm.addCommand (sendCommand);
        mainForm.addCommand (exitCommand);
        mainForm.setCommandListener(this);
    }

    public void startApp() {
        display = Display.getDisplay (this);
        display.setCurrent (mainForm);
    }

    public void clear() {
        receiverName.setString ("");
        receiverAccount.setString ("");
        amountWhole.setString ("");
        amountFraction.setString ("");
        transferReason.setString ("");
    }

    public void send() {
        Alert alert = new Alert ("Send");
        alert.setString ("transfer " + amountWhole.getString()
                         + "." + amountFraction.getString()
                         + " " + amountWhole.getLabel()
                         + "\nto Acc#" + receiverAccount.getString()
                         + "\nof " + receiverName.getString());
        alert.setTimeout (2000);
        display.setCurrent (alert);
        clear();
    }

    public void pauseApp() {
    }

    public void destroyApp (boolean unconditional) {
    }

    public void back() {
        display.setCurrent (mainForm);
    }
```

```
    public void commandAction (Command c, Displayable d) {
        if (c == exitCommand) {
            notifyDestroyed();
        }
        else if (c == sendCommand) {
            sendTransferInformation();
        }
        else if (c == clearCommand) {
            resetTransferInformation();
        }
        else if (c == currencyCommand) {
            display.setCurrent (currencyList);
        }
        else if (c == reasonCommand) {
            display.setCurrent (transferReason);
        }
    }
}
```

Low-Level API

In contrast to the high-level API, the low-level API allows full control of the MID display at pixel level. For this purpose, the `lcdui` package contains a special kind of screen called `Canvas`. The `Canvas` itself does not provide any drawing methods, but it does provide a `paint()` callback method similar to the `paint()` method in AWT components. Whenever the program manager determines that it is necessary to draw the content of the screen, the `paint()` callback method of `Canvas` is called. The only parameter of the `paint()` method is a `Graphics` object. In contrast to the `lcdui` high-level classes, there are many parallels to AWT in the low-level API.

The `Graphics` object provides all the methods required for actually drawing the content of the screen, such as `drawLine()` for drawing lines, `fillRect()` for drawing a filled rectangular area or `drawstring()` for drawing text strings.

In contrast to AWT, `lcdui` does not let you mix high-level and low-level graphics. It is not possible to display high-level and low-level components on the screen simultaneously.

The program manager knows that it must call the `paint()` method of `Canvas` when the instance of `Canvas` is shown on the screen. However, a repaint can also be triggered by the application at any time. By calling the `repaint()` method of `Canvas`, the system is notified that a repaint is necessary, and it will call the `paint()` method. The call of the `paint()` method is not performed immediately; it may be delayed until the control flow returns from the current event handling method. The system may also collect several repaint requests before `paint()` is actually called. This delay normally is not a problem, but when you're doing animation, the safest way to trigger repaints is to use `Display.callSerially()` or to request the repaint from a separate `Thread` or `TimerTask`. Alternatively, the application can force an immediate repaint by calling `serviceRepaints()`. (For more information, see the section "Animation" at the end of this chapter.)

The `Canvas` class also provides some input callback methods that are called when the user presses or releases a key or touches the screen with the stylus (if one is supported by the device).

Basic Drawing

Before we go into the details of user input or animation, we will start with a small drawing example showing the concrete usage of the `Canvas` and `Graphics` classes.

The example clears the screen by setting the color to white and filling a rectangle the size of the screen, determined by calling `getWidth()` and `getHeight()`. Then it draws a line from coordinates (0,0) to (100,200). Finally, it draws a rectangle starting at (20,30), 30 pixels wide and 20 pixels high:

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

class DrawingDemoCanvas extends Canvas {

    public void paint (Graphics g) {
        g.setGrayScale (255);
        g.fillRect (0, 0, getWidth(), getHeight());

        g.setGrayScale (0);
        g.drawLine (0, 0, 100, 200);
        g.fillRect (20, 30, 30, 20);
    }
}
```

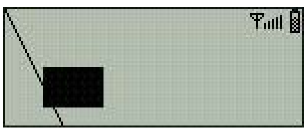
As you can see in the example code, you create a custom class `DrawingDemoCanvas` in order to fill the `paint()` method. Actually, it is not possible to draw custom graphics without creating a new class and implementing the `paint()` method.

In order to really see your Canvas implementation running, you still need a corresponding MIDlet. Here's the missing code:

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

public class DrawingDemo extends MIDlet {

    public void startApp() {
        Display.getDisplay (this).setCurrent (new
DrawingDemoCanvas());
    }

    public void pauseApp() {}

    public void destroyApp (boolean forced) {}
}
```

Now you can start your `DrawingDemo` MIDlet. Depending on the screen size of the device, it will create output similar to Figure 3.9. In most subsequent examples, you will omit the MIDlet since it is basically the same as this one, except that the name of your `Canvas` class will be different.

Figure 3.9. Output of the `DrawingDemo` MIDlet.



In the example, the screen is cleared before drawing because the system relies on the `paint()` method to fill every pixel of the draw region with a valid value. You don't erase the previous content of the screen automatically because doing so may cause flickering of animations. The application cannot make any assumptions about the content of the `Screen` before `paint()` is called. The screen may be filled with the content drawn at the last call of `paint()`, but it may also be filled with an alert box remaining from an incoming phone call, for example.

Drawing Style and Color

In the `DrawingDemoCanvas` implementation, you can find two calls to `setGrayScale()`. The `setGrayScale()` method sets the gray scale value for the following drawing operations. Valid grayscale values range from 0 to 255, where 0 means black and 255 means white. Not all possible values may actually render to different gray values on the screen. If the device provides fewer than 256 shades of gray, the best fitting value supported by the device is chosen. In the example, the value is first set to white, and the screen is cleared by the following call to `drawRect()`. Then, the color is set to black for the subsequent drawing operations.

The `setGrayScale()` method is not the only way to influence the color of subsequent drawing. MIDP also provides a `setColor()` method. The `setColor()` method has three parameters holding the red, green, and blue components of the desired color. Again, the values range from 0 to 255, where 255 means brightest and 0 means darkest. If all three parameters are set to the same value, the call is equivalent to a corresponding call of `setGrayScale()`. If the device is not able to display the desired color, it chooses the best fitting color or grayscale supported by the device automatically. Some examples are listed in Table 3.7.

Table 3.7. Example Color Parameter Settings

Parameter Settings	*Resulting Color*
`setColor` (255, 0, 0)	Red
`setColor` (0, 255, 0)	Green
`setColor` (0, 0, 255)	Blue
`setColor` (128, 0, 0)	Dark red
`setColor` (255, 255, 0)	Yellow
`setColor` (0, 0, 0)	Black
`setColor` (255, 255, 255)	White
`setColor` (128, 128, 128)	50% gray

The only other method that influences the current style of drawing is the `setStrokeStyle()` method. The `setStrokeStyle()` command sets the drawing style of lines to dotted or solid. You determine the style by setting the parameter to one of the constants `DOTTED` or `SOLID`, defined in the `Graphics` class.

When the `paint()` method is entered, the initial drawing color is always set to black and the line style is `SOLID`.

Simple Drawing Methods

In the example, you have already seen `fillRect()` and `drawLine()`. Table 3.8 shows all drawing primitives contained in the `Graphics` class. All operations where the method names begin with `draw`, except `drawString()` and `drawImage()`, are influenced by the current color and line style. They draw the outline of a figure, whereas the `fill` methods fill the corresponding area with the current color and do not depend on the line style.

Table 3.8. Drawing Methods of the `Graphics` Class

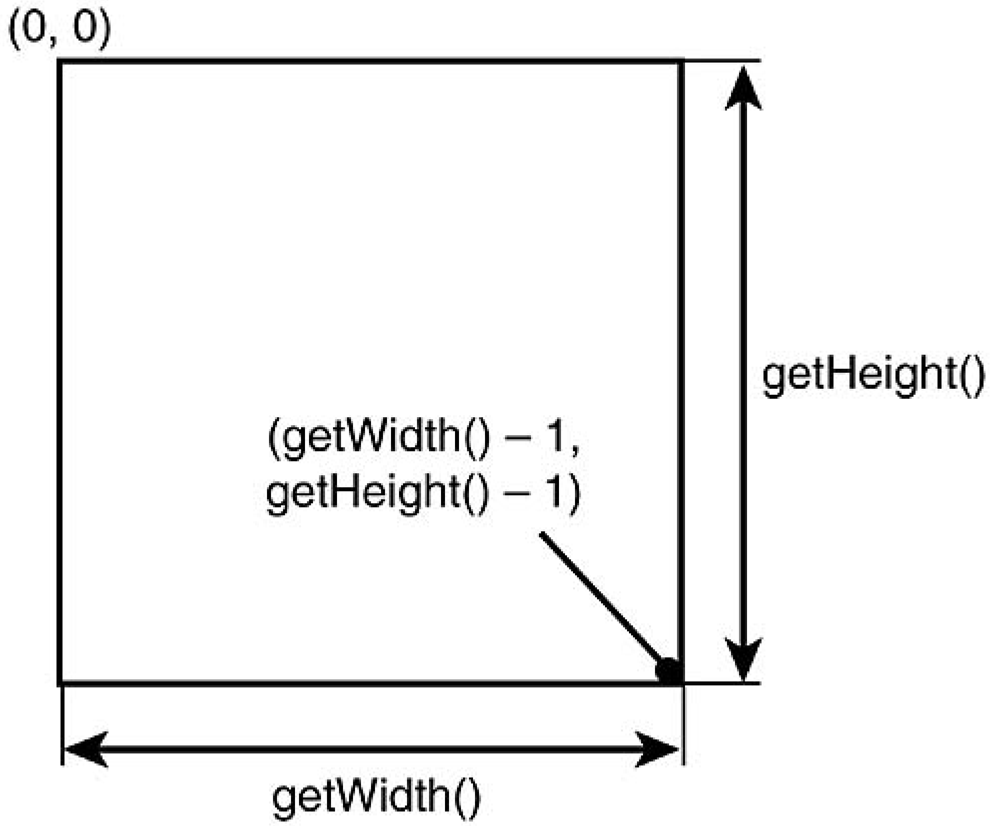
Method	*Purpose*
`drawImage (Image image, int x, int y, int align)`	Draws an Image. Explained in detail in the "Images" section.
`drawString (String text,`	Draws a text string at the given position in the current color; see "Text and Fonts."

`int x, int y, int align)`	
`drawRect (int x, int y, int w, int h)`	Draws an empty rectangle with the upper-left corner at the given (x,y)coordinate, with the given width and a height. The next section explains why the rectangle is one pixel larger than you might expect.
`drawRoundRect (int x, int y, int w, int h, int r)`	Like `drawRect()`, except that an additional radius is given for rounded corners of the rectangle.
`drawLine (int x0, int y0, int x1, int y1)`	Draws a line from (x0,y0) to (x1,y1).
`drawArc (int x, int y, int w, int h, int startAng, int arcArc)`	Draws the outline of a circular or elliptical arc covering the specified rectangle, using the current color and stroke style. The resulting arc begins at `startAng` and extends for `arcAng` degrees. Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation.
`fillRect (int x, int y, int w, int h)`	Similar to `drawRect()`, but fills the given area with the current color.
`fillRoundRect (int x, int y, int w, int h, int startAng, int endAng);`	Related to `fillRect()` as `drawRoundRect()` is related to `drawRect()`.
`fillArc (int x, int y, int w, int h, int startAng, int endAng);`	Like `drawArc()`, but fills the corresponding region.

Coordinate System and Clipping

In the drawing example, we already have used screen coordinates without explaining what they actually mean. You might know that the device display consists of little picture elements (pixels). Each of these pixels is addressed by its position on the screen, measured from the upper-left corner of the device, which is the origin of the coordinate system. Figure 3.10 shows the `lcdui` coordinate system.

Figure 3.10. The `lcdui` coordinate system.



Actually, in Java the coordinates do not address the pixel itself, but the space between two pixels, where the "drawing pen" hangs to the lower right. For drawing lines, this does not make any difference, but for rectangles and filled rectangles this results in a difference of one pixel in width and height: In contrast to filled rectangles, rectangles become one pixel wider and higher than you might expect. While this may be confusing at first glance, it respects the mathematical notation that lines are infinitely thin and avoids problems when extending the coordinate system to real distance measures, as in the J2SE class `Graphics2D`.

In all drawing methods, the first coordinate (x) denotes the horizontal distance from the origin and the second coordinate (y) denotes the vertical distance. Positive coordinates mean a movement down and to the right. Many drawing methods require additional width and height parameters. An exception is the `drawLine()` method, which requires the absolute coordinates of the destination point.

The origin of the coordinate system can be changed using the `translate()` method. The given coordinates are added to all subsequent drawing operations automatically. This may make sense if addressing coordinates relative to the middle of the display is more convenient for some applications, as shown in the section "Scaling and Fitting," later in the chapter.

The actual size of the accessible display area can be queried using the `getWidth()` and `getHeight()` methods, as performed in the first example that cleared the screen before drawing. The region of the screen where drawing takes effect can be further limited to a rectangular area by the `clipRect()` method. Drawing outside the clip area will have no effect.

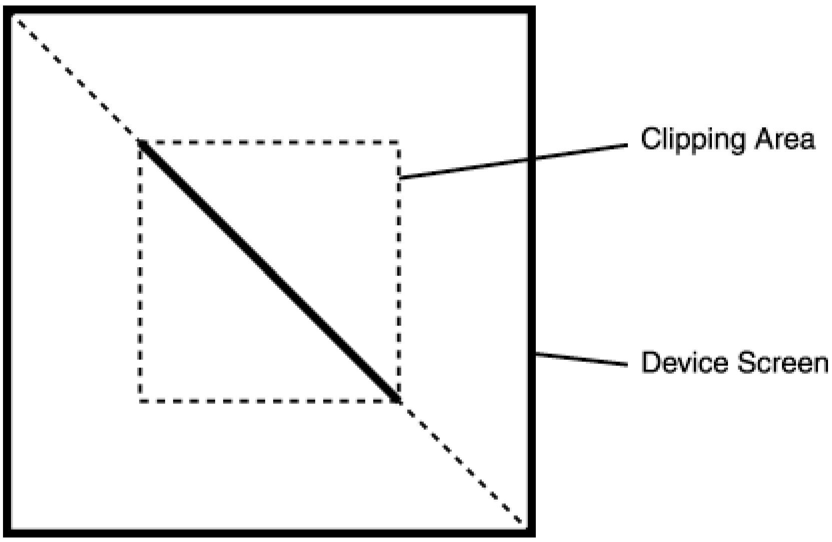
The following example demonstrates the effects of the `clipRect()` method. First, a dotted line is drawn diagonally over the display. Then a clipping region is set. Finally, the same line as before is drawn using the `SOLID` style:

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

```
class ClipDemoCanvas extends Canvas {

    public void paint (Graphics g) {
        g.setGrayScale (255);
        g.fillRect (0, 0, getWidth(), getHeight());

        int m = Math.min (getWidth(), getHeight());
        g.setGrayScale (0);

        g.setStrokeStyle (Graphics.DOTTED);
        g.drawLine (0, 0, m, m);

        g.setClip (m / 4, m / 4, m / 2, m / 2);

        g.setStrokeStyle (Graphics.SOLID);
        g.drawLine (0, 0, m, m);
    }
}
```

Figure 3.11 shows the resulting image. Although both lines have identical start and end points, only the part covered by the clipping area is replaced by a solid line.

Figure 3.11. Output of the `clipRect()` example: Only the part covered by the clipping area is redrawn solid, although the line coordinates are identical.



When the `paint()` method is called from the system, a clip area may already be set. This may be the case if the application just requested repainting of a limited area using the parameterized repaint call, or if the device just invalidated a limited area of the display, for example if a pop-up dialog indicating an incoming call was displayed but did not cover the whole display area.

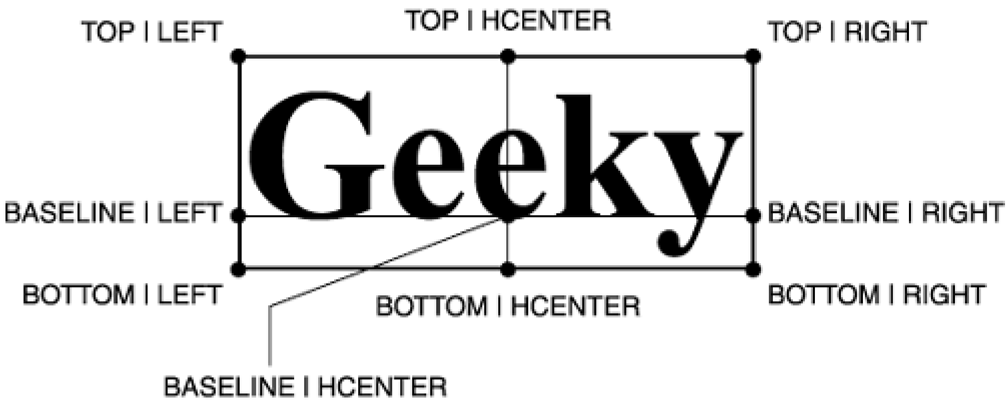
Actually, `clipRect()` does not set a new clipping area, but instead shrinks the current clip area to the intersection with the given rectangle. In order to enlarge the clip area, use the `setClip()` method.

The current clip area can be queried using the `getClipX()`, `getClipY()`, `getClipWidth()`, and `getClipHeight()` methods. When drawing is computationally expensive, this information can be taken into account in order to redraw only the areas of the screen that need an update.

Text and Fonts

For drawing text, `lcdui` provides the method `drawString()`. In addition to the basic `drawString()` method, several variants let you draw partial strings or single characters. (Details about the additional methods can be found in the `lcdui` API documentation.) The simple `drawString()` method takes four parameters: The character string to be displayed, the x and y coordinates, and an integer determining the horizontal and vertical alignment of the text. The alignment parameter lets you position the text relative to any of the four corners of its invisible surrounding box. Additionally, the text can be aligned to the text baseline and the horizontal center. The sum or logical or (`|`) of a constant for horizontal alignment (`LEFT`, `RIGHT`, and `HCENTER`) and constants for vertical alignment (`TOP`, `BOTTOM`, and `BASELINE`) determine the actual alignment. Figure 3.12 shows the anchor points for the valid constant combinations.

Figure 3.12. Valid combinations of the alignment constants and the corresponding anchor points.



The following example illustrates the usage of the `drawString()` method. By choosing the anchor point correspondingly, the text is displayed relative to the upper-left and lower-right corner of the screen without overlapping the screen border:

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

class TextDemoCanvas extends Canvas {

    public void paint (Graphics g) {
        g.setGrayScale (255);
        g.fillRect (0, 0, getWidth(), getHeight());

        g.setGrayScale (0);
        g.drawString ("Top/Left", 0, 0, Graphics.TOP |
Graphics.LEFT);
        g.drawString ("Baseline/Center", getWidth() / 2, getHeight()
/ 2,
                      Graphics.HCENTER | Graphics.BASELINE);
        g.drawString ("Bottom/Right", getWidth(), getHeight(),
                      Graphics.BOTTOM | Graphics.RIGHT);
    }
}
```

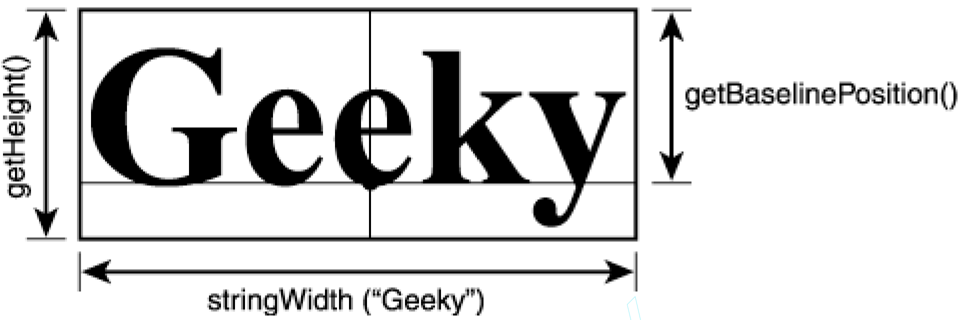
Figure 3.13 shows the output of the `TextDemo` example.

Figure 3.13. Output of the `TextDemo` example.

In addition to the current drawing color, the result of the `drawString()` method is influenced by the current font. MIDP provides support for three different fonts in three different sizes and with the three different attributes: bold, italic, and underlined.

A font is not selected directly, but the `setFont()` method takes a separate `Font` object, describing the desired font, as a parameter. The explicit `Font` class provides additional information about the font, such as its width and height in pixels, baseline position, ascent and descent, and so on. Figure 3.14 illustrates the meaning of the corresponding values. This information is important for operations such as drawing boxes around text strings. In addition, word-wrapping algorithms rely on the actual pixel width of character strings when rendered to the screen.

Figure 3.14. Font properties and the corresponding query methods.



A `Font` object is created by calling the static method `createFont()` of the class `Font` in the `lcdui` package. The `createFont()` method takes three parameters: the font type, style, and size of the font. Similar to the text alignment, there are predefined constants for setting the corresponding value; these constants are listed in Table 3.9.

Table 3.9. `createFont()` Property Constants

Property	***Constants***
Size	`SIZE_SMALL`, `SIZE_MEDIUM`, `SIZE_LARGE`
Style	`STYLE_PLAIN`, `STYLE_ITALICS`, `STYLE_BOLD`, `STYLE_UNDERLINED`
Face	`FACE_SYSTEM`, `FACE_MONOSPACE`, `FACE_PROPORTIONAL`

The style constants can be combined—for example, `STYLE_ITALICS | STYLE_BOLD` will result in a bold italics font style.

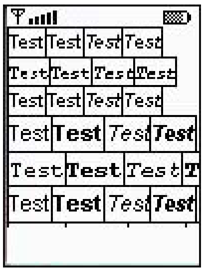
The following example shows a list of all fonts available, as far as the list fits on the screen of the device:

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

class FontDemoCanvas extends Canvas {

    static final int [] styles = {Font.STYLE_PLAIN,
                                  Font.STYLE_BOLD,
                                  Font.STYLE_ITALIC} ;
    static final int [] sizes = {Font.SIZE_SMALL,
                                 Font.SIZE_MEDIUM,
                                 Font.SIZE_LARGE} ;
    static final int [] faces = {Font.FACE_SYSTEM,
                                 Font.FACE_MONOSPACE,
                                 Font.FACE_PROPORTIONAL} ;
    public void paint (Graphics g) {
        Font font = null;
```

```
        int y = 0;
        g.setGrayScale (255);
        g.fillRect (0, 0, getWidth(), getHeight());
        g.setGrayScale (0);

        for (int size = 0; size < sizes.length; size++) {
            for (int face = 0; face < faces.length; face++) {
                int x = 0;
                for (int style = 0; style < styles.length; style++) {
                    font = Font.getFont
                              (faces [face], styles [style], sizes
[size]);
                    g.setFont (font);
                    g.drawString
                        ("Test", x+1, y+1, Graphics.TOP |
Graphics.LEFT);
                    g.drawRect
                        (x, y, font.stringWidth ("Test")+1,
                         font.getHeight() + 1);
                    x += font.stringWidth ("Test")+1;
                }
                y += font.getHeight() + 1;
            }
        }
    }
}
```

Figure 3.15 shows the output of the `FontDemo` example.

Figure 3.15. Output of the `FontDemo` example.



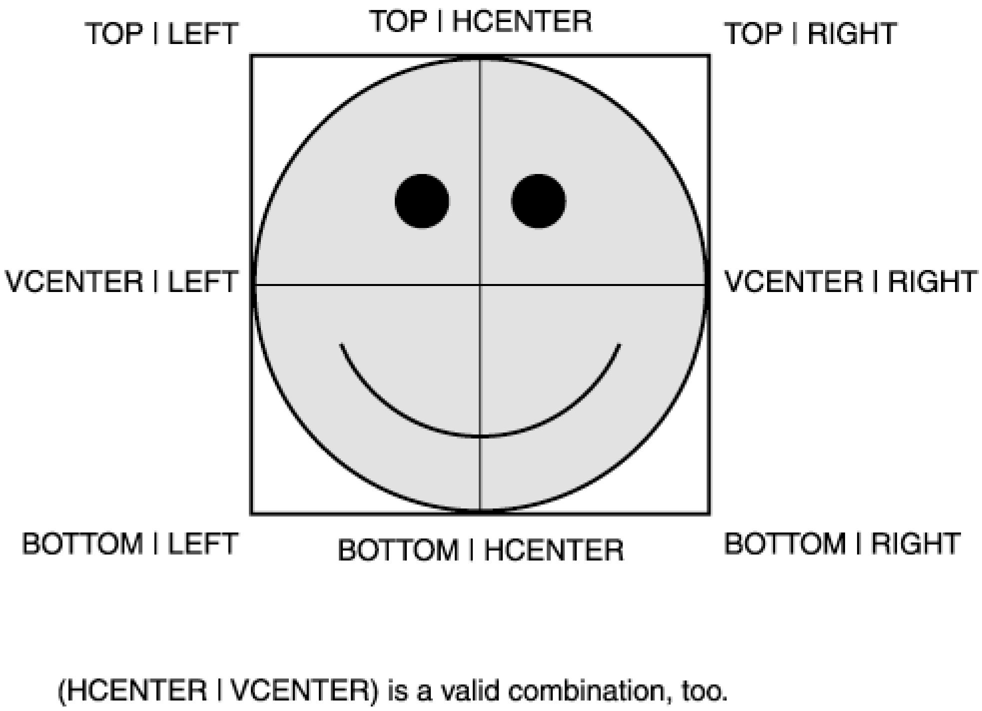
Images

The `Graphics` class also provides a method for drawing images. As shown in the final version of `TeleTransfer` application, `Image`s can be predefined and contained in the JAR file of the MIDlet. The only file format that is mandatory for MIDP is the Portable Network Graphics (PNG) file format. The PNG format has several advantages over other graphics formats; for example, it is license free and supports true color images, including a full transparency (alpha) channel. PNG images are always compressed with a loss-less algorithm. The algorithm is identical to the algorithm used for JAR files, so the MIDP implementation can save space by using the same algorithm for both purposes.

An image can be loaded from the JAR file using the static method `Image.create (String` *`name`*`)`. The *`name`* parameter denotes the filename of the image in the JAR file. Please note that this `create()` method may throw an `IOException`.

The `drawImage()` method in `Graphics` requires an `Image` object, the coordinates, and an integer denoting the alignment as parameters. The alignment parameter is similar the alignment of `drawString()`, except that the `BASELINE` constant is not supported. An additional alignment constant available for images only is `VCENTER`, which forces the image to be vertically centered relative to the given coordinates. Figure 3.16 shows the valid constant combinations and the corresponding anchor points.

Figure 3.16. Alignment constant combinations valid for images and the corresponding anchor points.

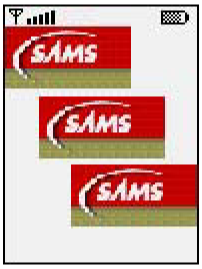


The following example first loads the image `logo.png` from the `MIDlet` JAR file in the constructor, and then displays the image three times. One image is drawn in the upper-left corner, one in the lower-right corner, and one in the center of the display, as shown in Figure 3.17:

```
import java.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
class ImageDemoCanvas extends Canvas {

    Image image;

    public ImageDemoCanvas() {
        try {
            image = Image.createImage ("/logo.png");
        }
        catch (IOException e) {
            throw new RuntimeException ("Unable to load Image: "+e);
        }
    }

    public void paint (Graphics g) {
        g.setGrayScale (255);
        g.fillRect (0, 0, getWidth(), getHeight());

        g.drawImage (image, 0, 0, Graphics.TOP |  Graphics.LEFT);
        g.drawImage (image, getWidth() / 2, getHeight() / 2,
                     Graphics.HCENTER | Graphics.VCENTER);
        g.drawImage (image, getWidth(), getHeight(),
                     Graphics.BOTTOM | Graphics.RIGHT);
    }
```

```
}
```

Figure 3.17. Output of the `ImageDemo` example.



Images can also be created at runtime from scratch. The static method `Image.create (int width, int height)` creates a new dynamic image of the given size. In contrast to images loaded from a JAR file, these images are mutable. Mutable images can be modified by calling `getGraphics()`. The `Graphics` object returned can be used for modifying the image with all the methods provided by the `Graphics` class. Please note that images loaded from a JAR file cannot be modified. However, it is possible to create a mutable image, and then draw any other image in the mutable image.

By modifying the constructor of the previous example canvas as follows, the image drawn in the `paint()` method is created and filled at runtime instead of loading an image from the JAR file:

```
public ImageDemoCanvas() {
    image = Image.createImage (10,10);
    image.getGraphics().fillArc (0,0,10,10,0, 360);
}
```

The disadvantage of mutable images is that they cannot be used in high-level GUI elements since it is possible to modify them at any time, possibly leading to inconsistent display of widgets. For that reason, another static create method, `createImage(Image image)`, is provided that creates an immutable image from another image.

Interaction

Because the `Canvas` class is a subclass of `Displayable`, it provides the same support for commands as the high-level screen classes. Here, you will concentrate on the additional interaction possibilities the `Canvas` class offers: direct key input and pointer support.

Please note that all input events and command notifications and the `paint()` method are called serially. That means that the application manager will call none of the methods until the previous event handling method has returned. So all these methods should return quickly, or the user will be unable to interact with the application. For longer tasks, a separate thread can be started.

Key Input

For key input, the `Canvas` class provides three callback methods: `keyPressed()`, `keyReleased()`, and `keyRepeated()`. As the names suggest, `keyPressed()` is called when a key is pressed, `keyRepeated()` is called when the user holds down the key for a longer period of time, and `keyReleased()` is called when the user releases the key.

All three callback methods provide an integer parameter, denoting the Unicode character code assigned to the corresponding key. If a key has no Unicode correspondence, the given integer is negative. MIDP defines the following constant for the keys of a standard ITU-T keypad: `KEY_NUM0`, `KEY_NUM1`, `KEY_NUM2`, `KEY_NUM3`, `KEY_NUM4`, `KEY_NUM5`, `KEY_NUM6`, `KEY_NUM7`, `KEY_NUM8`, KEY_NUM9, KEY_POUND, and KEY_STAR. Applications should not rely on the presence of any additional key codes. In particular, upper- and lowercase or characters generated by pressing a key multiple times are not supported by low-level key events. A "name" assigned to the key can be queried using the getKeyName() method.

Some keys may have an additional meaning in games. For this purpose, MIDP provides the constants UP, DOWN, LEFT, RIGHT, FIRE, GAME_A, GAME_B, GAME_C, and GAME_D. The "game" meaning of a keypress can be determined by calling the getGameAction() method. The mapping from key codes to game actions is device dependent, so different keys may map to the same game action on different devices. For example, some devices may have separate cursor keys; others may map the number pad to four-way movement. Also, several keys may be mapped to the same game code. The game code can be translated back to a key code using the getKeyCode() method. This also offers a way to get the name of the key assigned to a game action. For example, the help screen of an application may display

```
"press "+getKeyName (getKeyCode (GAME_A))
```

instead of "press GAME_A".

The following canvas implementation shows the usage of the key event methods. For each key pressed, repeated, or released, it shows the event type, character and code, key name, and game action.

The first part of the implementation stores the event type and code in two variables and schedules a repaint whenever a key event occurs:

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

class KeyDemoCanvas extends Canvas {

    String eventType = "- Press any!";
    int keyCode;

    public void keyPressed (int keyCode) {
        eventType = "pressed";
        this.keyCode = keyCode;
        repaint();
    }

    public void keyReleased (int keyCode) {
        eventType = "released";
        this.keyCode = keyCode;
        repaint();
    }

    public void keyRepeated (int keyCode) {
        eventType = "repeated";
        this.keyCode = keyCode;
        repaint();
    }
```

The second part prints all event properties available to the device screen. For this purpose, you first implement an additional write() method that helps the paint() method to identify the current y position on the screen. This is necessary because drawText() does not advance to a new line automatically. The write() method draws the string at the given y position and returns the y position plus the line height of the current font, so paint() knows where to draw the next line:

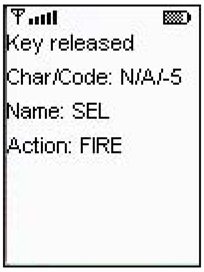
```
public int write (Graphics g, int y, String s) {
    g.drawString (s, 0, y, Graphics.LEFT|Graphics.TOP);
```

```
    return y + g.getFont().getHeight();
}
```

The `paint()` method analyzes the `keyCode` and prints the result by calling the `write()` method defined previously, as shown in Figure 3.18:

```
    public void paint (Graphics g) {
        g.setGrayScale (255);
        g.fillRect (0, 0, getWidth(), getHeight());

        g.setGrayScale (0);

        int y = 0;
        y = write (g, y, "Key "+ eventType);
        if (keyCode == 0) return;

        y = write (g, y, "Char/Code: "+ ((keyCode < 0) ? "N/A" : ""
                   +(char) keyCode) + "/" + keyCode);
        y = write (g, y, "Name: "+getKeyName (keyCode));
        String gameAction;
        switch (getGameAction (keyCode)) {
        case LEFT: gameAction = "LEFT"; break;
        case RIGHT: gameAction = "RIGHT"; break;
        case UP: gameAction = "UP"; break;
        case DOWN: gameAction = "DOWN"; break;
        case FIRE: gameAction = "FIRE"; break;
        case GAME_A: gameAction = "GAME_A"; break;
        case GAME_B: gameAction = "GAME_B"; break;
        case GAME_C: gameAction = "GAME_C"; break;
        case GAME_D: gameAction = "GAME_D"; break;
        default: gameAction = "N/A";
        }
        write (g, y, "Action: "+gameAction);
    }
}
```

Figure 3.18. Output of the `KeyDemo` example when the "Fire" key was released.



Pointer Events

For devices supporting a pointer device such as a stylus, touch screen, or trackball, the `Canvas` class provides three notification methods: `pointerPressed()`, `pointerDragged()`, and `pointerReleased()`. These methods work similarly to the key event methods, except that they provide two integer parameters, denoting the x and y position of the pointer when the corresponding event occurs. (Please note that pointer support is optional in MIDP, so the application should not rely on the presence of a pointer. Such devices are uncommon for devices such as mobile phones.) The following sample program demonstrates the usage of the three methods:

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

```
class PointerDemoCanvas extends Canvas {

    String eventType = "Press Pointer!";
    int x;
    int y;

    public void pointerPressed (int x, int y) {
        eventType = "Pointer Pressed";
        this.x = x;
        this.y = y;
        repaint();
    }

    public void pointerReleased (int x, int y) {
        eventType = "Pointer Released";
        this.x = x;
        this.y = y;
        repaint();
    }

    public void pointerDragged (int x, int y) {
        eventType = "Pointer Repeated";
        this.x = x;
        this.y = y;
        repaint();
    }

    public void paint (Graphics g) {
        g.setGrayScale (255);
        g.fillRect (0, 0, getWidth(), getHeight());
        g.setGrayScale (0);
        g.drawString (eventType + " " +x +"/"+y,
                      0, 0, Graphics.TOP|Graphics.LEFT);
        g.drawLine (x-4, y, x+4, y);
        g.drawLine (x, y-4, x, y+4);
    }
}
```

Foreground and Background Notifications

For several reasons, the `Canvas` may move into the background—for example, if the display is set to another displayable object or if the device displays a system dialog. In these cases, the `Canvas` is notified by the `hideNotify()` method. When the `Canvas` becomes visible (again), the corresponding counterpart, `showNotify()`, is called.

Javagochi Example

Now that you are familiar with the `Canvas` object and the basic drawing methods of the `Graphics` class, you are ready to develop a small interactive application, the `Javagochi`.

As you can see in the following code, the MIDlet implementation of `Javagochi` is already finished, but the `Face` class is missing:

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

public class Javagochi extends MIDlet {
```

```
    static final int IDEAL_WEIGHT = 100;
    Display display = Display.getDisplay (this);
    Face face = new Face (this);
    int weight = IDEAL_WEIGHT;
    Timer consumption;
    int score;
```

Before you begin development, let us first say a few words about the `Javagochi` itself. A `Javagochi` has a `weight` that is initialized with its `IDEAL_WEIGHT`. It also owns an instance of `Display`, `Face`, and `Consumption`, which will be explained later. Finally, it stores a score value for the care the owner spends on the `Javagochi`.

The happiness of the `Javagochi` is determined by the deviation of its current weight from the ideal weight, ranging from 10 to 0:

```
public int getHappiness() {
    return 20 - (weight > IDEAL_WEIGHT
                 ? 10 * weight / IDEAL_WEIGHT
                 : 10 * IDEAL_WEIGHT / weight);
    if (happiness < 0) happiness = 0;
    if (happiness > 10) happiness = 10;
}
```

This formula also demonstrates how to circumvent problems with the absence of floating point arithmetic. In order to avoid loss of significant fractions, the values are scaled up before division.

Like all other known life forms, the `Javagochi` can die. `Javagochi`es only die from sadness when their happiness level reaches zero:

```
public boolean isDead() {
    return getHappiness <= 0;
}
```

The only other action a `Javagochi` can perform besides dying is to transform energy to matter and back. Since a weight change may change the `Javagochi`'s look, a repaint is requested in the `transform()` method:

```
public void transform (int amount) {
    if (!isDead()) {
        weight += amount;
        face.repaint();
    }
}
```

When the `Javagochi` MIDlet is started, it displays itself and starts a consumption `Timer` that keeps track of the power the `Javagochi` needs for living:

```
public void startApp() {
    display.setCurrent (face);
    consumption = new Consumption (this).start();
}
```

When the MIDlet is paused, the `Javagochi` goes to sleep by telling the consumption thread to terminate itself. The `destroyApp()` method does nothing because the life cycle will enter sleep anyway, and no further cleanup is needed:

```
    public void pauseApp() {
```

```
        consumption.leave = true;
    }
    public void destroyApp (boolean forced) {
    }
}
```

The consumption `Thread` is a separate class that monitors the power the `Javagochi` needs for living. In the `run()` method, every 0.5 seconds the score is updated depending on the `Javagochi`'s happiness and the small amount of body mass that is transformed back to life energy:

```
public class Consumption extends Thread {

    Javagochi javagochi;
    boolean leave = false;

    public Consumption (Javagochi javagochi) {
        this.javagochi = javagochi;
    }

    public void run() {
        while (!leave) {
            try {
                sleep (500);
            }
            catch (InterruptedException e) {break;}
            javagochi.score += 10 - javagochi.deviation;
            javagochi.transform (-5);
        }
    }
}
```

Now that you know how a `Javagochi` works, it is your job to give the `Javagochi` an appropriate appearance by implementing the missing `Face` class.

Scaling and Fitting

In many cases, it is a good idea to scale displayed graphics depending on the actual screen size. Otherwise, the display will look nice on one particular device type, but won't fit the screen on devices with a lower screen resolution or become unnecessarily small on devices with higher screen resolutions.

We will now show how scaling works for the `Javagochi` example. A picture of a `Javagochi` is shown in Figure 3.19. You will start by drawing the shape of the face, a simple ellipse. In this case, the ellipse will reflect the `Javagochi`'s weight. If the `Javagochi` is at its ideal weight, the ellipse becomes a circle.

Figure 3.19. A happy Javagochi at its ideal weight.

The Javagochi at IDEAL_WEIGHT

In order to leave some space for the `Javagochi` to grow, the diameter of the ideal circle is half the minimum of the screen width and height. Thus, the height of the `Javagochi` is calculated using the following formula:

```
int height = Math.min (getHeight(), getWidth()) / 2;
```

Based on the current weight, the ideal weight, and the calculated height, which is also the diameter of the "ideal" `Javagochi`, you can now calculate the width of the `Javagochi`:

```
int width = height * javagochi.weight  / javagochi.IDEAL_WEIGHT;
```

Other applications may of course have other dependencies from the actual screen size, but this example should be sufficient to show the general idea.

The `Javagochi`'s skin color is dependent on its happiness. If the `Javagochi` feels well, its skin has a bright yellow color. With decreasing happiness, the `Javagochi` becomes pale. This is reflected by the following `setColor()` command:

```
setColor (255, 255, 28 * javagochi.happiness);
```

Using the given width and height, you can now implement your first version of the `Javagochi`'s `Face` class:

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

class Face extends Canvas implements CommandListener {
    Javagochi javagochi;

    Face (Javagochi javagochi) {
        this.javagochi = javagochi;
    }

    public void paint (Graphics g) {
        g.setColor (255, 255, 255);
        g.fillRect (0, 0, getWidth(), getHeight());

        int height = Math.min (getHeight(), getWidth()) / 2;
        int width = height * javagochi.weight /
javagochi.IDEAL_WEIGHT;
```

```
        g.translate (getWidth() / 2, getHeight() / 2);

        g.setColor (255, 255, 255 - javagochi.getHappiness() * 25);
        g.fillArc (- width / 2, - height / 2, width, height, 0, 360);

        g.setColor (0, 0, 0);
        g.drawArc (- width / 2, - height / 2, width, height, 0, 360);
    }
}
```

In order to simplify the centered display of the `Javagochi`, you set the origin of the coordinate system to the center of the screen using the `translate()` method. The outline of the `Javagochi`'s face is then drawn using the `drawArc()` method.

Unfortunately, the outline of the `Javagochi` looks a bit boring, so you will add a simple face now. In order to avoid duplicated code, you put the drawing of the eyes in a separate method. The `drawEye()` method takes the `Graphics` object, the coordinates of the eye, and a size parameter:

```
void drawEye (Graphics g, int x, int y, int size) {
    if (javagochi.isDead()) {
        graphics.drawLine (x - size/2, y, x + size/2, y);
        graphics.drawLine (x, y - size/2, x, y + size/2);
    }
    else
        graphics.drawArc (x-size/2, y-size/2, size, size, 0, 360);
}
```

Now you can insert the rest of the drawing code into the `paint()` method, just after `drawArc()`. You will start with the eyes by calling the `drawEye()` method defined previously. By using fractions of the current width and height of the `Javagochi`, the eyes are positioned and sized correctly:

```
drawEye (g, - width / 6, - height / 5, height / 15 + 1);
drawEye (g, width / 6, - height / 5, height / 15 + 1);
```

Now you draw the mouth, depending on the current happiness of the `Javagochi`. Again, you use fractions of the `Javagochi` size for positioning and sizing:

```
switch (javagochi.getHappiness() / 3) {
case 0:
case 1: g.drawArc (-width/6,  height/7, width/3, height/6, 0, 180);
break;
case 2: g.drawLine (-width/6,  height/7, width/6, height/7); break;
default: g.drawArc (-width/6,  height/7, width/3, height/6, 0, -180);
}
```

Simple Interaction

When you run the first version of the `Javagochi` application, the `Javagochi` starts out happy, but dies quickly from starvation. Obviously, you need a way to transfer energy from the device's battery to the `Javagochi`. One possibility would be to add a corresponding command.

However, in the "High-Level API" section you learned that commands may be delegated to a sub-menu. When the `Javagochi` urgently needs feeding, you would like to be able to react quickly.

So you just use the key event corresponding to the game action `FIRE` for feeding the `Javagochi`:

```
public void keyPressed (int keyCode) {
    if (getGameAction (keyCode) == FIRE)
        javagochi.transform (10);
}
```

Now you can save the `Javagochi` from starvation using the `FIRE` game key.

Canvas and Text Input

As mentioned in the introduction to interaction, it is not possible to receive composed key events using the low-level API. But what can you do if you need this kind of input, such as for a text input trainer?

Let's just assume simple feeding is not enough for your `Javagochi`. Depending on its current state, it needs special vitamins, denoted by letters ranging from A to Z. On phones providing keys 0 through 9 only, this is a problem. The only solution is to emulate the key input mechanism in software. On cellular phones, there are also three to four letters printed on the number keys. In text input mode, pressing a number makes the first letter appear. If the same number is pressed again in a limited period of time, the second letter appears instead of the first one. This way you can cycle through all the letters on a number key. When no key is pressed for about three quarters of a second, or another key is pressed, the letter currently displayed is confirmed as input key.

For emulation of this mechanism, you define the letters on the keys 2 through 9 in a `String` array inside the `Face` class:

```
public static final String[] keys = {"abc", "def", "ghi", "jkl",
                                     "mno", "pqrs", "tuv", "wxyz"} ;
```

You also need a timer to measure the time until confirmation of the current key. The timer is stored in `keyTimer`. The variables `keyMajor` and `keyMinor` contain the index in the keys array and the index inside the corresponding string. The variable `needed` stores the vitamin currently needed by the `Javagochi`:

```
Timer keyTimer;
int keyMajor = -1;
int keyMinor;
char needed = 'a';
```

What do you do if a numeric key is pressed? If you already have a timer running, you cancel it since a key was pressed. Then, you subtract the code of the 2 key from the current key code in order to calculate the index in your key array. If the given event does not represent a numeric key between 2 and 9, you set `keyMajor` to the special value –1, denoting that no valid character is being entered. Otherwise, you check whether the key is identical to the last key. If so, `keyMinor` is incremented in order to cycle through the letters assigned to a single numeric key. If another key is pressed, `keyMajor` is changed accordingly and `keyMinor` is set back to 0. A new timer is scheduled for half a second later:

```
public synchronized void keyPressed (int keyCode) {

    if (keyTimer != null) keyTimer.cancel();

    int index = keyCode - KEY_NUM2;

    if (index < 0 || index > keys.length)
        keyMajor = -1;
    else {
        if (index != keyMajor) {
            keyMinor = 0;
            keyMajor = index;
```

```
        }
        else {
            keyMinor++;
            if (keyMinor >= keys [keyMajor].length())
                keyMinor = 0;
        }

        keyTimer = new Timer();
        keyTimer.schedule (new KeyConfirmer (this), 500);
    }
    repaint();
}
```

Now you need to implement a timer task that confirms the letter if no other key is pressed for half a second. In that case, the `KeyConfirmer` class just calls `keyConfirmed()` in the original `Face` class:

```
import java.util.*;

public class KeyConfirmer extends TimerTask {

    Face face;

    public KeyConfirmer (Face face) {
        this.face = face;
    }

    public void run() {
        face.keyConfirmed();
    }
}
```

Back in the `Face` class, you can now implement the functionality performed when the letter is finally confirmed. You just compare the letter to the vitamin needed by the `Javagochi`. If the right vitamin is fed, the weight of the `Javagochi` is increased 10 units by calling `transform()`:

```
synchronized void keyConfirmed() {
    if (keyMajor != -1) {

        if (keys [keyMajor].charAt (keyMinor) == needed) {
            javagochi.score += javagochi.getHappiness();

            if (!javagochi.isDead())
                needed = (char) ('a'
                    + ((System.currentTimeMillis() / 10) % 26));

            javagochi.transform (10);
        }

        keyMajor = -1;
        repaint();
    }
}
```

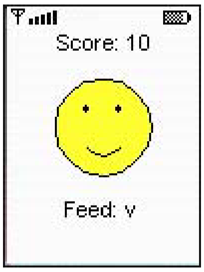
Finally, you add some status information about the current score and selected key to the `Face.paint()` method. Just insert the following code at the end of the previous implementation of `paint()`:

```
String keySelect = "";
```

```
if (keyMajor != -1) {
    String all = keys [keyMajor];
    keySelect = all.substring (0, keyMinor) + "[" + all.charAt
(keyMinor)
                               + "]" + all.substring (keyMinor+1);
}

g.drawString ("Feed: " + needed + " " + keySelect, 0,
              getHeight()/2, Graphics.BOTTOM|Graphics.HCENTER);
g.drawString ("Score: "+javagochi.score, 0,
              -getHeight()/2, Graphics.TOP|Graphics.HCENTER);
```

Figure 3.20 shows the `Javagochi` being fed with vitamins. The complete source code is contained in Listing 3.2.

Figure 3.20. A `Javagochi` being fed with vitamins.



Listing 3.2 `Javagochi.java`—The Complete `Javagochi` Sample Source Code

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


class Consumption extends TimerTask {

    Javagochi javagochi;

    public Consumption (Javagochi javagochi) {
        this.javagochi = javagochi;
    }
    public void run() {
        javagochi.transform (-1 - javagochi.score/100 );
    }
}


class KeyConfirmer extends TimerTask {

    Face face;

    public KeyConfirmer (Face face) {
        this.face = face;
    }

    public void run() {
        face.keyConfirmed();
    }
}
```

```
class Face extends Canvas {

    public static final String[] keys = {"abc", "def", "ghi", "jkl",
                                         "mno", "pqrs", "tuv",
"wxyz"} ;

    Javagochi javagochi;
    Timer keyTimer;

    int keyMajor = -1;
    int keyMinor;
    char needed = 'a';

    Face (Javagochi javagochi) {
        this.javagochi = javagochi;
    }

    public void paint (Graphics g) {
        g.setColor (255, 255, 255);
        g.fillRect (0, 0, getWidth(), getHeight());

        int height = Math.min (getHeight(), getWidth()) / 2;
        int width = height * javagochi.weight
            / javagochi.IDEAL_WEIGHT;

        g.translate (getWidth() / 2, getHeight() / 2);
        g.setColor (255, 255, 255 - javagochi.getHappiness() * 25);
        g.fillArc (- width / 2, - height / 2, width, height, 0, 360);

        g.setColor (0, 0, 0);
        g.drawArc (- width / 2, - height / 2, width, height, 0, 360);

        g.drawString ("Score: "+javagochi.score, 0, -getHeight()/2,
                      Graphics.TOP|Graphics.HCENTER);

        String keySelect = "";
        if (keyMajor != -1) {
            String all = keys [keyMajor];
            keySelect = all.substring
                            (0, keyMinor) + "[" + all.charAt
(keyMinor)
                             + "]" + all.substring (keyMinor+1);
        }

        g.drawString ("Feed: " + needed + " " + keySelect,
                      0, getHeight()/2,
Graphics.BOTTOM|Graphics.HCENTER);

        drawEye (g, - width / 6, - height / 5, height / 15 + 1);
        drawEye (g,  width / 6, - height / 5, height / 15 + 1);

        switch (javagochi.getHappiness() / 3) {
        case 0:
        case 1:
            g.drawArc (-width/6,  height/7, width/3, height/6, 0,
180);
            break;
        case 2:
            g.drawLine (-width/6,  height/7, width/6, height/7);
            break;
```

```
        default:
            g.drawArc (-width/6,  height/7, width/3, height/6, 0, -
180);
        }
    }

    void drawEye (Graphics graphics, int x0, int y0, int w) {
        if (javagochi.isDead()) {
            graphics.drawLine (x0 - w/2, y0, x0 + w/2, y0);
            graphics.drawLine (x0, y0 - w/2, x0, y0 + w/2);
        }
        else
            graphics.fillArc (x0-w/2, y0-w/2, w, w, 0, 360);
    }
    public synchronized void keyPressed (int keyCode) {

        int index = keyCode - KEY_NUM2;

        if (keyTimer != null) keyTimer.cancel();

        if (index < 0 || index > keys.length)
            keyMajor = -1;
        else {
            if (index != keyMajor) {
                keyMinor = 0;
                keyMajor = index;
            }
            else {
                keyMinor++;
                if (keyMinor >= keys [keyMajor].length())
                    keyMinor = 0;
            }

            keyTimer = new Timer();
            keyTimer.schedule (new KeyConfirmer (this), 500);
        }
        repaint();
    }

    synchronized void keyConfirmed() {
        if (keyMajor != -1) {

            if (keys [keyMajor].charAt (keyMinor) == needed) {
                javagochi.score += javagochi.getHappiness();

                if (!javagochi.isDead())
                    needed = (char) ('a'
                        + ((System.currentTimeMillis() / 10) % 26));

                javagochi.transform (10);
            }

            keyMajor = -1;
            repaint();
        }
    }
}
public class Javagochi extends MIDlet {

    static final int IDEAL_WEIGHT = 100;
```

```
    Display display;
    Face face = new Face (this);
    int weight = IDEAL_WEIGHT;
    Timer consumption;
    int score;

    public int getHappiness() {
        int happiness = 20 - (weight > IDEAL_WEIGHT
                          ? 10 * weight / IDEAL_WEIGHT
                          : 10 * IDEAL_WEIGHT / weight);
        if (happiness < 0) happiness = 0;
        else if (happiness > 10) happiness = 10;
        return happiness;
    }

    public boolean isDead() {
        return getHappiness() == 0;
    }

    public void transform (int amount) {
        if (!isDead()) {
            weight += amount;
            face.repaint();
        }
    }

    public void startApp() {
        display = Display.getDisplay (this);
        display.setCurrent (face);
        consumption = new Timer();
        consumption.scheduleAtFixedRate (new Consumption (this), 500,
500);
    }

    public void pauseApp() {
        consumption.cancel();
    }

    public void destroyApp (boolean forced) {
    }
}
```

Animation

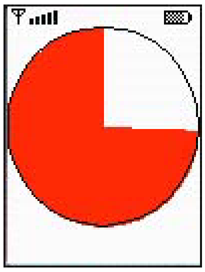
With animation, there are normally two main problems: `Display` flickering and synchronization of painting with calculation of new frames. We will first address how to get the actual painting and application logic in sync, and then solve possible flickering.

Synchronization of Frame Calculation and Drawing

When you perform animations, you can first calculate the display content and then call `repaint()` in order to paint the new frame. But how do you know that the call to `paint()` has finished? One possibility would be to call `serviceRepaints()`, which blocks until all pending display updates are finished. The problem with `serviceRepaints()` is that `paint()` may be called from another thread. If the thread calling `serviceRepaints()` holds any locks that are required in `paint()`, a deadlock may occur. Also, calling `serviceRepaints()` makes sense only from a thread other than the event handling thread. Otherwise, key events may be blocked until the animation is over. An alternative to `serviceRepaints()` is calling `callSerially()` at the end of the `paint()` method. The `callSerially()` method lets you put `Runnable` objects in the event queue. The run() method of the Runnable object is then executed serially like any other event handling method. In the run() method, the next frame can be set up, and a new repaint can be requested there.

To demonstrate this execution model, you will build a simple stopwatch that counts down a given number of seconds by showing a corresponding pie slice using the fillArc() method, as shown in Figure 3.21.

Figure 3.21. A very simple stopwatch.



The Canvas implementation stores the current slice in degree, the start time, the total amount of seconds and the MIDlet display in local variables. In order to make use of callSerially(), your Canvas implements the Runnable interface:

```
class StopWatchCanvas extends Canvas implements Runnable {
    int degree = 360;
    long startTime;
    int seconds;
    Display display;
```

When the StopWatchCanvas is created, you store the given display and seconds. Then, the current time is determined and stored, too:

```
StopWatchCanvas (Display display, int seconds) {
    this.display = display;
    this.seconds = seconds;
    startTime = System.currentTimeMillis();
}
```

In the paint() method, you clear the display. If you need to draw more than 0 degrees, you fill a corresponding arc with red color and request recalculation of the pie slice using callSerially(). Finally, you draw the outline of the stopwatch by setting the color to black and calling drawArc():

```
public void paint (Graphics g) {
    g.setGrayScale (255);
    g.fillRect (0, 0, getWidth(), getHeight());

    if (degree > 0) {
        g.setColor (255, 0, 0);
        g.fillArc (0,0, getWidth(), getHeight(), 90, degree);
        display.callSerially (this);
    }
    g.setGrayScale (0);
    g.drawArc (0, 0, getWidth()-1, getHeight()-1, 0, 360);
}
```

This method is invoked by the event handling thread as a result of the previous display.callSerially(this) statement. In this case, it just calculates a new pie slice and requests a repaint():

```
    public void run() {
        int permille = (int) ((System.currentTimeMillis()
                              - startTime) / seconds);
        degree = 360 - (permille * 360) / 1000;
        repaint();
    }
}
```

As always, you need a MIDlet to actually display your `StopWatchCanvas` implementation. The following code creates a stopwatch set to 10 seconds when the application is started:

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

public class StopWatch extends MIDlet {
    public void startApp() {
        Display display = Display.getDisplay (this);
        display.setCurrent (new StopWatchCanvas (display, 10));
    }

    public void pauseApp() {
    }

    public void destroyApp (boolean forced) {
    }
}
```

Avoiding Flickering

On some devices, the stopwatch implementation will flicker. This is due to the fact that the display is cleared completely before a new stopwatch is drawn. However, on some other devices, the stopwatch will not flicker because those devices provide automated double buffering. Before the screen is updated, all drawing methods are performed in a hidden buffer area. Then, when the `paint()` method is finished, the complete display is updated from the offscreen buffer at once. The method `isDoubleBuffered()` in the `Canvas` class is able to determine whether the device screen is double buffered.

In order to avoid flickering of your animation in all cases, you can add your own offscreen image, which is allocated only if the system does not provide double buffering:

```
Image offscreen = isDoubleBuffered() ? null :
                      Image.createImage (getWidth(), getHeight());
```

In the `paint()` method, you just check if the offscreen image is not `null`, and if so, you delegate all drawing to your offscreen buffer. The offscreen buffer is then drawn immediately at the end of the `paint()` method, without first clearing the screen. Clearing the screen is not necessary in that case since the offscreen buffer was cleared before drawing and it fills every pixel of the display:

```
public void paint (Graphics g) {
    Graphics g2 = offscreen == null  ? g : offscreen.getGraphics();

    g2.setGrayScale (255);
    g2.fillRect (0, 0, getWidth(), getHeight());

    if (degree > 0) {
        g2.setColor (255, 0, 0);
        g2.fillArc (0,0, getWidth(), getHeight(), 90, degree);

        display.callSerially (this);
```

```
    }
    g2.setGrayScale (0);
    g2.drawArc (0, 0, getWidth()-1, getHeight()-1, 0, 360);

    if (offscreen != null)
        g.drawImage (offscreen, 0, 0, Graphics.TOP | Graphics.RIGHT);
}
```

Listing 3.3 gives the complete source code for the buffered stopwatch.

Listing 3.3 `BufferedStopWatch.java`—The Complete Source Code of the Buffered Stopwatch

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

class BufferedStopWatchCanvas extends Canvas implements Runnable {
    int degree = 360;
    long startTime;
    int seconds;
    Display display;
    Image offscreen;

    BufferedStopWatchCanvas (Display display, int seconds) {
        this.display = display;
        this.seconds = seconds;

        if (!isDoubleBuffered() && false)
            offscreen = Image.createImage (getWidth(), getHeight());

        startTime = System.currentTimeMillis();
    }

    public void paint (Graphics g) {

        Graphics g2 = offscreen == null
            ? g
            : offscreen.getGraphics();

        g2.setGrayScale (255);
        g2.fillRect (0, 0, getWidth(), getHeight());

        if (degree > 0) {
            g2.setColor (255, 0, 0);
            g2.fillArc (0,0, getWidth(), getHeight(), 90, degree);

            display.callSerially (this);
        }
        g2.setGrayScale (0);
        g2.drawArc (0, 0, getWidth()-1, getHeight()-1, 0, 360);

        if (offscreen != null)
            g.drawImage (offscreen, 0, 0, Graphics.TOP |
Graphics.RIGHT);
    }

    public void run() {
        int permille = (int) ((System.currentTimeMillis()
                               - startTime) / seconds);
        degree = 360 - (permille * 360) / 1000;
```

```
        repaint();
    }
}


public class BufferedStopWatch extends MIDlet {

    public void startApp() {
        Display display = Display.getDisplay (this);
        display.setCurrent (new BufferedStopWatchCanvas (display,
10));
    }

    public void pauseApp() {
    }

    public void destroyApp (boolean forced) {
    }
}
```

MIDP 2.0 Additions

Version 2.0 of MIDP improves the existing high-level user interface API significantly. Additionally, there are three new packages packages, `javax.microedition.lcdui.game`, `javax.microedition.media`, and `javax.microedition.media.control`, especially for game and multimedia programming.

LCDUI High-Level Improvements

For the high-level part of LCDUI, there are several significant additions and improvements:

- The new Class `CustomItem` allows user drawn items
- `Commands` can also be assigned to `Items`
- All items have a new `setLayout()` method

The `CustomItem` class allows you to combine the flexibility of the low-level API directly with the high-level API on the same screen. The ability to assign commands to items allows context-sensitive menus, depending on the current focus position. For example, this could be used for nested option dialogs or to build a hypertext browser based on the high-level API only. The `setLayout()` method gives MIDP 2.0 applications better control over the actual layout of the items contained in a form.

The Game Package

The new game package contains graphical objects specially designed for games and animations. The class `Sprite` is designed for adding active objects to games. Sprites can consist of a single image or a sequence of images representing an animated sprite. In contrast to plain images, sprites have status information such as the current position and animation frame. The `Sprite` class also contains methods for detecting collisions with other sprites.

In addition to the `Sprite`, the game package contains another graphical object, the `TiledLayer` class. Tiled layers are large images constructed from equally sized cells and can be used for implementing a scrolling screen background, for example. The cells are obtained from a single image that is divided into a number of rows and columns, as defined by the `TiledLayer` constructor. Each cell contains an index that points to a tile obtained from the image given to the constructor. When the `TiledLayer` is displayed on the screen, the cells are rendered from the tile corresponding to the index. While positive indices are direct pointers to a portion of the source image, negative indices are indirect pointers and can be remapped for animating a set of cells with the same virtual index at once. An index of zero represents a transparent cell.

Both classes, `Sprite` and `TiledLayer` are derived from the abstract class `Layer`. While instances of both classes can be drawn directly using their paint method, the game package also contains a `LayerManager` that can manage a set of graphical objects including their Z-Order.

Finally, the game package contains the class `GameCanvas`, a subclass of `Canvas`. Each instance of `GameCanvas` has its own offscreen buffer. In contrast to the simple `Canvas`, it is safe to assume that the `GameCanvas` offscreen buffer is not obscured. Moreover, the `GameCanvas` provides access to the corresponding `Graphics` object outside of the paint method and a method to flush the offscreen buffer to the screen.

The Media Packages

The `javax.microedition.media` package contains audio support and consists of three main building blocks. The `Manager` class can be seen as the entry point to media access. It is able to list the supported media types and create new `Player` instances. A player can be used to actually play a media object, such as a midi or mp3 file (depending on the supported media types).

The code snipped below shows an example of how to play a midi file from the Web using a player object obtained from the Manager.

```
try {
    Player p = Manager.createPlayer("http://ringtones.org/song.mid");
    p.start();
}
catch (MediaException pe) {
}
catch (IOException ioe) {
}
```

The `javax.microedition.media.control` package contains additional components for controlling the properties of the played media objects such as volume and pitch.

Summary

In this chapter, you learned the general life cycle of MIDP applications. You know how to build a user interface using the high-level `lcdui` widgets, and how to interact using the listener mechanism. You have learned to perform custom graphics using the low-level API, including device-independent flicker-free animation and coordination of graphics calculation and drawing.

The next chapter gives a corresponding overview of the PDAP life cycle and user interface. The PDAP introduction focuses on the differences between the J2SE AWT classes and the subset included in PDAP, but still gives a basic introduction to AWT programming.

Chapter 4. PDAP Programming

IN THIS CHAPTER

This chapter discusses the life cycle and user interface of PDA applications. First, we'll talk about the general design of PDA applications. Then, we'll explain the AWT subset that forms the PDAP user interface API.

PDAP Application Life Cycle

PDAP is a complete superset of MIDP. As for MIDP, PDAP applications are based on the MIDlet class and share the MIDlet life cycle, as illustrated in Figure 3.1. However, PDAP contains several additional packages. For example, the AWT classes allow much more sophisticated user interfaces than lcdui, giving the programmer fine-grained control over component layout.

In this chapter, we will concentrate on the user interface enhancements of PDAP.

`HelloPdap` Revisited

The `HelloPdap` example from Chapter 1, "Java 2 Micro Edition Overview," is already a complete PDAP application. Now that you have the necessary foundation, you can revisit `HelloPdap` from an API point of view.

First, you import the necessary `midlet` and `awt` packages:

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

Like all PDAP applications, the `HelloPdap` example is required to extend the `MIDlet` class:

```
public class HelloPdap extends MIDlet {
```

In the constructor, you create a `Frame` titled `"HelloPdap"`:

```
Frame frame;

public HelloPdap() {
   frame = new Frame ("HelloPdap");
}
```

You do not add content to the frame yet, so only the title will be displayed. (A description of the `awt` classes is contained in the "PDA User Interface" section.)

When your MIDlet is started the first time or when the MIDlet resumes from a paused state, the `startApp()` method is called by the program manager. Here, the `show()` method of the frame is called, requesting that the frame be displayed. If the frame is already visible, it is brought to the foreground:

```
public void startApp() {
   frame.show();
}
```

When the application is paused, you do nothing because you do not have any allocated resources to free. However, you need to provide an empty implementation, because implementation of `pauseApp()` is mandatory:

```
public void pauseApp(){
}
```

Like `startApp()` and `pauseApp()`, the implementation of `destroyApp()` is mandatory. Here, we dispose of the frame, freeing the associated system resources:

```
    public void destroyApp(boolean unconditional) {
        frame.dispose();
    }
}
```

PDA User Interface

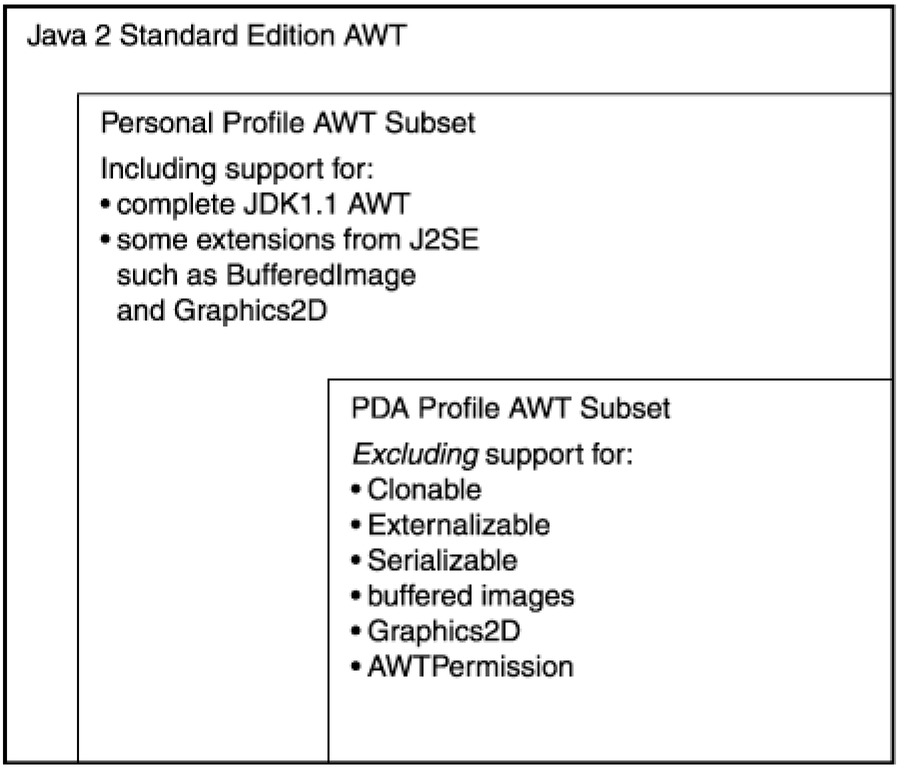
The PDA user interface is based on the J2SE Abstract Window Toolkit (AWT). This section of the chapter is structured as follows: First, we compare the PDA AWT subset with other AWT versions. Then, a short general introduction to the AWT programming is given. The sections "Custom Components" and "Custom Layout Managers" cover some aspects with special importance for programming limited devices. The section "Multiple Threads in the PDAP AWT Subset" shows how you can build multithreaded AWT programs, although the PDAP AWT subset is limited to a SWING-like single thread model. Finally, we'll implement an AWT application combining some of the discussed concepts.

Comparison of the PDAP AWT Subset to Other AWT Versions

The `java.awt` classes included in PDAP are based on the (CDC-based) Personal Profile AWT building block. The Personal Profile AWT is a clean subset of Java 2.0 AWT. It covers the complete Java 1.1 AWT, and contains several additions from Java 2, such as buffered images, support for thread-safe access and several convenience methods. Thread-safe AWT programming is discussed in more detail in the section "Multiple Threads in the PDAP AWT Subset." Additionally, some methods helping to reduce creation of intermediate objects introduced with Java 2.0 were taken over into the Personal Profile. Examples for this type of methods are `getWidth()`, which replaces a call to `getSize()`returning an intermediate dimension object.

Compared to the Personal Profile AWT, the PDAP AWT is slightly more restricted. The PDAP AWT subset contains only the packages `java.awt`, `java.awt.event`, and `java.awt.image`. Moreover, those packages are not complete; several classes and methods are missing. For example, more Java 2 additions such as buffered images and `Graphics2D` support were left out. The `Cloneable`, `Serializable`, and `Externalizable` interfaces are not supported in CLDC. Also, the `AWTPermission` was left out because the implied security model is not part of CLDC. A complete comparison of the PDAP AWT subset to the Java 2 AWT API can be found in Appendix B,"Comparison Charts." Figure 4.1 illustrates the relations between the different AWT versions.

Figure 4.1. Subset relations among the different AWT versions.



The PDAP AWT subset supports an unlimited number of frames, but the device might show only the top frame on the screen. The frames may be restricted to a fixed size. When using both LCDUI screens and AWT frames, the LCDUI display behaves similar to an additional frame. Mixing LCDUI and AWT isn't recommended, and it isn't possible to mix AWT and LCDUI components in a single `Frame` or `Screen`.

Note

PDA applications that don't rely on special microedition packages other than midlet can run on the desktop without an emulation environment by implementing a dummy `javax.microedition.midlet.MIDlet` class like that shown in the following code snippet:

```
package javax.microedition.midlet;

public class MIDlet {

    protected abstract void startApp();
    protected abstract void pauseApp();
    protected abstract void destroyApp (boolean unconditional);

    public void notifyDestroyed() {
        System.exit (0);
    }
}
```

In addition, you need to add a main method to the actual MIDlet implementation that creates a corresponding instance and calls the `startApp()` method.

The corresponding `main()` method for the `HelloPdap` example is

```
public static void main (String [] argv) {
    new HelloPdap().startApp();
}
```

A Short Introduction to AWT Programming

Although this book is targeted at programmers who have some basic Java experience, a short introduction to AWT programming is included here. The motivation is that modern Java desktop applications are mostly based on SWING, and servlets are based on an HTML interface, so a large fraction of Java programmers might not have much experience with AWT. Also, Rapid Application Development tools contained in several J2SE IDEs might not yet be able to generate PDAP-compatible code.

For a more detailed explanation of AWT, refer to a general Java book covering AWT or to the online tutorials provided by Sun http://java.sun.com/docs/books/tutorial/. An overview about AWT is available from Sun's AWT Web site at http://java.sun.com/products/jdk/awt/.

Basic Component Model

Graphical user interfaces usually consist of a number of components like buttons, text fields, or check boxes. In the Abstract Window Toolkit, these elements are represented by Java classes such as `Button`, `TextField`, and `Checkbox`. These classes handle the drawing of the corresponding components, as well as basic handling of user interactions. Most elements of the user interface are derived from the abstract base class `Component`. The `Component` class provides access methods for the common properties of the user interface classes, such as the size and position.

A special subclass of `Component` is `Container`, which can contain a set of other components—including containers. A special example of a container is a `Frame`. The `Frame` class represents a regular screen window. Other containers are `Panel`s, which are used for grouping components, and dialog windows.

Using a `Frame` and a `Label`, you are already able to build an AWT-based "Hello World" program (see Listing 4.1). In the constructor, a `Frame` with the title `"Hello World"` is created and then a `Label` showing `"Hello World"` is added to the `Frame`.

Listing 4.1 `ComponentSample.java`

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

public class ComponentSample extends MIDlet {
    Frame frame;

    public ComponentSample() {
        frame = new Frame ("Hello World");
        frame.add(new Label ("Hello World"));
    }


    public void startApp() {
        frame.show();
    }

    public void pauseApp() {
    }
    public void destroyApp (boolean conditional) {
        frame.dispose();
    }
}
```

User Interaction and Event Handling

The original `ComponentSample` MIDlet doesn't allow user interaction. However, some kind of exit button would be useful. Buttons are represented by the component class `Button`. The only parameter of the constructor is the button label text. The following lines add a button to the south area of the frame (the layout areas will be described in more detail in the next section):

```
Button b = new Button ("Exit");
frame.add(b, BorderLayout.SOUTH);
```

When the program is started with the additional lines, a button will appear at the bottom of the `Frame`, but nothing will happen when the button is clicked. The button still needs to be linked to the desired action, in this case leaving the program. In AWT, all kinds of user interface interactions—like clicking a button—are represented by event objects. In the case of a `Button`, the corresponding event object is an instance of `ActionEvent`. If an application is interested in some kind of event, it must implement the corresponding listener interface, containing one or more callback methods. For `ActionEvent`s, the listener interface is

```
public interface ActionListener {
   public void actionPerformed (ActionEvent e);
}
```

Both the event classes and listener interfaces are contained in the package `java.awt.event`.

In order to handle events, an object implementing the listener interface must be registered with the component that is the source of the events. For registering an action listener, the button class provides the method `addActionListener(ActionListener l)`.

In order to keep your sample program simple, you can add the `actionPerformed()` method to the `MIDlet` class directly and thus let it implement the `ActionListener` interface (see Listing 4.2).

Listing 4.2 `ComponentSample2.java`—Enhanced Version of the `ComponentSample` Handling User Interaction

```
import java.awt.*;
import java.awt.event.*;
import javax.microedition.midlet.*;

public class ComponentSample2 extends MIDlet implements
ActionListener {
    Frame frame;
    public ComponentSample2() {
        frame = new Frame ("Hello World");
        frame.add(new Label ("Hello World"));
        Button b = new Button ("Exit");
        b.addActionListener(this);
        frame.add(b, BorderLayout.SOUTH);
    }

    public void actionPerformed(ActionEvent ae) {
        frame.dispose();
        notifyDestroyed();
    }

    public void startApp() {
        frame.show();
    }
```

```
    public void pauseApp() {
    }

    public void destroyApp (boolean conditional) {
        frame.dispose();
    }
}
```

AWT contains lots of other active components and events. Introducing them all here would exceed the scope of this book. Table 4.1 gives an overview of the AWT components. Please note that the component events apply to all other (derived) components, although listed only once in the component description. Table 4.2 gives an overview of all events that are available in PDAP and the corresponding listener interfaces that are needed for handling those events.

Table 4.1. AWT Components and Corresponding Events

Component Name	***Events***	***Description***
`Button`	`ActionEvent`	The Button class is a component that is used to create a labeled button in order to invoke an action if it is pushed.
`Canvas`	None	The `Canvas` is a component for custom drawing by overriding the paint method.
`Checkbox`	`ItemEvent`	A `Checkbox` is a component that can be in either selected (true) or deselected (false) state. An `ItemEvent` is fired, when the `CheckBox` is (de)selected.
`CheckboxMenuItem`	`ItemEvent`	A `CheckboxMenuItem` represents one item of a menu combining the functionality of a `MenuItem` and a `Checkbox` as well. It can be in either selected (true) or deselected (false) state. An `ItemEvent` is fired, when the `CheckBox` is (de)selected.
`Choice`	`ItemEvent`	The `Choice` class presents a component for selecting one of a set of possible choices. An An `ItemEvent` is fired, when the `CheckBox` is (de)selected.

`Component`	`FocusEvent`
	`KeyEvent`
	`MouseEvent`
		The `Component` class is the abstract superclass of all nonmenu-related AWT components. `Component` can be derived directly to implement lightweight components.

`Container`	`ContainerEvent`	A `Container` is a specialized component that is used to hold other components. A `ContainerEvent` is fired if a `Component` is added or removed from a `Container`.
`Label`	None	A `Label` is a component for placing a single read-only line of text in a container.
`List`	`ItemEvent`	A `List` represents a scrolling list of text items for user selection. An `ItemEvent` is fired if the user selects an item in the List.
`MenuComponent`	None	The class `MenuComponent` is the superclass of all menu-related components.
`MenuItem`	`ActionEvent`	The class `MenuItem` represents one item of a menu. An `ActionEvent` is fired if the `MenuItem` is selected.

Scrollbar	AdjustmentEvent	A Scrollbar provides a convenient option to select a value from a given range. An AdjustmentEvent is fired if the value of the Scrollbar component is adjusted.
TextComponent	TextEvent	The TextComponent class is the superclass of any component that allows text input. A TextEvent is fired if text in a Component derived from TextComponent is changed.
TextArea	None	The TextArea is a component capable of displaying a multiline region of text. If the text inside the TextArea changes, a TextEvent is fired.
TextField	ActionEvent	The TextField is a component allowing the user to edit a single line of text. If the text inside the TextField changes, a TextEvent is fired. An ActionEvent is fired if the text is confirmed by a return keystroke.

Table 4.2. Available AWT Events and Corresponding Listeners

Event Name	*Listener*	*Description and Interface Methods*
ActionEvent	ActionListener	Invoked by a specific component to indicate that a component-specific action occurred. Classes implementing an ActionListener in order to receive ActionEvents need to implement the following method: void actionPerformed(ActionEvent e)
AdjustmentEvent	AdjustmentListener	Invoked by components indicating that their value has been adjusted. Classes implementing an AdjustmentListener in order to receive AdjustmentEvents need to implement the following method: void adjustmentValueChanged (AdjustmentEvent e)
ComponentEvent	ComponentListener	Invoked by componentes in order to indicate that a component has moved, changed its size or changed its visibility.
ContainerEvent	ContainerListener	Invoked in order to indicate that a component has beed added or removed.
FocusEvent	FocusListener	Invoked by components in order to indicate that they have gained or lost the keyboard focus. Classes implementing a FocusListener in order to receive FocusEvents need to implement the following methods: void focusGained(FocusEvent e) void focusLost(FocusEvent e)
ItemEvent	ItemListener	Invoked by ItemSelectable components in order to indicate that an item is selected or deselected. Classes implementing an ItemListener in order to receive ItemEvents need to implement the following

		method: `void itemStateChanged(ItemEvent e)`

`KeyEvent`	`KeyListener`	Indicates that a keystroke has occurred in a component that is capable of accepting keystrokes. Classes implementing a `KeyListener` in order to receive `KeyEvents` need to implement the following methods:
		`void keyPressed(KeyEvent e)`
		`void keyReleased(KeyEvent e)`
		`void keyTyped(KeyEvent e)`

`MouseEvent`	`MouseListener/ MouseMotionListener`	Indicates that a mouse action occurred in a component. The `MouseEvent` is used both for mouse events (click, enter, exit) and mouse motion events (moves and drags). The only difference is where the events are indicated.
		
		Clicks, presses, and releases are received by a `MouseListener` consisting of the the following methods:
		
		`void mouseClicked(MouseEvent e)`
		`void mouseEntered(MouseEvent e)`
		`void mouseExited(MouseEvent e)`
		`void mousePressed(MouseEvent e)`
		`void mouseReleased(MouseEvent e)`
		
		Classes that are interested in receiving mouse drags and moves need to implement the `MouseMotionListener` and the following methods:
		
		`void mouseDragged(MouseEvent e)`
		`void mouseMoved(MouseEvent e)`

`PaintEvent`	None	In contrast to the other events, the paint event has no corresponding listener or adapter. The `PaintEvent` is internally handled by the event queue when a component needs to be repainted. Applications should override the paint/update methods in order react to repaint events.
`TextEvent`	`TextListener`	Indicates that the text of a component has been changed. Classes implementing a `TextListener` in order to receive `TextEvents` need to implement the following method: `void textValueChanged(TextEvent e)`
`WindowEvent`	`WindowListener`	Indicates that a window has changed its status. Classes implementing a `WindowListener` in order to receive `WindowEvents` need to implement the following methods:

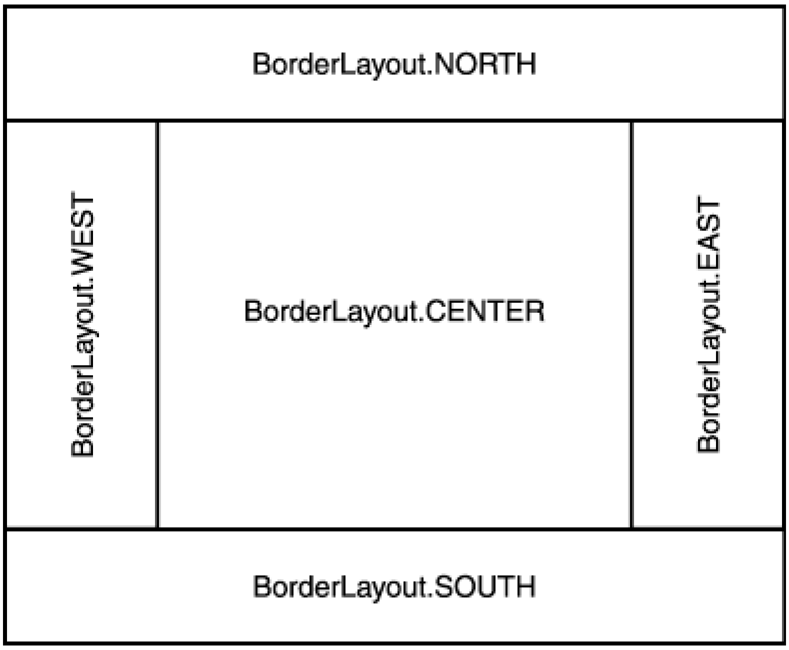
		void windowActivated(WindowEvent e) void windowClosed(WindowEvent e) void windowClosing(WindowEvent e) void windowDeactivated(WindowEvent e) void windowDeiconified(WindowEvent e) void windowIconified(WindowEvent e) void windowOpened(WindowEvent e)

Containers and Layout

As soon as the user interface consists of more than one or two components, the screen layout of the components becomes an issue. For some devices, it would be possible to place the components at fixed pixel positions. However, for portable Java applications, this approach isn't really suitable. In addition to the different screen sizes and layouts, the components might also have different sizes on different PDAs.

Fortunately, Java provides a powerful mechanism to cope with layout problems: Layout Managers. Layout Managers can be assigned to subclasses of `Container` such as `Frame`, `Dialog`, or `Panel` using the `setLayoutManager()` method. Layout Managers place the components contained in the assigned `Container` with respect to special rules determined by the concrete subclass of `LayoutManager` used. For example, `GridLayout` arranges all contained components in a grid where each component has exactly the same size. `BorderLayout` divides the `Container` into five regions as shown in Figure 4.2. The desired region is given as a parameter to the `add()` method when adding components to the container. The border region takes the minimum space that is required for displaying the contained components, and the remainder goes to the center area. The `BorderLayout` is the default layout manager of `Dialog`s and `Frame`s. The `FlowLayout` arranges all components in a horizontal line where the width of the components is minimized and the height is aligned to the minimal height of the highest component. `FlowLayout` is the default layout manager of Panel.

Figure 4.2. The five regions of the `BorderLayout`.



Although the layout managers provide basic layout options, quite flexible layouts can be achieved by nesting containers and thus combining layout managers. For example, in order to show a list of aligned labels and input elements, two panels with a grid layout can be placed in the west and center areas of a border layout. If the number of labels in the west grid matches the number of input components in the center grid, the labels and input components are vertically aligned automatically. Furthermore, the space for the labels is limited to the minimum, whereas the input components get all the remaining space. Dialog buttons are usually placed in a panel with flow layout in the south area of the dialog.

As a sample application demonstrating nested `Panel`s (and `BorderLayout` and `FlowLayout`), you will implement a PDA user interface for the British Museum Algorithm. Basically, the idea behind the British Museum Algorithm is that, given a set of monkeys typing on typewriters, eventually all existing literature in the world would be generated—you just need to wait long enough. We simulate a monkey typing a line by randomly generating characters. The user interface should consist of a button for generating a sentence, an exit button, and a list of sentences generated so far. How can we distribute the screen space in a way that the buttons are displayed and all the remaining space goes to the list of sentences? The answer is quite simple: We put the list in the center of a `BorderLayout`, which gets all remaining space not taken by the other regions. In the south area, we put a `Panel` with `FlowLayout`, where we add the buttons. Fortunately, `BorderLayout` is the default layout of frames and `FlowLayout` is the default layout of `Panel`s, so we do not need to explicitly set a layout. (We will do so in an extended example.)

Listing 4.3 shows the implementation of the application. The components are created with the corresponding variables and arranged in the constructor of the MIDlet. The call of the `pack()` method performs the layout of the components and adjustment of the frame size. (Some PDAs might have a fixed frame size covering the whole screen.) The `actionPerformed()` method terminates the application or generates a new sentence by calling `generateSentence()`, depending on the button pressed. Figure 4.3 shows the actual layout of the application on a Palm Pilot.

Figure 4.3. The typing monkeys application.

Listing 4.3 `GhostWriter.java`—The Typing Monkeys Source Code

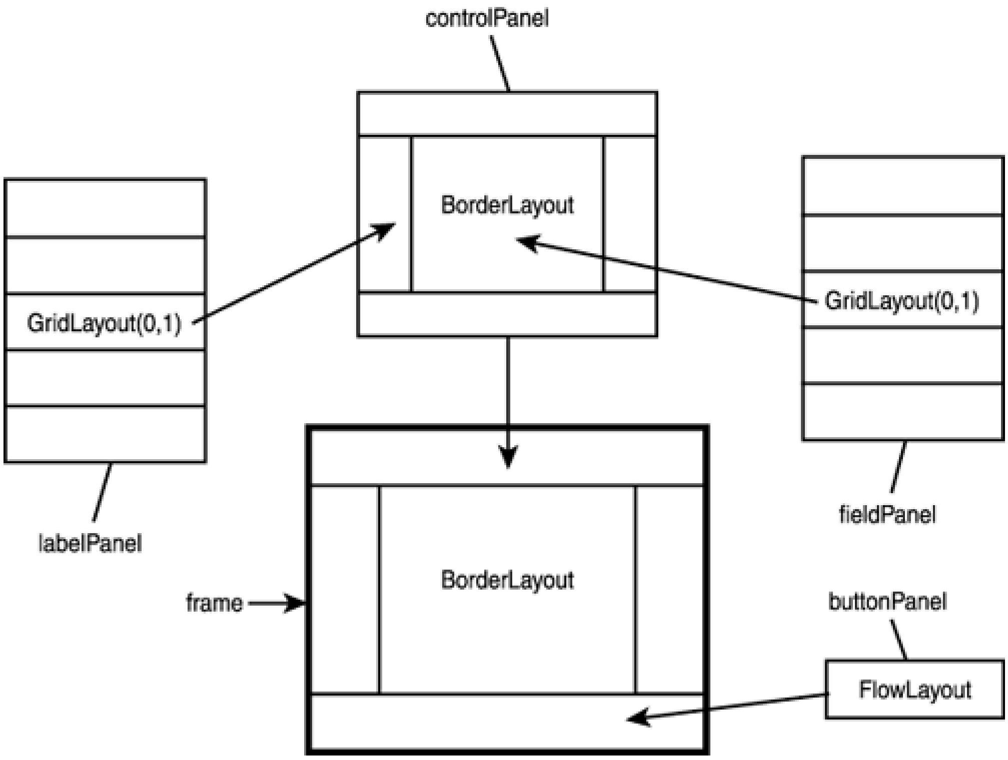
```
import java.awt.*;
import java.awt.event.*;
import java.util.Random;
import javax.microedition.midlet.*;

class GhostWriter extends MIDlet implements ActionListener {

    List list = new List();
    Frame frame = new Frame ("The Monkeys type....");
```

```
    Button exitButton = new Button ("exit");
    Button generateButton = new Button ("generate");
    Random random = new Random();

    public GhostWriter() {

        Panel buttonPanel = new Panel();

        buttonPanel.add(exitButton);
        buttonPanel.add(generateButton);
        exitButton.addActionListener(this);
        generateButton.addActionListener(this);

        frame.add(list, BorderLayout.CENTER);
        frame.add(buttonPanel, BorderLayout.SOUTH);
        frame.pack();

        frame.addWindowListener(new WindowAdapter() {
                public void windowClosing (WindowEvent ev) {
                    notifyDestroyed();
                }
            } );
    }


    public void startApp() {
        frame.show();
    }


    public void actionPerformed (ActionEvent ev) {

        if (ev.getSource() == exitButton)
            notifyDestroyed();
        else if (ev.getSource() == generateButton)
            generateNewSentence();
    }

    public void generateNewSentence() {
        StringBuffer buf = new StringBuffer();

        for (int i = 0; i < 60; i++) {
            char c = (char) (((int) 'a') + (random.nextInt() & 31));
            if (c > 'z') c = ' ';
            buf.append (c);
        }

        list.add(buf.toString());
    }

    public void pauseApp() {
    }

    public void destroyApp (boolean forced) {
    }
}
```

Another common layout is, as we mentioned earlier, a list of labels and corresponding fields, such as the input mask of an address book. Suppose that we want to log only the good sentences typed by the simulated monkeys because of the limited memory of a PDA. Additionally, we have two monkeys to choose from. In order to display these options, we would like to show a label "generated," a

TextField with the generated sentence, a label "monkey," and a choice containing the names of the monkeys, as shown in Figure 4.4.

Figure 4.4. The nested layouts used in the improved version of the typing monkeys application.



In order to achieve this layout, we create a new control panel having the layout set to BorderLayout. In the east area of the new control panel we put a grid-layout label panel, and in the center area we put a grid-layout field panel. Because both subpanels will contain the same number of components and GridLayout distributes space equally between the components, the components will be aligned vertically as desired. The new control panel is then inserted in the north area of the application Frame.

Listing 4.4 shows our enhanced typing monkeys application. The new layout code is marked bold in order to highlight the interesting additions to the previous example. Figure 4.5 shows the complete user interface of the extended application.

Figure 4.5. The improved version of the typing monkeys application.

Listing 4.4 `GhostWriter2.java`—Enhanced Typing Monkeys Source Code

```
import java.awt.*;
import java.awt.event.*;
import java.util.Random;
import javax.microedition.midlet.*;

public class GhostWriter2 extends MIDlet implements ActionListener {

    List logList = new List();
    Frame frame = new Frame ("The Monkeys type....");
    Button exitButton = new Button ("exit");
    Button generateButton = new Button ("generate");
    Button logButton = new Button ("log");
    TextField generatedField = new TextField();
    Choice monkeyChoice = new Choice();
    Random random = new Random();


    public GhostWriter2() {

        monkeyChoice.add("Dumbo");
        monkeyChoice.add("Sally");

        Panel labelPanel = new Panel (new GridLayout (0, 1));
        labelPanel.add(new Label ("Monkey:"));
        labelPanel.add(new Label ("Generated:"));

        Panel fieldPanel = new Panel (new GridLayout (0, 1));
        fieldPanel.add(monkeyChoice);
        fieldPanel.add(generatedField);

        Panel controlPanel = new Panel (new BorderLayout());
        controlPanel.add(labelPanel, BorderLayout.WEST);
        controlPanel.add(fieldPanel, BorderLayout.CENTER);

        Panel buttonPanel = new Panel (new FlowLayout());
        buttonPanel.add(exitButton);
        buttonPanel.add(generateButton);
        buttonPanel.add(logButton);

        exitButton.addActionListener(this);
        generateButton.addActionListener(this);
```

```
        logButton.addActionListener(this);

        frame.add(controlPanel, BorderLayout.NORTH);
        frame.add(logList, BorderLayout.CENTER);
        frame.add(buttonPanel, BorderLayout.SOUTH);
        frame.pack();

        frame.addWindowListener(new WindowAdapter() {
                public void windowClosing (WindowEvent ev) {
                    notifyDestroyed();
                }
            } );
    }

    public void startApp() {
        frame.show();
    }

    public void actionPerformed (ActionEvent ev) {

        if (ev.getSource() == exitButton)
            notifyDestroyed();
        else if (ev.getSource() == generateButton)
            generateNewSentence();
        else if (ev.getSource() == logButton)
            logList.add(generatedField.getText());
    }

    public void generateNewSentence() {
        StringBuffer buf = new StringBuffer();

        int baseChar = monkeyChoice.getSelectedItem().equals ("Dumbo")
            ? 'A' : 'a';

        for (int i = 0; i < 60; i++) {
            char c = (char) (baseChar + (random.nextInt() & 31));
            if (Character.toUpperCase (c) > (baseChar+26)) c = ' ';
            buf.append (c);
        }
        generatedField.setText (buf.toString());
    }

    public void pauseApp() {
    }

    public void destroyApp (boolean forced) {
    }
}
```

When you're designing PDA applications, saving the limited screen space is especially important. For cases in which the screen is too small to hold all the required information, the `ScrollPane` and `CardLayout` classes are provided. The `CardLayout` class is a layout manager that allows distribution of components over several cards. The cards are denoted by a `String`, and only one card is shown at once. For example, using a `Choice` component, the user can be allowed to switch between cards. The `ScrollPane` can contain an area that is larger than its space on the screen by using scrollbars for navigation. Both classes are described in more detail in the section "Switching Layouts Depending on the Screen Resolution Available."

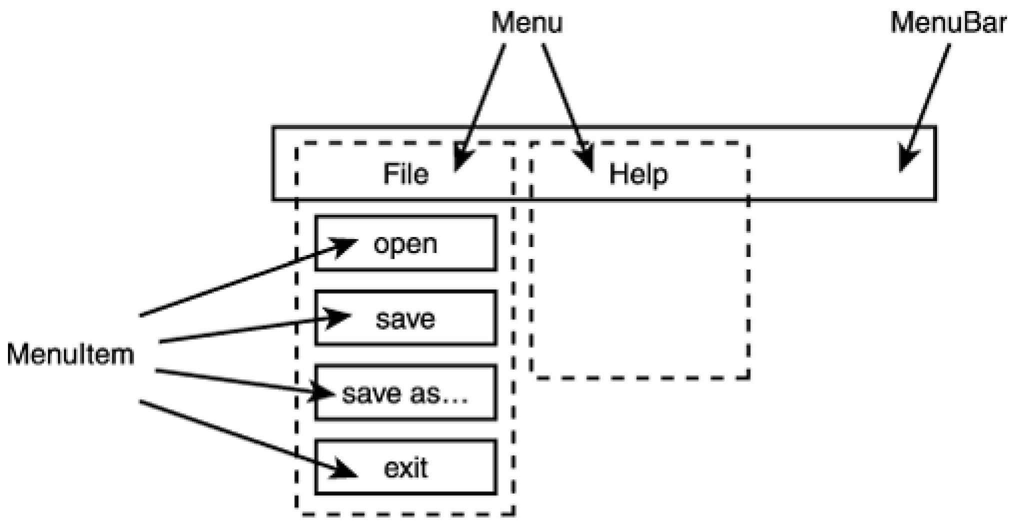
Dialogs and Menus

In larger applications, the user interface might become overloaded with buttons and other widgets for invoking different program actions. In order to minimize the space that is occupied by those widgets, pull-down menus can be used instead, especially for less important actions or configuration options. This approach gives you the opportunity to add many menu items to a menu without wasting the scarce screen space of a mobile device. An AWT menu consists of at least three classes:

- One `MenuBar` containing a set of `Menu`s
- Some `Menu`s consisting of multiple `MenuItem`s
- `MenuItem`s sending `ActionEvent`s to registered `Listener`s

This coherence is illustrated in Figure 4.6.

Figure 4.6. A `MenuBar` consisting of two `Menus` where the File menu holds some `MenuItems`.



Another opportunity to save screen space is to move widgets to dialogs. `Dialog` is a class similar to `Frame`. Both classes are direct subclasses of the `Window` class. Like a frame, a dialog represents a rectangular area of the screen and can hold a set of widgets. In contrast to frames, dialogs can be modal and are designed as a temporary display for obtaining user input or similar purposes. A dialog needs to have a frame as a parent.

In order to show how menus and dialogs can be used in a PDAP application, we create a small shopping chart application, showing a list consisting of Amount and Item columns (see Listing 4.5).

For displaying the list, we use a `Panel` with `GridLayout`. In order to get as much space as possible for the user data, we create a separate dialog class for adding new rows, `InsertItemDialog`. The dialog is invoked by selecting the menu item Insert of the Item menu. As already illustrated in Figure 4.6, there is a 1-to-n relation between `MenuBar`, `Menu`, and `MenuItem`. The following code snippet creates a `MenuBar`, registers the `MenuItem` insert with the `ActionListener` of the `Frame`, and finally concatenates the `MenuBar` with the frame:

```
MenuBar menuBar = new MenuBar();
Menu menu = new Menu ("Items");
MenuItem insertItem = new MenuItem ("insert");
insertItem.addActionListener(this);
menu.add(insertItem);
menuBar.add(menu);
frame.setMenuBar (menuBar);
```

If the user selects the menu item insert, the `actionPerformed()` method of the registered handler is called, where the generated `ActionEvent` is handled. In the `actionPerformed()` method, a dialog will be shown enabling the user to add text into two columns labeled Amount and Item. The following code snippet is responsible for creating an instance of our `InsertItemDialog`, showing the dialog and adding the results to the main panel:

```
public void actionPerformed(ActionEvent ae) {
    InsertItemDialog dialog = new InsertItemDialog(frame);
    dialog.show();
    textPanel.add(new Label (dialog.getAmount()));
    textPanel.add(new Label (dialog.getItem()));
    frame.validate();
}
```

Because the dialog is modal, it will block the event handler of the frame until the dialog is dismissed using the OK button. In this case, the event handler switches back to the frame. To make sure that the new content of the grid layout is arranged properly, we call the `invalidate()` method of the frame.

Listing 4.5 `ShoppingChart.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.microedition.midlet.*;

public class ShoppingChart extends MIDlet implements ActionListener {

    class InsertItemDialog extends Dialog implements ActionListener {

        TextField amount = new TextField();
        TextField item = new TextField();

        public InsertItemDialog (Frame owner) {
            super (owner, "Insert Item", true);

            Panel panel = new Panel (new GridLayout (2, 0));
            panel.add(new Label ("Amount"));
            panel.add(amount);
            panel.add(new Label ("Item"));
            panel.add(item);

            add(panel, BorderLayout.CENTER);

            Panel buttonPanel = new Panel (new FlowLayout());
            Button b = new Button ("ok");
            b.addActionListener(this);
            buttonPanel.add(b);
            add(buttonPanel, BorderLayout.SOUTH);
            pack();
        }

        public String getAmount() {
            return amount.getText();
        }

        public String getItem() {
            return item.getText();
        }

        public void actionPerformed (ActionEvent ae) {
            setVisible (false);
```

```
        }
    }

    Panel textPanel;
    Frame frame;

    public ShoppingChart() {
        frame = new Frame("Shopping Chart");

        MenuBar menuBar = new MenuBar();
        Menu menu = new Menu ("Items");
        MenuItem insertItem = new MenuItem ("insert");
        insertItem.addActionListener(this);
        menu.add(insertItem);
        menuBar.add(menu);
        frame.setMenuBar (menuBar);

        textPanel = new Panel(new GridLayout (0, 2));
        frame.add(textPanel, BorderLayout.NORTH);

        frame.addWindowListener(new WindowAdapter() {
                public void windowClosing (WindowEvent e) {
                    destroyApp(true);
                }
            } );

        frame.pack();
    }
    public void actionPerformed(ActionEvent ae) {
        InsertItemDialog dialog = new InsertItemDialog(frame);
        dialog.show();
        textPanel.add(new Label (dialog.getAmount()));
        textPanel.add(new Label (dialog.getItem()));
        frame.validate();
    }

    public void startApp() {
        frame.show();
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
        notifyDestroyed();
    }
}
```

Figure 4.7 shows a shopping chart that is displayed using the `GridLayout` of the `java.awt` package.

Figure 4.7. The `ShoppingChart` application showing a shopping list layout using the `GridLayout`.

Note

This introduction gives a rough overview of the PDAP subset of AWT only. Appendix B contains a general comparison of the J2SE AWT and the AWT subset contained in PDAP.

Custom Components

Compared to SWING, the component set of AWT is quite limited. Most additional third-party AWT components probably won't fit to the constraints of the subset contained in the PDA Profile in most cases. In order to close this gap, it is often necessary to create custom components.

In order to be prepared for this task, you need to recall the main functionality that is provided by a component:

- Displaying the component on the screen
- Handling user events

In this section, we will start with a passive component not handling user events. Implementing a progress bar component, we describe all steps necessary to display a custom component on the screen. After we have created the non-interactive progress bar component, we will implement an image button capable of handling user events.

The `Component` class is the abstract super class of all non-menu related AWT components. It can be extended directly to create a customized component necessary for our progress bar and image button component as well.

The `ProgressBar` should provide a graphical representation of integer values in a bar graph style, comparable with a non-interactive `Gauge` of the MIDP `lcdui` package.

Listing 4.6 contains the source code of our `ProgressBar` implementation. The current progress value in the range from 0 to 100 is stored in the `value` variable. The method `setValue()` is used to change the state of the `ProgressBar` during application runtime.

In order to create a component that is capable of displaying itself, we need to overwrite the paint method. The paint method is responsible for drawing the component itself. In the progress bar `paint()` method, we use the given graphic object to draw a progress bar depending on the value variable and the current size of the component.

Most custom components need to overwrite two additional methods that are important for the appropriate layout of the component. The methods `getMinimumSize()` and `getPreferredSize()` are used by layout managers to query the size information of the component. Overwriting these methods with custom implementations makes sure that the component is displayed in an appropriate size.

Listing 4.6 ProgressBar.java

```
import java.awt.*;

public class ProgressBar extends Component {
    int currentValue = 0;
    final int MAX_VALUE = 100;


    public ProgressBar() {
    }

    public void setValue (int currentValue) {
        if (currentValue >= 0
            && currentValue <= MAX_VALUE) {
            this.currentValue = currentValue;
        }
    }


    public Dimension getPreferredSize() {
        return new Dimension (100, 20);
    }


    public Dimension getMinimumSize() {
        return new Dimension (10, 10);
    }


    public void paint (Graphics g) {

        Dimension dim = getSize();
        int progressPosition = (dim.width-4) * currentValue /
MAX_VALUE;

        g.setColor (Color.black);
        g.drawRect (0, 0, dim.width-1, dim.height-1);

        g.setColor (Color.white);
        g.drawRect (1, 1, dim.width-3, dim.height-3);

        g.setColor (SystemColor.activeCaption);

        g.fillRect (2, 2, progressPosition, dim.height-4);
        g.setColor (Color.white);
        g.fillRect (progressPosition + 2, 2,
                    dim.width - progressPosition - 4, dim.height-4);
    }
}
```

For testing purposes, we provide a small application shown in Listing 4.7 to show how the `ProgressBar` can be integrated into a MIDlet; the application is shown in Figure 4.8. The application just provides a `Scrollbar` for setting the current progress value and the `ProgressBar` itself. Real applications using the progress bar will probably be multithreaded. Note that for thread-safe access to AWT, it is necessary to use the `EventQueue.invokeLater()` method, described in the section "Multiple Threads in the PDAP AWT Subset."

Figure 4.8. The `ProgressTest` application for testing the `ProgressBar` component.

Listing 4.7 `ProgressTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.microedition.midlet.*;

public class ProgressTest extends MIDlet implements
AdjustmentListener {

    Frame frame;
    ProgressBar pBar = new ProgressBar();
    Scrollbar sBar = new Scrollbar();

    public ProgressTest() {
        frame = new Frame("ProgressBar Test");

        sBar = new Scrollbar (Scrollbar.HORIZONTAL, 0, 10, 0, 100);
        sBar.addAdjustmentListener(this);

        frame.add(pBar, BorderLayout.NORTH);
        frame.add(sBar, BorderLayout.SOUTH);
        frame.pack();
    }


    public void adjustmentValueChanged(AdjustmentEvent e) {
        pBar.setValue (sBar.getValue());
        pBar.repaint();
    }

    public void startApp() {
        frame.show();
    }

    public void pauseApp() {
    }
```

```
    public void destroyApp(boolean unconditional) {
        notifyDestroyed();
    }
}
```

After the implementation of the `ProgressBar`, we will now focus on an active component that is capable of handling user events. In order to show how this can be achieved, we will implement a button that displays an image instead of a text label. When the button is pressed by tapping the stylus on it, it will create an `ActionEvent` and send it to all registered listeners.

As in the `ProgressBar` implementation, we derive our `ImageButton` component from the `Component` class. Listing 4.8 contains the source code of the `ImageButton`. The `paint()`, `getPreferredSize()`, and `getMinimumSize()` methods correspond to their counterparts of the `ProgressBar` implementation. The constructor takes an image object and a command string and stores the parameters in object variables.

For handling user input, one possibility would be to register a mouse listener. However, for custom components, it is more appropriate to overwrite the `processMouseEvent()` method, which receives all mouse events when enabled. Enabling the events roughly corresponds to the registration process of the listener interface. This is done by calling the `enableEvents()` method of the `Component` class with the parameter `AWTEvent.MOUSE_EVENT_MASK`. Without this call, our implemented `processMouseEvent()` method would never be called by the event handler.

In addition to handling mouse events, it is necessary to enable users of the component to register action listeners, just like for regular buttons. Thus, the methods `addActionListener()` and `removeActionListener()` must be provided. The `listeners` variable contains a `Vector` that keeps track of the listeners registered with the `ImageButton` component.

In the `processMouseEvent()` method, a check for the event ID `MOUSE_CLICKED` is performed. If the mouse event passes the test, an `ActionEvent` object is created and sent to the `actionPerformed()` methods of all registered listeners by iterating through the listeners.

Listing 4.8 `ImageButton.java`

```
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

public class ImageButton extends Component  {

    Image image;
    String command;
    Vector listeners = new Vector();

    public ImageButton (Image image, String command) {
        this.image = image;
        this.command = command;

        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
    }


    public void addActionListener(ActionListener listener) {
        listeners.addElement (listener);
    }
```

```
    public void removeActionListener(ActionListener listener) {
        listeners.removeElement (listener);
    }


    public void paint (Graphics g) {
        Dimension dim = getSize();
        int w = image.getWidth (this);
        int h = image.getHeight (this);
        g.drawImage (image,
                     (dim.width - w) / 2,
                     (dim.height - h) / 2,
                     this);
        Color fg = g.getColor();
        g.setColor (fg);
        g.drawRoundRect (0, 0, dim.width-1, dim.height-1, 4, 4);
    }


    public Dimension getMinimumSize() {
        return new Dimension (image.getWidth (null),
                              image.getHeight (null));
    }


    public Dimension getPreferredSize() {
        return getMinimumSize();
    }


    public void processMouseEvent(MouseEvent e)  {

        if (e.getID() == MouseEvent.MOUSE_CLICKED) {

            ActionEvent ae = new ActionEvent
                (this, ActionEvent.ACTION_PERFORMED, command);

            for (int i = 0; i < listeners.size(); i++)
                ((ActionListener) listeners.elementAt (i))
                    .actionPerformed (ae);
        }

        super.processMouseEvent(e);
    }
}
```

For testing purposes, we provide a small application shown in Listing 4.9 to show how the `ImageButton` can be integrated into a MIDlet. The application uses an `ImageButton` that is registered to an `ActionListener` of the `Frame`. When the `ImageButton` is clicked, we increment a `clickCount` variable and display its value in the frame title.

Listing 4.9 `ImageTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.microedition.midlet.*;

public class ImageTest extends MIDlet implements ActionListener {
```

```
    Frame frame;
    int clicks = 0;
    ImageButton button;


    public ImageTest() {
        frame = new Frame("ImageButton Test");

        button = new ImageButton
            (Toolkit.getDefaultToolkit().getImage ("logo.png"),
"logo");
        button.addActionListener(this);

        frame.add(button, BorderLayout.CENTER);
        frame.pack();
    }


    public void actionPerformed(ActionEvent ae) {
        frame.setTitle ("clicks: " + (++clicks));
    }


    public void startApp() {
        frame.show();
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
        notifyDestroyed();
    }
}
```

Figure 4.9 shows the `ImageTest` application containing the `ImageButton` component created in this section.

Figure 4.9. The `ImageTest` application for testing the `ImageButton` component.

Switching Layouts Depending on the Screen Resolution Available

The screen sizes of PDAs differ significantly between different models. Figure 4.10 illustrates a set of different PDAs from various vendors showing diverse screen formats and orientations. The Nokia 9210 has a horizontal display with a resolution of 640x240 pixels. Regular Palms have a screen resolution of 160x160 pixels. The HandEra 330 has a vertical screen format with a resolution of 240x320 pixels similar to the screen of most PocketPCs such as the Compaq IPaq.

Figure 4.10. PDAs with different screen sizes available on the market.

In some cases, to make optimal use of the screen space available, a layout that dynamically adopts to the screen resolution available might make the most sense. In order to demonstrate this concept, assume that we would like to display two components with a fixed size. Depending on the screen format, there might be space enough for both components, or only one component might fit on the screen. If both components fit, they might fit only if horizontally or vertically arranged. It might also be possible that not even a single component will fit in the available space. Clearly, we want to make optimal use of the screen space available, so if both components fit, both should be displayed. If one component fits, we need a control to switch the displayed component, similar to a `JTabbedPane` in Swing. Finally, if not even one component fits, a scrollbar should allow the user to select a region of the component(s) to be displayed.

How can this goal be achieved? It isn't as difficult as it might seem. Fortunately, there is a special method `doLayout()` that the AWT system calls whenever an arrangement of a component is required. So it is possible to overwrite this method, look at the space available, and then arrange the child components accordingly.

For the example code, assume that `width` and `height` are the actual dimensions of the components to be displayed. The following subclass of `Panel` arranges the two components `rect1` and `rect2` with respect to the `width` and `height` variables as described in the previous paragraph.

We start the implementation of `doLayout()` by removing all contained components, disabling a choice control for switching between the images, and storing the actual dimensions in a local variable `d`:

```
class DynPanel extends Panel {

      public void doLayout() {
          removeAll();
          cardChoice.setEnabled (false);
          Dimension d = getSize();
```

Now we can figure out if enough space is available for displaying at least one of the images by comparing the actual size with the image width and height. If one image fits, further tests are performed to determine whether both images can fit in the space available. If so, the images are arranged accordingly by setting the layout manager to a corresponding grid layout, and both images, `rect1` and `rect2`, are added to the panel:

```
if (d.width >= width && d.height >= height) {

    boolean dw = d.width >= 2 * width;
    boolean dh = d.height >= 2 * height;
            if (dw | dh) {
                if ((dw && !dh) || (dw && dh && d.width > d.height))
                    setLayout (new GridLayout (1, 2));
                else
                    setLayout (new GridLayout (2, 1));

                add(rect1);
                add(rect2);
            }
```

If there is enough space for one image but not enough for both of them, only one is displayed, depending on the state of a choice in the main frame. Also, the choice is enabled in order to let the user select the component she would like to view:

```
    else {
        setLayout (new BorderLayout());
        add(cardChoice.getSelectedIndex() == 0 ?
             rect1 : rect2);
        cardChoice.setEnabled (true);
    }
}
```

Finally, if not even one image fits in the space available, we create a `ScrollPane` that contains the two child components in a subpanel with a grid layout:

```
    else {
        setLayout (new BorderLayout());
        ScrollPane scroll = new ScrollPane();
        Panel panel = new Panel (new GridLayout (2, 1));
        panel.add(rect1);
        panel.add(rect2);
        scroll.add(panel);
        add(scroll);
    }

    super.doLayout();
}
```

For the case in which the application is running on a system with flexible window sizes, we need to add a method returning the desired size of the dynamic panel. Here, the dimensions allowing a horizontal arrangement of the child components are returned:

```
    public Dimension getPreferredSize() {
        return new Dimension (2 * width, height);
    }
}
```

Listing 4.10 contains the code for the complete `DynLayout` example. The inner class `RectComponent` is a custom component that is inserted into the `DynPanel`, showing the effects. It just draws an empty or filled rectangle, depending on the `boolean` value given to the constructor. Besides the `DynLayout` with the two instances of `RectComponent` (`rect1` and `rect2`), the application frame contains text fields for setting the width and height variables. These are useful for demonstrating the effects of the dynamic layout because the screen size itself is usually fixed. The constructor of the `DynLayout` arranges the control components and sets the listener. The `itemSelected()` method is responsible for changing the displayed `RectComponent` if the choice control is switched.

Although this example is limited to a special case, it serves as a demonstration for some principles of dynamic layout. Feel free to extend it as needed, for example by querying the actual sizes of child components instead of using the width and height variables or by adding support for more than two child components.

Listing 4.10 `DynLayout.java`

```
import java.awt.*;
import java.awt.event.*;

import javax.microedition.midlet.*;


public class DynLayout extends MIDlet implements ActionListener,
ItemListener {

    int width = 100;
    int height = 100;

    Frame frame = new Frame ("DynLayout");
    TextField widthField = new TextField ("100");
    TextField heightField = new TextField ("100");
    Choice cardChoice = new Choice();
    Button applyButton = new Button ("Apply");
    DynPanel dynPanel = new DynPanel();

    class RectComponent extends Component {

        boolean fill;

        public RectComponent (boolean fill) {
            this.fill = fill;
        }
        public void paint (Graphics g) {
            Dimension d = getSize();

            g.setColor (Color.black);
            if (fill)
                g.fillRect ((d.width - (width - 5)) / 2,
                            (d.height - (height - 5)) / 2,
                            width - 5, height - 5);
            else
                g.drawRect ((d.width - (width - 5)) / 2,
                            (d.height - (height - 5)) / 2,
                            width - 5, height - 5);
        }

        public Dimension getPreferredSize() {
            return new Dimension (width, height);
        }

        public Dimension getMinimumSize() {
```

```
            return new Dimension (width, height);
        }
    }

    RectComponent rect1 = new RectComponent (false);
    RectComponent rect2 = new RectComponent (true);


    class DynPanel extends Panel {

        public void doLayout() {

            removeAll();
            cardChoice.setEnabled (false);
            Dimension d = getSize();

            if (d.width >= width && d.height >= height) {

                boolean dw = d.width >= 2 * width;
                boolean dh = d.height >= 2 * height;

                if (dw | dh) {
                    if ((dw && !dh) || (dw && dh && d.width >
d.height))
                        setLayout (new GridLayout (1, 2));
                    else
                        setLayout (new GridLayout (2, 1));
                    add(rect1);
                    add(rect2);
                }
                else {
                    setLayout (new BorderLayout());
                    add(cardChoice.getSelectedIndex() == 0 ?
                         rect1 : rect2);
                    cardChoice.setEnabled (true);
                }
            }
            else {
                setLayout (new BorderLayout());
                ScrollPane scroll = new ScrollPane();
                Panel panel = new Panel (new GridLayout (2, 1));
                panel.add(rect1);
                panel.add(rect2);
                scroll.add(panel);
                add(scroll);
            }

            super.doLayout();
        }

        public Dimension getPreferredSize() {
            return new Dimension (2 * width, height);
        }
    }


    public DynLayout() {

        Panel labelPanel = new Panel (new GridLayout (0, 1));
        Panel fieldPanel = new Panel (new GridLayout (0, 1));
        Panel buttonPanel = new Panel (new GridLayout (0, 1));
```

```
    labelPanel.add(new Label ("Width: "));
    labelPanel.add(new Label ("Height: "));

    fieldPanel.add(widthField);
    fieldPanel.add(heightField);

    cardChoice.add("outline");
    cardChoice.add("filled");
    buttonPanel.add(cardChoice);
    buttonPanel.add(applyButton);

    applyButton.addActionListener(this);
    cardChoice.addItemListener(this);

    Panel controlPanel = new Panel (new BorderLayout());
    controlPanel.add(labelPanel, BorderLayout.WEST);
    controlPanel.add(fieldPanel, BorderLayout.CENTER);
    controlPanel.add(buttonPanel, BorderLayout.EAST);

    frame.add(controlPanel, BorderLayout.NORTH);
    frame.add(dynPanel, BorderLayout.CENTER);

    frame.addWindowListener(new WindowAdapter() {
            public void windowClosing (WindowEvent ev) {
                notifyDestroyed();
            }
        } );

    frame.pack();
}


public void itemStateChanged (ItemEvent ev) {

    if (ev.getStateChange() == ItemEvent.SELECTED) {

        dynPanel.invalidate();
        frame.validate();
        dynPanel.repaint();
    }
}


public void actionPerformed (ActionEvent ev) {

    width = Integer.parseInt (widthField.getText());
    height = Integer.parseInt (heightField.getText());

    rect1.invalidate();
    rect2.invalidate();
    frame.validate();
}
public void startApp() {
    frame.show();
}

public void pauseApp() {
}

public void destroyApp (boolean unconditional) {
```

```
        frame.dispose();
    }
}
```

Custom Layout Managers

Although the PDAP AWT subset contains all AWT layout managers including the flexible GridBagLayout, there may still be cases that cannot be handled by the standard layout managers. Although nesting panels with different layout managers helps in many situations, this approach isn't suitable for all scenarios and is also very resource consuming. Thus, the design of custom layout managers, generating application dependent layouts, is of special importance for developing PDAP applications.

If we look back to the DialogMenuDemo application created in the section "Dialogs and Menus," the amounts and items are arranged using a GridLayout. Thus, the horizontal space is equally distributed to both columns regardless of their actual size. Using a GridBagLayout, it would be possible to distribute the space in a more flexible way. However, here you will learn the creation of custom layout managers by the example of a layout manager similar to GridLayout, but allowing different column widths and row heights. Because the layout is similar to the layout policy of HTML tables, this layout manager is called TableLayout.

The first step of creating a custom layout manager is to implement the LayoutManager interface, consisting of the methods listed in Table 4.3. Those methods are responsible for the interaction of the layout and the container it is assigned to.

Table 4.3. Methods of the LayoutManager Interface

Method	*Description*
void addLayoutComponent (String constraint, Component comp)	Adds the specified component with the specified constraint to the layout.
void layoutContainer (Container parent)	Lays out the contents of the given container.
Dimension minimumLayoutSize (Container parent)	Calculates the minimum size dimensions for the specified container.
Dimension preferredLayoutSize (Container parent)	Calculates the preferred size for the specified container.
void removeLayoutComponent (Component comp)	Removes the specified component from the layout.

In addition to the LayoutManager interface methods, the TableLayout needs a constructor. The TableLayout constructor takes the number of columns as parameter and stores it in the cols object variable. The number of rows isn't required because it can be calculated by dividing the total number of components by the number of columns. The following code snippet shows the TableLayout constructor and a helper method for calculating the row count:

```
public TableLayout (int cols) {
    this.cols = cols;

    if (cols < 1)
        throw new RuntimeException
            ("cols must be > 0");
}

int getRowCount (Container container) {
    return (container.getComponentCount() + cols - 1) / cols;
}
```

Before we can implement the methods for calculating the minimal or preferred layout size or performing the actual layout itself, it makes sense to think about a helper method caring about the common calculations of all those methods.

For both calculating the minimum layout sizes and performing the actual layout task, it is necessary to know the minimum sizes of the rows and columns. The height of a row is determined by the maximum cell height of that row. The width of a column is determined by the maximum cell width of that column. The helper method `getMinimumSizes()` fills the given `int` arrays with the corresponding calculations and returns the sums in a `Dimension` object:

```
public Dimension getMinimumSizes (Container container,
                                  int [] mcw, int [] mrh) {
    int count = container.getComponentCount();

    Dimension sum = new Dimension (0, 0);
    int i = 0;

    for (int y = 0; y < mrh.length; y++) {
        for (int x = 0; x < cols && i < count; x ++) {
            Dimension ms = container.getComponent
(i++).getMinimumSize();
            mcw [x] = Math.max (mcw [x], ms.width);
            mrh [y] = Math.max (mrh [y], ms.height);
        }
        sum.height += mrh [y];
    }


    for (int x = 0; x < cols; x++)
        sum.width += mcw [x];

    return sum;
}
```

Using this `getMinimumSizes()` method, the `minimumLayoutSize()` method of the `LayoutManager` interface can be implemented easily by just adding the container insets to the returned dimensions:

```
public Dimension minimumLayoutSize (Container container) {

    Insets insets = container.getInsets();
    int rows = getRowCount (container);

    Dimension result = getMinimumSizes
        (container, new int [cols], new int [rows]);

    result.width += cols - 1 + insets.left + insets.right;
    result.height += rows - 1 + insets.top + insets.bottom;

    return result;
}
```

For the limited screen sizes of PDAs, it seems appropriate to return the `minimumLayoutSize` also as a preferred layout size:

```
public Dimension preferredLayoutSize (Container container) {
    return minimumLayoutSize (container);
}
```

The main job of the `TableLayout` is done in the `layoutContainer()` method. The method resizes each component that is added to the container according to its height and width and the container size. If there is remaining space, all components are scaled with an equal factor, calculated by dividing the space available by the minimum layout size:

```
public void layoutContainer (Container container) {

    int count = container.getComponentCount();
    int rows = getRowCount (container);

    if (count == 0) return;

    Insets insets = container.getInsets();
    Dimension size = container.getSize();

    int x0 = insets.left;
    int y0 = insets.top;

    int w0 = size.width - x0 - insets.right;
    int h0 = size.height - y0 - insets.bottom;

    int [] mcw = new int [cols];
    int [] mrh = new int [rows];

    Dimension min = getMinimumSizes (container, mcw, mrh);

    // calculate a scale factor

    int scx = ((w0-cols+1) << 8) / min.width;
    int scy = ((h0-rows+1) << 8) / min.height;
    int i = 0;

    for (int y = 0; y < rows; y++) {
        int x1 = x0;
        int h = (mrh [y] * scy) >> 8;
        for (int x = 0; x < cols && i < count; x++) {
            int w = (mcw [x] * scx) >> 8;
            container.getComponent (i++).setBounds (x1, y0, w, h);
            x1 += w + 1;
        }
        y0 += h + 1;
    }
}
```

Because we don't need additional layout constraints such as `NORTH` or `CENTER` for the `BorderLayout`, we don't need to keep track of adding and removing components. Thus, the implementations of `addLayoutComponent()` and `removeLayoutComponent()` are left empty:

```
public void addLayoutComponent (String where, Component component) {
}

public void removeLayoutComponent (Component component) {
}
```

Note that if layout constraints are important, in most cases it is more appropriate to implement the improved `LayoutManager2` class, which can handle arbitrary objects as layout constraints instead of `String`s.

Listing 4.11 contains the full source code of the `TableLayout`. You can try the `TableLayout` by replacing the `GridLayout` in the `ShoppingChart` application. Figure 4.11 shows a corresponding screenshot. As you can see in the picture, in contrast to the original application, more space is distributed to the items and less to the amounts, resulting in a more adequate layout.

Figure 4.11. The `ShoppingChart2` application using the `TableLayout`.

Listing 4.11 `TableLayout.java`

```
public class TableLayout implements LayoutManager {

    int cols;

    public void addLayoutComponent (String where, Component component)
{
    }


    public void removeLayoutComponent (Component component) {
    }


    public TableLayout (int cols) {
        this.cols = cols;

        if (cols < 1)
            throw new RuntimeException
                ("cols must be > 0");
    }
    public Dimension getMinimumSizes (Container container,
                                      int [] mcw, int [] mrh) {

        int count = container.getComponentCount();

            Dimension sum = new Dimension (0, 0);
        int i = 0;

        for (int y = 0; y < mrh.length; y++) {
            for (int x = 0; x < cols && i < count; x ++) {
                Dimension ms = container.getComponent
(i++).getMinimumSize();
                mcw [x] = Math.max (mcw [x], ms.width);
```

```
                mrh [y] = Math.max (mrh [y], ms.height);
            }
            sum.height += mrh [y];
        }

        for (int x = 0; x < cols; x++)
            sum.width += mcw [x];

        return sum;
    }

    int getRowCount (Container container) {
        return (container.getComponentCount() + cols - 1) / cols;
    }

    public Dimension minimumLayoutSize (Container container) {

        Insets insets = container.getInsets();
        int rows = getRowCount (container);

        Dimension result = getMinimumSizes
            (container, new int [cols], new int [rows]);

        result.width += cols - 1 + insets.left + insets.right;
        result.height += rows - 1 + insets.top + insets.bottom;

        return result;
    }
    public Dimension preferredLayoutSize (Container container) {
        return minimumLayoutSize (container);
    }


    public void layoutContainer (Container container) {

        int count = container.getComponentCount();
        int rows = getRowCount (container);

        if (count == 0) return;

        Insets insets = container.getInsets();
        Dimension size = container.getSize();

        int x0 = insets.left;
        int y0 = insets.top;

        int w0 = size.width - x0 - insets.right;
        int h0 = size.height - y0 - insets.bottom;

        int [] mcw = new int [cols];
        int [] mrh = new int [rows];

        Dimension min = getMinimumSizes (container, mcw, mrh);

        // calculate a scale factor

        int scx = ((w0-cols+1) << 8) / min.width;
        int scy = ((h0-rows+1) << 8) / min.height;
        int i = 0;

        for (int y = 0; y < rows; y++) {
```

```
            int x1 = x0;
            int h = (mrh [y] * scy) >> 8;
            for (int x = 0; x < cols && i < count; x++) {
                int w = (mcw [x] * scx) >> 8;
                container.getComponent (i++).setBounds (x1, y0, w, h);
                x1 += w + 1;
            }
            y0 += h + 1;
        }
    }
}
```

Multiple Threads in the PDAP AWT Subset

As mentioned in the beginning of this chapter, one of the main restrictions of the PDAP AWT subset is that AWT methods might be called from the event dispatching thread only. This restriction is necessary because the implementation of a thread-safe AWT subset is more complex and slower than an AWT subset based on the single thread model. Actually, the modern J2ME SWING user interface doesn't allow calls from multiple threads for the same reasons. However, this does not mean that the PDAP AWT cannot work with threads at all. It is just necessary to take some extra steps when making calls to AWT methods from separate threads.

For calls from separate threads, the AWT class `EventQueue` provides two static methods, `invokeLater()` and `invokeAndWait()`. Both methods take an object implementing the `Runnable` interface as parameter. Calls to these methods can be performed from threads other than the event handling thread. AWT automatically ensures that the `run()` method of the given `Runnable` class is then called from the AWT event thread. The difference between `invokeLater()` and `invokeAndWait()` is that `invokeLater()` returns immediately, whereas `invokeAndWait()` does not return until the `run()` method of the given object has been executed.

An example using multiple threads is a simulation running in a separate thread, which needs to update some components showing the simulation state from time to time.

The `PdaLander` application is a simplified simulation of a lunar landing. It displays the landing parameters such as altitude, velocity, and fuel remaining and allows the user to set the thrust of the engine to a value between 0 and 100%.

The core of the application, the simulation thread can be implemented as an inner class of the `PdaLander` application, accessing the simulation state variables `velocity`, `height`, `fuel`, and `thrust`. The constants `GRAVITY` and `ACCELERATION` reflect the gravity of the moon and the acceleration available at 100% thrust. Note that all values are measured in fine-grained units in order to avoid slow floating-point operations. All values are measured in metric units in order to simplify the calculations and to save our lunar lander from the fate of the Mars Polar lander.

The simulation is performed by measuring the elapsed time since the last simulation step and then updating all variables accordingly. The current velocity is recalculated based on the acceleration and time, the height is calculated based on the velocity and time, and finally the remaining fuel is adjusted. Then the `invokeAndWait`() method is called in order to update the user interface, ensuring synchronization with the AWT event thread:.

```
import java.awt.*;
import java.awt.event.*;
import javax.microedition.midlet.*;

public class PdaLander extends MIDlet {

    static final long GRAVITY = 1620; // mm/s2
    static final long ACCELERATION = 2*GRAVITY;
```

```
    long velocity = 0;       // 0 mm/s (= 0 m/s)
    long height = 1000000;  // 1000000 mm = 1000 m = 1 km
    long fuel = 100000;      // ms = 120 s = 2 min
    long thrust = 0;

    class Simulation extends Thread {

        long time = System.currentTimeMillis();

        public void run() {
            do {
                long dt = System.currentTimeMillis() - time;

                velocity += ((GRAVITY - (ACCELERATION * thrust)
                              / 100) * dt) / 1000;
                height -= (velocity * dt) / 1000;
                time  += dt;
                fuel -= (thrust * dt) / 100;

                try {
                    EventQueue.invokeAndWait (screenManager);
                }
                catch (Exception e) {
                    throw new RuntimeException (e.toString());
                }
            }
            while (height > 0);
        }
    }
```

The `screenManager`, which contains the run method indirectly called from the `Simulation` object, is responsible for updating the user interface according to the simulation state. It also reads the new thrust setting and determines if the lander has landed safely or crashed into the ground. The helper method `milliToStr()` just converts the fine grained units to the usual units by dividing them by 1000. The maximum allowable landing speed is one meter per second:

```
static final long MAX_VELOCITY = 1099;

Label velocityDisplay = new Label();
Label heightDisplay = new Label();
Label fuelDisplay = new Label();
Scrollbar thrustSlider =
    new Scrollbar (Scrollbar.HORIZONTAL, 0, 0, 0, 100);

ScreenManager screenManager = new ScreenManager();

class ScreenManager implements Runnable {

    public String milliToStr (long milli) {
        return (milli / 1000) + "." + Math.abs ((milli % 1000) / 100);
    }

    public void run() {

        if (height <= 0)
            heightDisplay.setText
                (velocity <= MAX_VELOCITY ? "Landed" : "Crashed!");
        else
            heightDisplay.setText (milliToStr (height));
```

```
        velocityDisplay.setText (milliToStr (velocity));
        fuelDisplay.setText (milliToStr (fuel < 0 ? 0 : fuel));

        thrust = fuel <= 0 ? 0 : thrustSlider.getValue();
    }
}
```

The constructor just sets up the user interface by adding the controls and labels to the application frame:

```
    public PdaLander() {

        Panel intermediate = new Panel (new BorderLayout());
        Panel controlPanel = new Panel (new GridLayout (0, 2));

        controlPanel.add(new Label ("Height:"), Label.LEFT);
        controlPanel.add(heightDisplay);
        controlPanel.add(new Label ("Velocity:"));
        controlPanel.add(velocityDisplay);
        controlPanel.add(new Label ("Fuel:"));
        controlPanel.add(fuelDisplay);
        controlPanel.add(new Label ("Thrust:"));
        controlPanel.add(thrustSlider);

        intermediate.add(controlPanel, BorderLayout.NORTH);
        frame.add(intermediate, BorderLayout.CENTER);
        frame.addWindowListener(new WindowAdapter() {
                public void windowClosing (WindowEvent ev) {
                    notifyDestroyed();
                }
            } );

        frame.pack();
    }
```

When the application is brought on the screen and not yet running, the simulation thread is started:

```
public void startApp() {
    frame.show();
    if (simulation == null) {
        simulation = new Simulation();
        simulation.start();
    }
}
```

When the application is paused by the Application Management System (AMS), we do nothing. This behavior could be improved by suspending the ScreenManager thread.

```
public void pauseApp() {
}
```

When the application is terminated, the height is set to a negative value in order to make sure that the simulation thread terminates immediately. Then the frame is disposed:

```
    public void destroyApp (boolean unconditional) {
        height = -100000000;
        frame.dispose();
    }
}
```

A simple approach to master the landing is to let the lander fall to 550 meters and then go to full thrust. When the speed is reduced to 1 m/s, go to 50% thrust.

A full listing of an improved lunar lander example without inserted text is contained in the next section.

Combined Application Example: A Lunar Lander with Graphical Display

As an example of combining some of the techniques demonstrated in this chapter, we will enhance the lunar lander example from the previous section by adding a graphical display and slightly modifying the screen layout depending on the ratio between height and width.

For the external camera view component of the lander, we create a new inner class `ExternalView` derived from `Canvas`. In the paint method, we draw a triangle representing the lander and three lines representing the engine exhaust, depending on the thrust level. The screen position of the lander is calculated in the `getScrY`() method by multiplying the real height with the screen height and then dividing by the maximum height. The old thrust level and display position are saved in order to be able to determine if the values have changed in the check method. The check method is called from the `ScreenManager`. It forces a repaint only if the thrust level or screen height has changed in order to avoid unnecessary flickering.

The animation could be improved further by repainting only the area of the `ExternalView` component that was actually affected by the move of the lander. Even smoother animation would be possible by using an offscreen buffer as in the MIDP stopwatch example. However, for games MIDP is probably the more appropriate profile anyway, so we do not repeat the corresponding code here:

```
class ExternalView extends Canvas {

    int oldY;
    long oldThrust;

    public void paint (Graphics g) {

        int x = getSize().width / 2;
        int y = getScrY();

        g.drawLine (x-5, y-1, x+5, y-1);
        g.drawLine (x-5, y-1, x, y - 10);
        g.drawLine (x+5, y-1, x, y - 10);

        if (thrust > 10) {
            g.drawLine (x-3, y+1, x-3, (int) (y + 1 + thrust /
20));
            g.drawLine (x+3, y+1, x+3, (int) (y + 1 + thrust /
20));
            g.drawLine (x, y+1, x,  (int) (y + 1 + thrust / 15));
        }

        oldY = y;
        oldThrust = thrust;
    }

    public Dimension getPreferredSize() {
        return new Dimension (50, 100);
    }

    public Dimension getMinimumSize() {
        return new Dimension (20, 50);
    }
```

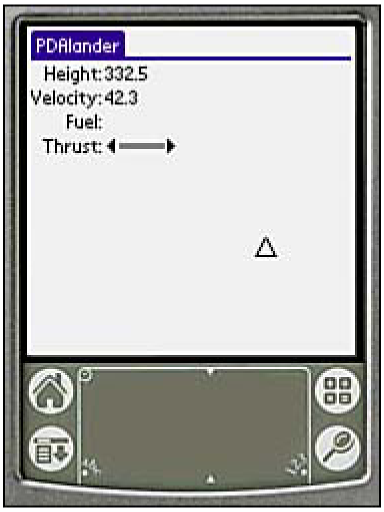
```
        int getScrY() {
            int scrH = getSize().height;
            return scrH - (int) (height * scrH / START_HEIGHT);
        }

        public void check() {
            if (thrust != oldThrust || oldY != getScrY())
                repaint();
        }

    }
```

The second improvement of our new `PdaLander` version is an automatic adoption to the screen size ratio. Because the graphical display takes some additional space to the right of the controls, the labels are automatically displayed above the controls instead of to the left if the height of the screen is greater than the width:

```
    public PdaLander2() {

        Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
        boolean vertical = d.height > d.width;

        Panel intermediate = new Panel (new BorderLayout());
        Panel controlPanel = new Panel (new GridLayout (0, vertical ?
1 : 2));

        int align = vertical ? Label.LEFT : Label.RIGHT;

        controlPanel.add(new Label ("Height:", align));
        controlPanel.add(heightDisplay);
        controlPanel.add(new Label ("Velocity:", align));
        controlPanel.add(velocityDisplay);
        controlPanel.add(new Label ("Fuel:", align));
        controlPanel.add(fuelDisplay);
        controlPanel.add(new Label ("Thrust:", align));
        controlPanel.add(thrustSlider);

        intermediate.add(controlPanel, BorderLayout.NORTH);
        frame.add(intermediate, BorderLayout.WEST);
        frame.add(externalView, BorderLayout.CENTER);

        frame.addWindowListener(new WindowAdapter() {
                public void windowClosing (WindowEvent ev) {
                    notifyDestroyed();
                }
            } );

        frame.pack();
    }
```

Listing 4.12 shows the complete source code of the enhanced `PdaLander` example and Figure 4.12 shows a screenshot of the `PdaLander` running on a Palm Pilot.

Figure 4.12. The `PdaLander2` application running on a Palm Pilot.



Similar to the other examples, there is much room for your own improvements. For example, the display of a crashed lander could be designed differently from a successful landing. A more sophisticated extension would be to allow vertical movements and to add some kind of terrain structure. Another possible improvement would be to add a double buffered display, eliminating any screen flickering.

Listing 4.12 PdaLander2.java

```
import java.awt.*;
import java.awt.event.*;
import javax.microedition.midlet.*;

public class PdaLander2 extends MIDlet {

    Frame frame = new Frame ("PDAlander");
    Simulation simulation;
    ScreenManager screenManager = new ScreenManager();

    Label velocityDisplay = new Label();
    Label heightDisplay = new Label();
    Label fuelDisplay = new Label();
    Scrollbar thrustSlider =
        new Scrollbar (Scrollbar.HORIZONTAL, 0, 0, 0, 100);

    ExternalView externalView = new ExternalView();

    static final long START_HEIGHT = 1000000;
    static final long GRAVITY = 1620; // mm/s2
    static final long ACCELERATION = 2*GRAVITY;
    static final long MAX_VELOCITY = 1000;

    long velocity = 0;          // mm/s
    long height = START_HEIGHT; // mm
    long fuel = 100000;         // ms
    long thrust = 0;


    class ExternalView extends Canvas {

        int oldY;
        long oldThrust;

        public void paint (Graphics g) {
```

```
            //    g.drawRect (0, 0, d.width, d.height);

            int x = getSize().width / 2;
            int y = getScrY();

            g.drawLine (x-5, y-1, x+5, y-1);
            g.drawLine (x-5, y-1, x, y - 10);
            g.drawLine (x+5, y-1, x, y - 10);

            if (thrust > 10) {
                g.drawLine (x-3, y+1, x-3, (int) (y + 1 + thrust /
20));
                g.drawLine (x+3, y+1, x+3, (int) (y + 1 + thrust /
20));
                g.drawLine (x, y+1, x,  (int) (y + 1 + thrust / 15));
            }

            oldY = y;
            oldThrust = thrust;
        }

        public Dimension getPreferredSize() {
            return new Dimension (50, 100);
        }

        public Dimension getMinimumSize() {
            return new Dimension (20, 50);
        }


        int getScrY() {
            int scrH = getSize().height;
            return scrH - (int) (height * scrH / START_HEIGHT);
        }

        public void check() {
            if (thrust != oldThrust || oldY != getScrY())
                repaint();
        }

    }


    class ScreenManager implements Runnable {

        public String milliToStr (long milli) {
            return (milli / 1000) + "." + Math.abs ((milli % 1000) /
100);
        }

        public void run() {

            if (height <= 0)
                heightDisplay.setText
                    (velocity <= MAX_VELOCITY
                     ? "Landed"
                     : "Crashed!");
            else
                heightDisplay.setText (milliToStr (height));
```

```
            velocityDisplay.setText (milliToStr (velocity));
            fuelDisplay.setText (milliToStr (fuel < 0 ? 0 : fuel));

            thrust = fuel <= 0 ? 0 : thrustSlider.getValue();

            externalView.check();
        }
    }


    class Simulation extends Thread {

        long time = System.currentTimeMillis();

        public void run() {  // 0..100

            do {
                long dt = System.currentTimeMillis() - time;
                velocity += ((GRAVITY - (ACCELERATION * thrust)
                                 / 100) * dt) / 1000;
                height -= (velocity * dt) / 1000;
                time  += dt;
                fuel -= (thrust * dt) / 100;

                try {
                    EventQueue.invokeAndWait (screenManager);
                }
                catch (Exception e) {
                    throw new RuntimeException (e.toString());
                }
            }
            while (height > 0);
        }
    }


    public PdaLander2() {

        Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
        boolean vertical = d.height > d.width;

        Panel intermediate = new Panel (new BorderLayout());
        Panel controlPanel = new Panel (new GridLayout (0, vertical ?
1 : 2));

        int align = vertical ? Label.LEFT : Label.RIGHT;

        controlPanel.add(new Label ("Height:", align));
        controlPanel.add(heightDisplay);
        controlPanel.add(new Label ("Velocity:", align));
        controlPanel.add(velocityDisplay);
        controlPanel.add(new Label ("Fuel:", align));
        controlPanel.add(fuelDisplay);
        controlPanel.add(new Label ("Thrust:", align));
        controlPanel.add(thrustSlider);

        intermediate.add(controlPanel, BorderLayout.NORTH);
        frame.add(intermediate, BorderLayout.WEST);
        frame.add(externalView, BorderLayout.CENTER);

        frame.addWindowListener(new WindowAdapter() {
```

```
                public void windowClosing (WindowEvent ev) {
                    notifyDestroyed();
                }
            } );
        frame.pack();
    }


    public void startApp() {
        frame.show();
        if (simulation == null) {
            simulation = new Simulation();
            simulation.start();
        }
    }

    public void pauseApp() {
    }

    public void destroyApp (boolean unconditional) {
        height = -100000000;
        frame.dispose();
    }
}
```

Summary

After the short introduction in which we introduced the use of AWT components, you should now be familiar with handling those that are contained in the AWT. Moreover, you have learned to use layout managers to place AWT components in a container such as panels and frames and even know how to handle events that might be invoked by user interactions.

Chapter 5. Data Persistency

IN THIS CHAPTER

Mobile devices such as cellular phones or PDAs normally don't have a file system available like that found on desktop computers. However, these devices still provide a mechanism to store data persistently. This mechanism is based on memory techniques such as flash memory. These techniques provide an advantage in that the head-positioning times associated with disk-based media are avoided. Thus, a more appropriate system based on random record access instead of sequential file reading is provided for these devices. For access to this record system, PDAP and MIDP contain the Record Management System (RMS).

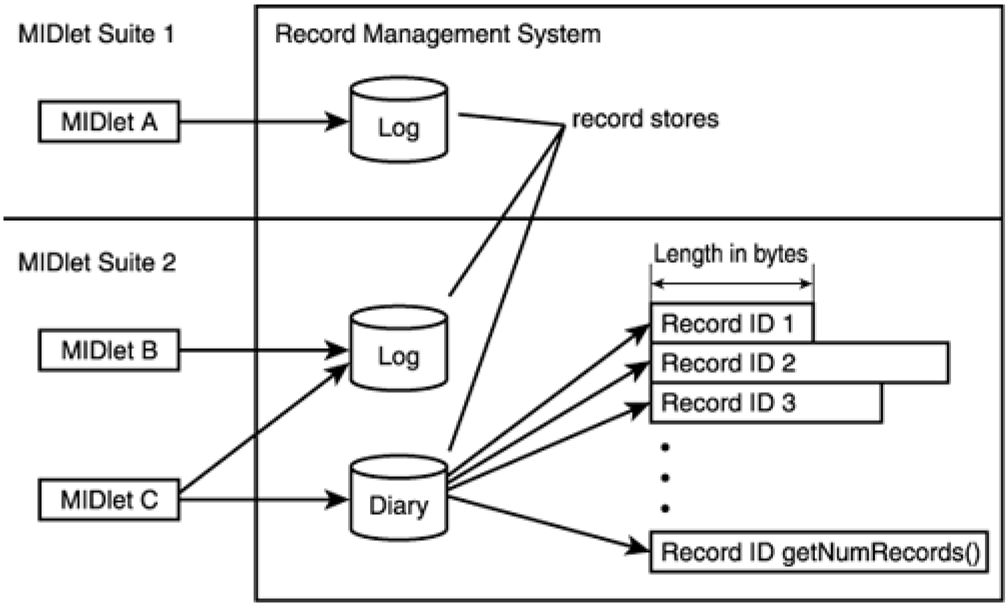
RMS Basics

The RMS API is contained in the package `javax.microedition.rms`. A MIDlet can create and access a number of *record stores*. Each record store has a name that is unique in the MIDlet suite. It consists of a number of variable-length records. The records themselves are simple byte arrays without any further predefined structure.

The RMS API does not include routines for integrity maintenance. All maintenance is performed by the platform automatically.

MIDlets can access record stores of other MIDlets only if they are in the same MIDlet suite. For instance, it is possible to include a MIDlet A inside MIDlet suite 1 and include a MIDlet B in MIDlet suite 2. Although both MIDlet suites are on the same device, MIDlet A is not allowed to manipulate the log or diary information of MIDlet B because it's in a different suite. Figure 5.1 shows how the RMS maintains multiple records stores of different MIDlets.

Figure 5.1. The record store's visibility and structure.



Within a MIDlet suite, MIDlets can create record stores with case-sensitive names up to 32 Unicode characters long. Inside a MIDlet suite, the names must be unique. However, MIDlets are allowed to create record stores with the same name in different MIDlet suites. In that case, the platform is responsible for handling them as separate record stores. Additionally, the platform is responsible for deleting the record stores that are created by a MIDlet on the device when the MIDlet is deleted.

All operations on the record store are *atomic*. That means that if two threads write the same record in parallel, these calls are serialized by the system automatically. This process avoids corruption of the internal structure of the record store, but does not prevent the application data structure from becoming invalid: Serializing two write operations to the same record means that the second write operation overwrites the result of the previous operation, possibly causing application problems.

The RMS API does not provide any predefined mechanisms for locking transactions.

The record store maintains a date/time stamp indicating the last modification of the record store and a version number. The date/time stamp consists of a long integer representing the time in the format of the `System.currentTimeMillis()` method. Each time the record store is modified, the date/time stamp is updated. The version number is represented by an integer value. It is incremented for each modifying operation on the record store.

The primary key to records in a record store is an integer value called the record ID. The ID of the first record created is 1. The IDs of subsequent records are incremented by 1.

Basic Functionality of the Class RecordStore

Record stores are represented in the RMS API by the class `RecordStore`. The `RecordStore` class provides methods to open, close, read, and manipulate the record store. It also provides access to meta-information such as the number of records and the memory space still available and currently consumed.

Global Record Store Methods and Exceptions

An instance of the `RecordStore` class, representing a single record store, can be obtained via the static method `openRecordStore()`. The method call requires two parameters: the name of the `RecordStore` and a flag specifying whether the `RecordStore` should be created if it does not exist yet. The name of the record store should be a `String` consisting of no more than 32 Unicode characters. The names are case sensitive, so *Foo*, *foo*, and *FOO* would denote different record stores. The following line opens the record store named `myRecords` and creates it if it does not exist yet:

```
RecordStore myRecords = RecordStore.openRecordStore("myRecords",
true);
```

Other general static methods are provided for listing all record stores of a MIDlet suite and deleting a record store. Because these methods are static, it is possible to use them without obtaining an instance of `RecordStore` first. These general methods are listed in Table 5.1. To use all other `RecordStore` access methods, you must create a `RecordStore` instance by calling `openRecordStore()`.

Note

MIDP 2.0 introduces two additional `openRecordStore()` methods to share

RecordStores across multiple MIDlet suites. An additional `setMode()` method allows you to change the access mode of a RecordStore later.

The first new `openRecordStore()` method requires four parameters. The first two parameters are identical to the existing `openRecordStore()` method and define the name of the record store and whether the store should be created as if it does not yet exist. The third parameter defines the access mode. The constant `RecordStore.AUTHMODE_PRIVATE` indicates that only MIDlets in the same MIDlet suite can access the RecordStore. `RecordStore.AUTHMODE_ANY` enables any MIDlet to access the RecordStore. The fourth parameter is a boolean specifying whether the RecordStore is writable by other MIDlet suites in the `AUTHMODE_ANY` case.

The second new `openRecordStore()` method takes only three String parameters. Again, the first parameter defines the name of the RecordStore. The second and third parameter define a vendor and a MIDlet suite name required to get access to the RecordStore.

The authentication mode of a record store can be changed after creation using the `setMode()` method. This method takes the previously described authentication mode and the write flag as parameters.

All three methods can throw a SecurityException if the desired operation is not permitted.

For closing a record store, the API provides the method `closeRecordStore()`. At the latest, any open record store owned by an application should be closed when the application is terminated. Please note that the number of calls to close a certain record store must match the calls to `openRecordStore()`. Calling `closeRecordStore()` allows the system to free resources associated with a record store.

Table 5.1. Static RecordStore Methods

Method Name	***Purpose***
`String[] listRecordStores()`	Returns all record stores in a `String` array that are available in the current MIDlet suite.
`RecordStore openRecordStore (String recordStoreName, boolean`	Opens a record store.

`createIfNecessary)`	
`void deleteRecordStore (String recordStoreName)`	Deletes a complete record store from the MIDlet suite.

The `RecordStore` methods can throw different types of `Exception`s in case of a failure. The exceptions are listed in Table 5.2. For example, a `RecordStoreNotFoundException` is thrown if the desired record store does not exist in the MIDlet suite and the `openRecordStore()` method is called without setting the `create` option. In case of success, the method returns an instance of the `RecordStore` class.

Table 5.2. Possible `RecordStore` Exceptions

Exception Name	***Reason***
`InvalidRecordIDException`	Thrown if the index that is passed to an operation like `deleteRecord()`, `getRecord()`, `getRecordSize()`, or `setRecord()` is not a valid index.
`RecordStoreException`	Thrown when a general record store failure occurs.
`RecordStoreFullException`	Thrown to indicate that the operation cannot be completed because the complete storage space available for the record store is consumed already.
`RecordStoreNotFoundException`	Thrown by the methods `openRecordStore()` and `deleteRecordStore()` to indicate that the record store with the specified name does not exist.
`RecordStoreNotOpenException`	Thrown when a method to access a record is called after the record store was closed.

Manipulating Single Records

When the record store is successfully opened, you can add new records using the `addRecord()` method. This method takes three parameters. The first is the byte array containing the data that should be written to the record store. The second specifies the offset from which to start writing data of the byte array. The third specifies how many bytes of the given array should be written. The method returns the index of the newly created record. Like all `RecordStore` methods, `addRecord()` may throw different types of `RecordStoreException`s in case of a failure. If you want to write the complete byte array, set the second parameter to 0 and the third to the length of the byte array:

```
myRecords.addRecord (myBytes, 0, myBytes.length);
```

For replacing a record at a given index, `RecordStore` provides the method `setRecord()`. The usage of the `setRecord()` method is similar to `addRecord()`, except that the index of the record to be replaced must be given as the first parameter. In contrast to `addRecord()`, no index value is returned. Please note that the first record in a record store has the index number 1, and not 0 as in most other Java APIs dealing with indexed structures. Specifying a record ID of 0 or any other invalid record ID will result in an `InvalidRecordIDException`. The following call replaces the first record of a record store with the complete byte array given as the second parameter:

```
myRecords.setRecord (1, myBytes, 0, myBytes.length);
```

For reading records, RMS provides two different `getRecord()` methods. The simple version of `getRecord()` takes a record ID as a parameter and returns a byte array containing the corresponding record data. Using the following line of code, you can read the data of the first record in the `myRecords` record store:

```
byte[] recordData = myRecords.getRecord (1);
```

The second variant of getRecord() avoids allocating a new byte array by writing data to a byte array given by the application. It is invoked with more parameters and provides a different return value. As with the simple version, the first parameter is the record ID. The second parameter is a byte array that is used for storing the data read. The third parameter is the offset at which to start writing the record data in the given byte array. As a result, the method returns the number of bytes transferred to the specified buffer. The application is required to provide a buffer that is large enough to store the record with the given ID. If the buffer size is not sufficient, an ArrayIndexOutOfBoundException is thrown. The following lines allocate a byte array as a buffer and read the first record from the myRecords record store into the newly allocated buffer:

```
byte[] myBuffer = new byte [BIG_ENOUGH];
int numberOfBytesRead = myRecords.getRecord (1, myBuffer, 0);
```

Records that are no longer needed can be deleted using the deleteRecord() method. The only parameter of deleteRecord() is the ID of the record to be deleted. After deletion, the record is removed from the RecordStore irrevocably. Please note that the record ID of the removed record is not reused and cannot be reset with a new record using the setRecord() method. In case of resetting a previously deleted record, an InvalidRecordStoreIDException is thrown. All further records keep their record IDs.

Meta Information

In addition to methods manipulating whole record stores or single records, the RMS API also provides a set of methods for obtaining information about a record store or single records.

All methods to get record store meta information are listed in Table 5.3. One important method needed, for example, to iterate all records in a record store is getNumRecords(). This method is invoked on a record store instance and returns the number of records stored in the record store as an integer value. The following line shows how the method is called to get the number of records in the myRecords record store:

```
int numberOfRecords = myRecords.getNumRecords();
```

Table 5.3. Methods for Accessing RecordStore Meta Information

Method Name	***Purpose***
long getLastModified()	Returns the last modification of the record store in the format returned by System.currentTimeMillis().
String getName()	Returns the name of the current record store.
int getNextRecordID()	Returns the record ID of the next record that will be added to the record store.
int getRecordSize(int recordID)	Returns the byte size of the record at the given ID.
int getSize()	Returns the byte size of the complete record store.
int getSizeAvailable()	Returns the storage space that is currently available for storing records.
int getVersion()	Returns the integer value that represents the number of modifications of the record store.

A Simple Diary Application Using RMS

Using the methods described in the previous section, you can now build a simple diary application. The application will demonstrate how to store simple objects (Strings) in a record store.

The purpose of the application is to store one `String` per day. The user interface consists of a widget and buttons for browsing the diary creating new diary entries.

Because RMS is available in MIDP and PDAP, we will start with the RMS-related functions for loading and saving `String`s that can be used in both versions of the diary application. The complete sources of both diary versions, differing in the user interface parts, are given in Listing 5.1 for MIDP and in Listing 5.2 for PDAP. Here, we will focus on the RMS specific calls like opening the record store, loading and saving `String`s, and closing the record store.

Because the RMS only provides a mechanism for handling byte array records, you need to convert the diary entries represented as `String`s before adding them to the record store. Also, before you can display a diary record in a text widget, you need to convert the byte array back into a `String`. Fortunately, the functionality for these conversions is available in the `String` class. In order to convert a `String` instance to a byte array, you can just call the `getBytes()` method of the `String` class. The following line shows how to convert a `String` to a byte array using the `getBytes()` method:

```
byte[] byteArray = new String ("Hello World!").getBytes();
```

The resulting byte array contains the `String` *Hello World!*. This byte array can be stored in a record store using the `addRecord()` or `setRecord()` method. In order to convert a byte array read from a record store back into a `String` object, you can use the corresponding `String` constructor taking a byte array as parameter.

Before you begin implementing the diary functionality, you need two variables containing the diary record store and an index that points to the record currently displayed. In both versions of the application, these variables are declared as follows:

```
RecordStore diary;
int currentId = 1;
```

Now you can start with the `loadEntry()` method that loads a diary entry from the record store and converts it into a `String`. The `loadEntry()` method gets the ID of the desired record as input, sets the current ID to the given value, and returns the corresponding `String`. If the ID is not valid, a new record is created automatically, and the ID returned from `addRecord()` is set as the current ID. The following code snippet shows the implementation of `loadEntry()` for both the MIDP and PDAP versions:

```
public String loadEntry (int newId) throws RecordStoreException {
    if (newId < 1 || newId > diary.getNumRecords()) {
        byte [] data = "".getBytes();
        currentId = diary.addRecord (data, 0, data.length);
    }
    else
        currentId = newId;
    return new String (diary.getRecord (currentId));
}
```

Now you can load records and also append new records, but you still need a method to update diary entries when the user enters additional information or changes the entry. The `saveEntry()` method stores the given string with the ID stored in the `currentId` variable:

```
public void saveEntry (String entry) throws RecordStoreException {
    byte [] data = (entry).getBytes();
    diary.setRecord (currentId, data, 0, data.length);
}
```

Using the two methods `loadEntry()` and `saveEntry()`, you have implemented two basic methods that are responsible for loading and storing records to a record store in both the MIDP and PDAP versions of the diary application. The next step to implement the diary application is to open the diary record store in the constructor of the application.

As described in the previous section, the method `openRecordStore()` can throw `RecordStoreException`s when it is not able to open the record store for some reason. In order to handle these exceptions, it is necessary to put the call in a corresponding try-catch block. In the following snippet, you throw a `RuntimeException`, terminating the whole application in case of a failure. In real applications, more elaborate error handling may be necessary:

```
try {
   diary = RecordStore.openRecordStore ("diary", true);
   // load and display the last entry here.
}
catch (RecordStoreException e) {
    throw new RuntimeException ("Cannot open diary; reason: "+e);
}
```

After the record store is opened, you jump to the last record of the `Diary` record store and pass the stored `String` to the text widget of the `Diary` implementation. Because the UI widgets in MIDP and PDAP are different, we included a comment as placeholder for the actual code. The complete code of the constructors is contained in both full listings.

Note

If you compare the listings to the code snippets for `loadEntry()` and `saveEntry()`, you will notice that you save an additional empty space to each record, and use `String.trim()` when loading it from the `RecordStore`. You need this workaround because the J2ME WTK 1.0 throws an exception if empty records are stored in a record store.

In order to make sure that the current record is written back and to avoid orphan resources, you need to save the current entry and close the record store when the application is terminated. You do so by implementing a corresponding `destroyApp()` method. Again, we've included a comment instead of profile-dependent code where the current entry is obtained from the user interface widget:

```
public void destroyApp (boolean unconditional) {
    try {
        String text;
        // fill text with content of the UI widget here
        saveEntry (text);
        diary.closeRecordStore();
    }
    catch (RecordStoreException e) {
        throw new RuntimeException ("Cannot close Diary; reason: "+e);
    }
}
```

When the code is completed with a corresponding user interface, you have a full diary application as shown in Listings 5.1 and 5.2. Figures 5.2 and 5.3 show the running application.

Figure 5.2. The running `RmsDemoMidp` application.

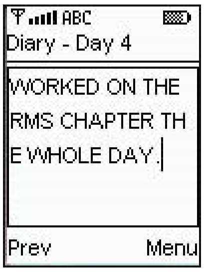


Figure 5.3. The running `RmsDemoPdap` application.

Listing 5.1 `RmsDemoMidp.java`—The Diary Application for MIDP

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

public class RmsDemoMidp extends MIDlet implements CommandListener {

    TextBox textBox = new TextBox ("Diary", "", 150, TextField.ANY);
    int currentId = 1;
    RecordStore diary;

    Display display;
    static final Command prevCommand = new Command ("Prev",
Command.SCREEN, 1);
    static final Command nextCommand = new Command ("Next",
Command.SCREEN, 2);
    static final Command newCommand = new Command ("New",
Command.SCREEN, 3);

    public RmsDemoMidp() {
        try {
            diary = RecordStore.openRecordStore ("diary", true);
            String text = loadEntry (diary.getNumRecords());
            textBox.setString (text);
            textBox.setTitle ("Diary - Day " + currentId);
        }
        catch (RecordStoreException e) {
            throw new RuntimeException ("Cannot open diary; reason:
"+e);
        }
```

```
        textBox.addCommand (prevCommand);
        textBox.addCommand (nextCommand);
        textBox.addCommand (newCommand);

        textBox.setCommandListener(this);
    }

    public String loadEntry (int newId) throws RecordStoreException {
        if (newId < 1 || newId > diary.getNumRecords()) {
            byte [] data = " ".getBytes();
            currentId = diary.addRecord (data, 0, data.length);
        }
        else
            currentId = newId;
        return new String (diary.getRecord (currentId)).trim();
    }


    public void saveEntry (String entry) throws RecordStoreException
{
        byte [] data = (entry + " ").getBytes();
        diary.setRecord (currentId, data, 0, data.length);
    }

    public void startApp() {
        display = Display.getDisplay (this);
        display.setCurrent (textBox);
    }

    public void destroyApp (boolean unconditional) {
        try {
            String text;
            text = textBox.getString();
            saveEntry (text);
            diary.closeRecordStore();
        }
        catch (RecordStoreException e) {
            throw new RuntimeException ("Cannot close Diary; reason:
"+e);
        }
    }

    public void pauseApp() {
    }
    public void commandAction (Command c, Displayable d) {
        try {
            saveEntry (textBox.getString());

            if (c == nextCommand && currentId < diary.getNumRecords())
{
                textBox.setString (loadEntry (currentId+1));
            }
            else if (c == prevCommand && currentId > 1) {
                textBox.setString (loadEntry (currentId-1));
            }

            else if (c == newCommand) {
                textBox.setString (loadEntry (diary.getNumRecords() +
1));
            }
```

```
            textBox.setTitle ("Diary - Day " + currentId);

        }
        catch (RecordStoreException e) {
            throw new RuntimeException ("Cannot perform; reason: "+e);
        }
    }
}
```

Listing 5.2 `RmsDemoPdap.java`—The Diary Application for PDAP

```
import java.awt.*;
import java.awt.event.*;
import javax.microedition.rms.*;
import javax.microedition.midlet.*;

public class RmsDemoPdap extends MIDlet implements ActionListener {
    Frame frame = new Frame();
    TextArea textArea = new TextArea();
    int currentId;
    RecordStore diary;

    Button buttonPrev = new Button ("Prev");
    Button buttonNext = new Button ("Next");
    Button buttonNew = new Button ("New");

    public RmsDemoPdap() {
        try {
            diary = RecordStore.openRecordStore ("diary", true);
            String text = loadEntry (diary.getNumRecords());
            textArea.setText (text);
            frame.setTitle ("Diary - Day " + currentId);
        }
        catch (RecordStoreException e) {
            throw new RuntimeException ("Cannot open diary; reason:
"+e);
        }


        frame.add("Center", textArea);

        Panel buttons = new Panel();

        buttons.add(buttonPrev);
        buttons.add(buttonNext);
        buttons.add(buttonNew);

        buttonPrev.addActionListener(this);
        buttonNext.addActionListener(this);
        buttonNew.addActionListener(this);

        frame.add("South", buttons);
        frame.pack();

        frame.addWindowListener(new WindowAdapter() {
                public void windowClosing (WindowEvent ev) {
                    destroyApp (true);
                    notifyDestroyed();
                }
            });
    }
```

```
    public String loadEntry (int newId) throws RecordStoreException {
        if (newId < 1 || newId > diary.getNumRecords()) {
            byte [] data = " ".getBytes();
            currentId = diary.addRecord (data, 0, data.length);
        }
        else
            currentId = newId;

        return new String (diary.getRecord (currentId));
    }


    public void saveEntry (String entry) throws RecordStoreException
{
        byte [] data = entry.getBytes();
        diary.setRecord (currentId, data, 0, data.length);
    }


    public void startApp() {
        frame.show();
    }


    public void destroyApp (boolean unconditional) {
        try {
            saveEntry (textArea.getText());
            diary.closeRecordStore();
        }
        catch (RecordStoreException e) {
            throw new RuntimeException ("Cannot close Diary; reason:
"+e);
        }
    }

    public void pauseApp(){
    }


    public void actionPerformed (ActionEvent ev) {

        try {
            saveEntry (textArea.getText());

            if (ev.getSource() == buttonNext
                && currentId < diary.getNumRecords())
                textArea.setText (loadEntry (currentId+1));
            else if (ev.getSource() == buttonPrev && currentId > 1)
                textArea.setText (loadEntry (currentId-1));
            else if (ev.getSource() == buttonNew)
                textArea.setText (loadEntry (diary.getNumRecords() +
1));
            frame.setTitle ("Diary - Day " + currentId);
        }
        catch (RecordStoreException e) {
            throw new RuntimeException ("Cannot perform; reason: "+e);
        }
    }
}
```

Record Listeners

In contrast to the desktop file system, it is possible to register listeners to record stores. A listener is informed whenever a record is changed in the record store it is registered to. This mechanism may be useful if different threads or different MIDlets of the same MIDlet suite are operating on one record store in parallel. A possible application could be some kind of automated logging service, writing records in the background. A MIDlet providing a GUI for the service could register itself as listener to the log record store when it is running in the foreground. Thus, it would be able to update its display of the log entries whenever a new record is added.

The corresponding `RecordListener` interface consists of the following three methods:

```
void recordAdded (RecordStore recordStore, int id)
```

is called when a new record with the given ID is added to the given record store.

```
void recordChanged (RecordStore recordStore, int id)
```

is called when the given record is changed.

```
void recordDeleted (RecordStore recordStore, int id)
```

is called when the given record is deleted.

A listener implementing these methods can be registered using the `addRecordListener()` method of the corresponding record store, where the only parameter is the `RecordListener`. You can unregister the listener by calling the `removeRecordListener()` method of the record store with the same listener as a parameter.

Storing Custom Objects

Up to this point, you have only stored byte arrays or strings in a record store. As explained in the diary example, `String`s have a method with which you can convert them to byte arrays, and a constructor that takes a byte array as input. But if you want to store your own objects, you need to provide a way to convert them to byte arrays and back yourselves.

The simplest solution would be to add the same constructor and `getBytes()` method to your custom objects that `String` already provides. You can implement the mapping two different ways. One possibility is to convert the byte array to different fields in your object. The other possibility is to keep the byte array as storage, and to implement field access methods as wrappers to portions of the byte array. The second possibility makes sense especially when your structure consists of elements having a fixed size. Here, we will focus on the first method, which is simpler to handle, especially for dynamic structures.

Data Streams

Assume you are going to implement a travel list management tool, where the entries consist of the journey destination, the date of the journey and the distance actually traveled.

Such an object could be implemented as follows:

```
class Journey {
```

```
    int distance;
    long date;
    String destination;
}
```

The simplest mechanism to serialize this kind of object is provided by a combination of a `DataOutputStream` and a `ByteArrayOutputStream`. The `DataOutputStream` allows you to write simple data types such as integers or byte arrays to an `OutputStream`, and the `ByteArrayOutputStream` is a special `OutputStream` that creates a byte array. To get a byte array representation of the whole object, you plug them together in the `getBytes()` method of your class `Journey`. There, you write the fields to the stream and finally obtain the byte array from the `ByteArrayOutputStream`:

```
byte [] getBytes() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream (baos);
    dos.writeInt (distance);
    dos.writeLong (date);
    dos.writeUTF (destination);
    dos.close();
    baos.close();
    return baos.toByteArray();
}
```

In the constructor, you can use a `ByteArrayInputStream` and a `DataInputStream` to perform the inverse operation:

```
Journey (byte [] data) throws IOException {
     ByteArrayInputStream bais = new ByteArrayInputStream (data);
     DataInputStream dis = new DataInputStream (bais);
     distance = dis.readInt();
     date = dis.readLong();
     destination = dis.readUTF();
     dis.close();
     bais.close();
 }
```

Direct Encoding

An alternative to using a set of streams—which may be faster but which also generates more implementation effort—is to decode the information in the byte array manually. For example, if the integer storing the distance is contained in the first four bytes of the byte array, it can be decoded using the following line:

```
distance = ((data [0] & 0xff) << 24) | ((data [1] & 0xff) << 16)
           | ((data [2] & 0xff) << 8) | (data [3] & 0xff);
```

The corresponding inverse operation is

```
data [0] = (byte) (0xff & (distance >> 24));
data [1] = (byte) (0xff & (distance >> 16));
data [2] = (byte) (0xff & (distance >> 8));
data [3] = (byte) (0xff & distance);
```

Because of the additional implementation effort, this method makes sense only in time-critical applications where the overhead of creating two additional streams is too expensive.

Ordered Traversal: Comparators and Record Enumerations

So far, you've accessed records by their primary index only. But what if you want to sort the journeys in the previous example's travel list by the length of the trip or the date, without requiring that they all be entered in the correct order? For this purpose, RMS provides *record enumerations.*

Record enumerations let you visit all the records in a record store in a custom order. The first step is to define the custom order. You do so by implementing the `compare()` method of the `RecordComparator` interface, which defines an order for the records. The `compare()` method takes two byte arrays as parameters. It returns one of the constants `EQUIVALENT`, `FOLLOWS`, or `PRECEDES`, depending on the relative order of both records. If in the desired search order the record given as the first parameter follows the second, `FOLLOWS` must be returned. The other cases are analogous.

For the `Journey` class, you can implement a `RecordComparator` that sorts all entries by date as follows:

```
public class JourneyDateComparator implements RecordComparator {
    public int compare (byte[] rec1, byte[] rec2) {
        Journey journey1 = new Journey (rec1);
        Journey journey2 = new Journey (rec2);
        if (journey1.date > journey2.date) return FOLLOWS;
        if (journey1.date == journey2.date) return EQUIVALENT;
        else return PRECEDES;
    }
}
```

Here, direct access to the portion of the byte array where the date is encoded may result in improved performance. However, the readability of the example would also suffer.

By giving the `RecordComparator` to the method `enumerateRecords()` in the class `RecordStore`, you can obtain a `RecordEnumeration`. The `RecordEnumeration` provides methods to move forward and backwards in the `RecordStore` with respect to the order defined by the `Comparator` implementation.

The `enumerateRecords()` method takes three parameters: a `RecordFilter`, the `RecordComparator`, and a `boolean` value determining if the `RecordEnumeration` should be kept updated, reflecting changes of the record store performed during traversal. There may be a significant tradeoff in speed when setting the "keep updated" parameter, but the parameter can be useful when the record store is changed during traversal. The `RecordFilter` parameter allows enumeration of a subset of the records. (It's explained in the next section.) Both the `RecordFilter` and `RecordComparator` parameters can be set to `null`, resulting in an unfiltered, unordered enumeration.

The `RecordEnumeration` returned from `enumerateRecords()` can be traversed using the `hasNextElement()` and `nextRecord()` methods. When the record enumeration is no longer needed, the application should call `destroy()` in order to notify the system that system resources allocated for the record enumeration can be released.

The following code snippet shows how your `JourneyDateComparator` and the `RecordEnumeration` can be used to traverse a travel record store, ordered by journey date:

```
RecordStore travelList = RecordStore.open ("travelList", true);
RecordEnumeration enumeration = travelList.enumerateRecords
                   (null, new JourneyDateComparator(), false);

while (enumeration.hasNext()) {
    Journey journey = new Journey (enumeration.next())
    // place real operations here
}
enumeration.destroy()
travelList.closeRecordStore();
```

You can also add methods for comparing the destination (or distance), allowing you to iterate the records ordered correspondingly. Table 5.4 shows a general overview of the methods of the `RecordEnumeration` class.

Table 5.4. Methods of `RecordEnumeration`

Name	***Purpose***
`void destroy()`	Releases system resources associated with this enumeration.
`boolean hasNextElement()`	Indicates whether records are left in the enumeration.
`boolean hasPreviousElement()`	Analogous to `hasNextElement()`, but in the inverse direction.
`boolean isKeptUpdated()`	Indicates whether the enumeration is kept updated.
`void keepUpdated (boolean keepUpd)`	Sets or resets the automatic update mode.
`byte[] nextRecord()`	Returns the next record.
`int nextRecordId()`	Returns the ID of the next record.
`int numRecords()`	Returns the number of records in this enumeration. Differs from `RecordStore.getNumRecords()` for filtered or outdated enumerations only.
`byte[] previousRecord()`	Returns the previous record.
`int previousRecordId()`	Returns the ID of the previous record.
`void rebuild()`	Rebuilds the enumeration to reflect all changes. Performed automatically in the updated mode.
`void reset()`	Resets the enumeration to the first record.

Filtered Record Enumerations

You have already seen that the `enumerateRecords()` method has a filter parameter, but we have not explained how to use it. The corresponding `RecordFilter` interface works similarly to the `RecordComparator` interface. It provides a single method, `matches()`, which takes a record byte array as input and returns a `boolean` value, determining whether the given record passes the filter.

For example, if you need an enumeration of the journeys where the distance was greater than x kilometers, you can implement it as follows:

```
Class MinDistanceFilter {
    int min;
    DistanceFilter (int min) {
        this.min = min;
    }
    boolean matches (byte[] record) {
        return new Journey (record).getDistance() >= min;
    }
```

```
}
```

The Search Problem

In the previous sections, you learned to sort and filter records in a record store. Obviously, you can search a record by iterating all records until the desired record is found, or just set a filter that filters out all other records.

Unfortunately, this approach requires looking at all records in the worst case and looking at half of the records in the average case. A search on a sorted set of records can be performed much faster by an algorithm called a *binary search*. The algorithm goes to the middle of the set and determines whether the element searched is greater or smaller than the record in the middle. Thus, the algorithm can decide in which half of the records the searched element is located. For this half, the procedure is applied again. Step by step, the number of records in question is reduced by half. So, compared to a linear search, the overall number of comparisons is reduced to a logarithmic function. For 100 records, a linear search requires 100 comparisons in the worst case and 50 in the average case; a binary search requires only 7 comparisons. This factor increases with larger number of records.

The record enumeration does not provide any means to jump over half of the records and go to the middle. If fast searching in ordered sets of records is important for your application, consider sorting the whole record store and performing a binary search instead of iterating record enumerations. In Chapter 9, "Advanced Application: Blood Sugar Log," we will present the corresponding algorithms.

Summary

In this chapter you learned about persistent storage of application data. You now know how to use the methods of the class `RecordStore` for basic access to the RMS. You know how to convert custom classes to byte arrays and back and how to iterate, filter, and order record stores using enumerators.

The next chapter explains how to make connections to other devices or the Internet using the generic connection framework.

Chapter 6. Networking: The Generic Connection Framework

IN THIS CHAPTER

For the Java 2 Standard Edition, the classes for handling network connections are located in the `java.net` package. This package contains a lot of different classes. It includes at least one class for each type of connection, such as socket connections, HTTP connections, datagram connections, and server sockets. It also contains many support classes, for instance classes for handling URLs or decoding Internet addresses. In sum, the `java.net` package includes more than 20 classes, interfaces, and exception classes.

The huge amount of classes and interfaces that is needed to support network capabilities would be too much to be adopted for the Java 2 Micro Edition. Thus, the *Connected Limited Device Configuration (CLDC)* takes a different approach: Instead of providing one class for each protocol like J2SE, CLDC offers a uniform approach for handling connections, the so-called *Generic Connection Framework (GCF).* GCF contains only one generic `Connector` class. The `Connector` class takes a URI as input and returns a corresponding connection object, depending on the protocol parameter of the URI string. The protocol parameter of a URI is the part from the beginning of the string to the first colon. For example, for an HTTP connection, the protocol parameter is the leading `http` of an address such as `http://java.sun.com`. The general form of URI strings that are passed to the `Connector` class is as follows:

```
<protocol>://<address>;<parameters>
```

The syntax of the strings that are passed to the `Connector.open()` method needs to follow the *Uniform Resource Indicator (URI)* syntax that is defined in the IETF standard RFC2396. The complete RFC can be found under the following URI: http://www.ietf.org/rfc/rfc2396.txt.

Note

The CLDC itself specifies interfaces, classes, and exceptions of the GCF only. No implementations of any concrete connection type are provided at the configuration level.

Table 6.1 gives an illustrative overview of connection types that can be implemented by a particular J2ME profile such as the MID or PDA Profile. J2ME profiles might also include additional protocols that are not listed in Table 6.1.

Table 6.1. Example URLs for the `Connector.open()` Method of the Generic Connection Framework

Protocol	*Sample String Parameter for `Connector.open()`*	*Connection Type*
HTTP	`http://java.sun.com`	`HttpConnection`
Sockets	`socket://time-a.nist.gov:13`	`StreamConnection`
ServerSockets	`serversocket://:4444`	`StreamConnectionNotifier`
Serial	`comm:0;baudrate=2400;`	`CommConnection`
Datagrams	`datagram://127.0.0.1`	`DatagramConnection`

File	`file://addesses.dat`	`FileConnection`
Bluetooth	"`bluetooth://psm=1001`"	`StreamConnection`

The only connection type that is guaranteed to be available in MIDP 1.0 is the HTTP protocol. PDAP adds specialized interfaces for access to serial ports and file systems. Whether a certain protocol is actually available depends on the device. For example, it does not make sense to support the comm protocol on devices without a serial port.

Creating a Connection—The Connector Class

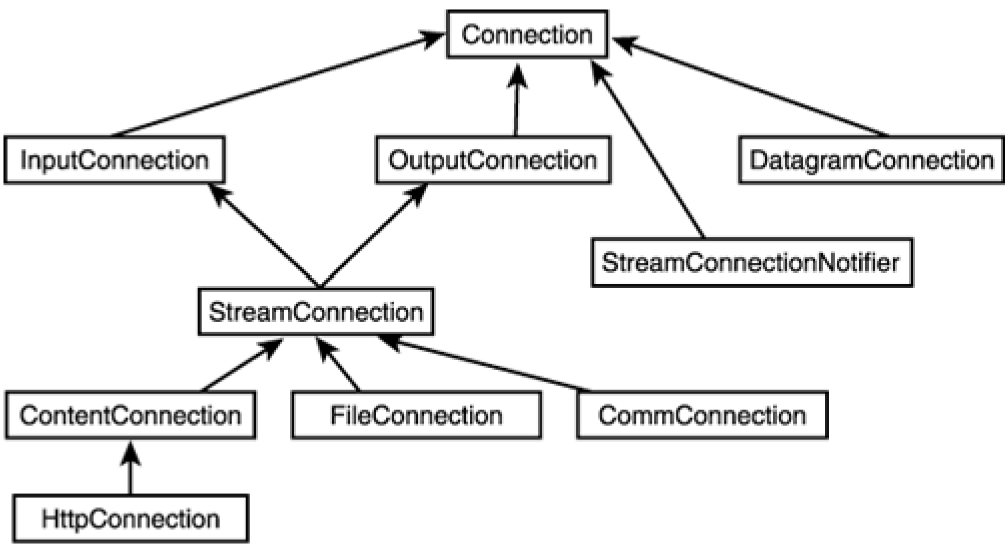
In the GCF, connections are established using the class `Connector`. By passing a URL describing the protocol to the open method of the `Connector` class, a connection is established. For example, the following line opens a *Hypertext Transfer Protocol (HTTP)* connection to the address `http://java.sun.com`:

```
try {
   Connection connection = Connector.open ("http://java.sun.com");
}
catch (IOException e) {
    // an error occurred while opening the connection.
}
```

In the case of an error, an `IOException` is thrown. Most of the GCF methods can throw an `IOException` in order to report I/O errors to the application.

If the connection is established successfully, an instance of a class implementing the `Connection` interfaces is returned. Other special interfaces derived from `Connection` are available for different connection types like datagram or stream connections. Figure 6.1 gives an overview of the corresponding GCF interfaces.

Figure 6.1. The connection interfaces of the GCF.



The general `Connection` interface itself does not implement much functionality. It only defines the `close()` method that is needed to terminate a connection that was opened using the `open()` method of the `Connector` class. The functionality for reading and writing data is added in `InputConnection` and `OutputConnection`. The `InputConnection` defines two methods for reading data from the connection: `openInputStream()` returns an `InputStream`, and `openDataInputStream()` returns a `DataInputStream`. Analogously, the `OutputConnection` provides two methods, `openOutputStream()` and `openDataOutputStream()`, returning the corresponding streams for writing data.

Probably the most important connection interface is the `StreamConnection`, combining the `InputConnection` and `OutputConnection` in one interface. It is used directly or as base interface for most connection types. It is the base interface for the `ContentConnection`, which adds functionality for accessing the meta-information available for HTTP connections. For datagram connections, the GCF provides the `DatagramConnection` interface, representing an endpoint for datagrams. Finally, the GCF supports the interface `StreamConnectionNotifier` that corresponds to J2SE `ServerSocket`s. This interface consists of only one `acceptAndOpen()` method. This method returns a `StreamConnection` when a client has successfully established a connection.

For filesystem and serial port connectivity, the PDAP provides the `FileConnection` and `CommConnection` interfaces. These connections are optional to a PDAP implementation depending on the underlying hardware and operating system.

We have already shown the simple `open()` method of the `Connector` class, taking a URI string as input and returning an instance of a class implementing the `Connection` interface. However, this is not the only `open()` method available; the `Connector` class provides two additional `open()` methods. The second variant of `open()` takes an additional mode integer constant that determines whether the connection is read-only, write-only, or a read-write connection. The corresponding constants are listed in Table 6.2. The last variant of the `open()` method takes a third parameter, indicating that the protocol may throw an `InterruptedIOException` in the case of a device-specific timeout. Without using additional parameters, the behavior of the `open()` method is to open a connection in `READ_WRITE` mode, without throwing timeout exceptions.

The optional parameters depend on the protocol to which they are passed. For instance, the connection mode `WRITE` might result in an `IllegalArgumentException` if it is used to open a protocol for accessing a bar code scanner allowing read-only access.

Table 6.2. Possible Access Modes for Opening a Connection Using the GCF

Mode	*Description*
`READ`	Read-only connections
`READ_WRITE`	Connections using read and write access
`WRITE`	Write-only connections

We already mentioned that a concrete instance of a class implementing the `Connection` interface is returned when a connection is established successfully. In order to access the functionality of the concrete type of connection, the returned `Connection` needs to be typecast to the corresponding sub-interface. For example, when opening an HTTP connection, it usually makes sense to cast the returned object to the more specific `HttpConnection` interface:

```
HttpConnection httpConnection =
    (HttpConnection) Connector.open ("http://java.sun.com");
```

In the following section, we will take a closer look at the different connection types that are listed in the CLDC specification.

Connection Types

Before describing the mandatory HTTP protocol and other connections, we will cover the optional socket connections because they form the basis for higher-level protocols such as HTTP or FTP. Actually, it might be possible to create a CLDC-based device without display, implementing a Web server as a user interface. Examples might be configurable network routers or production control hardware.

Note

We cannot cover all the underlying protocols in full detail here, so we will mainly focus on the Java API. For complete coverage, refer to the corresponding standard specifications or other protocol specific literature.

General Socket Connections

Socket and datagram connections are simple basic IP connection types. Although sockets and datagrams are usually available on machines connected to the Internet, for wireless connections the situation is different. Carriers often use a proprietary protocol over the air, and allow only special connection types such as HTTP connections. Thus, although sockets are usually required for HTTP connections, CLDC devices might provide HTTP but no socket connections. The main difference between sockets and datagrams is that sockets are a reliable and end-to-end connection established for a period of time, whereas datagrams are simple data packets with a limited length that might never reach their destination without notifying the sender.

TCP socket connections are usually not symmetric, but a client connects to a server. The server listens to a specific port, usually depending on the protocol that is used. For example, the default port used by HTTP servers is 80, and the default port for FTP is 21. A simple method to test the socket functionality in general is to connect to an existing server and to display what the server sends. Here, a good candidate is the clock server, usually listening on port 13. For each incoming connection, it sends its current date and time back to the client. If a certain server supports the time, service can easily be tested by connecting to that port with a simple terminal program (telnet). You can find a list of time servers at http://www.boulder.nist.gov/timefreq/service/time-servers.html.

Here, we will first describe the client side of TCP/IP sockets and then sketch how to set up a server socket. (For general coverage of TCP/IP, refer to *TCP/IP Unleashed* by Karanjit Rom Siyan and Tim Parker, ISBN 0672323516.)

Client-Socket Connections

The parameter given to the `open()` method in order to establish a socket connection at the client side consists of the protocol identifier (such as `socket://`), the IP address of the server that will be connected (such as `time-a.nist.gov`), and a port number that is separated from the host address using a colon (`:13`). For example,

```
StreamConnection streamConnection =
    (StreamConnection) connector.open ("socket://time-a.nist.gov:13");
```

When a socket connection is opened, the returned instance always implements the `StreamConnection`'s interface, providing access to corresponding input and output streams.

In the following code snippet, a socket connection to a time server is established, and then the time is read from the corresponding input stream:

```
try {
    Connection connection = Connector.open ("socket://time-
a.nist.gov:13");
```

```
    // converting the Connection to a StreamConnection
    StreamConnection streamConnection = (StreamConnection)connection;

    // getting Input
    InputStream in = streamConnection.getInputStream();

    // reading and writing of data
    StringBuffer buf = new StringBuffer();
    while (true) {
        int i=in.read();
        if (i==-1) break;
        buf.append ((char) i);
    }
    System.out.println ("server time: "+buf);
    // closing the streams and the connection
    in.close();
    streamConnection.close();
}
catch (IOException e) {
    // Handle the exception occurred while opening a socket
connection
}
```

Note

For reading character data, a `reader` is usually adequate. For obtaining a reader from an input stream, use the class `InputStreamReader`. The same holds for output streams, writers, and the class `OutputStreamWriter`.

For a more elaborate example including a user interface, refer to the sample application presented in the section "GCF Terminal Program."

Server Sockets

The counterpart to client sockets are server sockets. Server sockets listen to a fixed TCP/IP port number. When a client requests a connection to the port that the server socket is listening to, the server can accept the request and establish a connection. In GCF, a server socket is established by giving `serversocket://:`, followed by the desired port number, to the `open()` method of the `Connector` class. The returned `Connection` object needs to be cast to the interface `StreamConnectionNotifier`, which provides the method `acceptAndOpen()`. This method blocks the application until a client socket connects to the port the server is listening to. When a client establishes a connection, a `StreamConnection` to the client is returned.

In order to demonstrate the use of the server sockets, we have implemented a rudimentary HTTP based server. Thus, a standard Web browser can be used to connect to the server by entering a URL that points to the machine where the HTTP server is running and on which port the server is listening to. An example URL might look as follows:

```
http://<ip-number>:<portnumber>
```

The `HTTPServer` is extended from the `Thread` class, making it rather simple to integrate the `HTTPServer` in a MIDP or PDAP application of your own. Using the following code snippet, you will be able to start the `HTTPServer` on port `8080`:

```
new HTTPServer(8080).start();
```

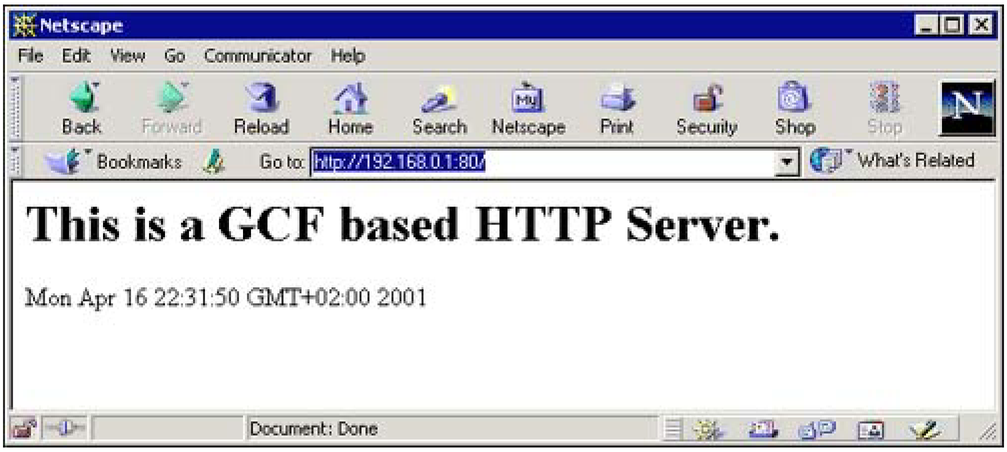
If there is currently a `HTTPServer` listening to the specified port, an `IOException` will be thrown as described in the section "Creating a Connection." To track the status of the `HTTPServer` that is printed to the console using the `System.out.println()` method, you can just write the results into a text widget of MIDP or PDAP.

The HTTP server application can also be run on a desktop machine without a MIDP or PDAP environment emulated. In order to do so, you need either to compile the SUN GCF for the desktop, or to access the ME4SE CLDC emulation available from http://www.me4se.org.

A short description of the strings that are transferred between client and server is given in the next section, "HTTP Connections."

Listing 6.1 shows the complete source code of the HTTP server. Figure 6.2 shows the HTML sample page that is sent to a Web browser by the HTTP thread if it is included in a server application.

Figure 6.2. The output of the HTTP server implementation shown in Listing 6.1 when connected with a Netscape 4.7 Web browser.



Listing 6.1 HttpServer.java—A Simple HTTP Server Thread Sending a Headline and the Actual Time to the Connecting Web Browser as an Identification String

```
import java.io.*;
import java.util.*;
import javax.microedition.io.*;

public class HttpServer extends Thread {

    StreamConnectionNotifier serverConnection;
    int port;

    public HttpServer (int port) throws IOException {
        this.port = port;
        serverConnection =
            (StreamConnectionNotifier) Connector.open
                ("serversocket://:"+port);
    }

    public void run() {
        try {
            while (true) {
                System.out.println ("Waiting for connection on port
"+port);
                StreamConnection clientConnection =
```

```
                    serverConnection.acceptAndOpen();
                System.out.println ("Connection to established.");
                sendAnswer (clientConnection);
            }
        }  catch (IOException e) {
            System.out.println (e.toString());
        }
    }

    public void sendAnswer (StreamConnection s) throws IOException {
       OutputStream o = s.openOutputStream();
       Date date = new Date (System.currentTimeMillis());
       o.write (("HTTP/1.1 200 OK\r\n\r\n"
                 +"<HTML><H1>This is a GCF based HTTP Server.</H1>"
                 + date.toString() + "\r\n\r\n").getBytes());
       o.close();
       s.close();
       System.out.println ("Connection closed.");
   }
}
```

Note

Please note, that the procedure described previously is based on the CLDC 1.0 spec. MIDP 2.0 introduces a new `PushRegistry` class that can to be used with the `PushListener` interface in order to establish inbound socket connections.

HTTP Connections

HTTP is the only communication protocol that must be supported by all MIDP and PDAP devices. The HTTP support must include the HTTP protocol version 1.1 and the `HEAD`, `GET`, and `POST` requests as described in RFC2616, which can be found at http://www.ietf.org/rfc/.

HTTP is the default protocol for transmitting HTML Web pages. The HTTP protocol is based on a request/response paradigm where a client establishes a connection to a server that is listening on the TCP port number 80 by default. Both the HTTP request and response consist of three parts: the request or response line, header entries, and the actual payload data. The payload is separated from the header by a single empty line. Line breaks are indicated by a sequence consisting of a carriage return and a line feed control character (`\r\n`).

The client initiates the HTTP connection by sending a request line. The request line consists of a method (for example, `GET`), the URL identifying the requested page, and the protocol version, all separated by space characters. The following lines contain header lines and the actual payload. The server processes the request and sends back a response to the client containing the status of the server, its own header information, and the requested information.

The following sample lines, taken from an actual HTTP transfer, illustrate the protocol. Note that the request payload is empty in this simple case. Actually, the request payload is empty in most HTTP requests for HTML pages except from `POST` requests transmitting, for example, form data to the server.

A client sends a request for the root page (/) with an additional header indicating the device profile and configuration:

```
GET / HTTP/1.1
User-Agent: Profile/MIDP-1.0 Configuration/CLDC-1.0
```

The server response of the server is

```
HTTP/1.1 200 OK
Date: Sat, 31 Mar 2001 21:09:48 GMT
Server: Apache/1.3.14 (Unix)
Last-Modified: Tue, 09 Jan 2001 09:29:10 GMT
ETag: "cd610-1fbc-3a5ad9e6"
Accept-Ranges: bytes
Content-Length: 8124
Connection: close
Content-Type: text/html

<html>
...
</html>
```

The request that is sent back from the server to the client can be split into three sections:

- The response line indicating the protocol version and status code:
- `HTTP/1.1 200 OK`
- The header containing meta-information provided by the server:
- `Date: Sat, 31 Mar 2001 21:09:48 GMT`
- `Server: Apache/1.3.14 (Unix)`
- `Last-Modified: Tue, 09 Jan 2001 09:29:10 GMT`
- `ETag: "cd610-1fbc-3a5ad9e6"`
- `Accept-Ranges: bytes`
- `Content-Length: 8124`
- `Connection: close`
- `Content-Type: text/html`
- The actual content (*entity* or *resource*):
- `<html>`
- `...`
- `</html>`

The `HTTPConnection` Interface

The GCF provides a special interface for HTTP connections, the `HTTPConnection`. `HTTPConnection` is a specialization of the `ContentConnection` interface, which is a specialization of `Connection`.

The advantage of specialized HTTP connection is that it provides comfortable access methods to the header fields of the connection. Only the actual content is transferred using streams. If the socket protocol is supported for a device, it is also possible to connect to HTTP servers using the socket protocol. However, in that case, you would need to enter the additional protocol lines manually along with the actual content of the transfer, as shown in our server socket example in the previous section.

In addition to the access methods derived from the `StreamConnection`, a `ContentConnection` provides the following methods for getting meta-information:

- `getType()`— Returns a `String` denoting the MIME type of the content, or `null` if there is no such information available. In the preceding example, the content type is `text/html`.
- `getEncoding()`— Returns a `String` denoting that character encoding is used for the content, for example `UTF-8`. If this information is not available, `null` is returned. The encoding information may be used in the InputStreamReader constructor for reading character-based data.

- `getLength()`— Returns a `long` value returning the actual length of the content in bytes. If this information is not available, the method returns `-1`.

The `HTTPConnection` specializes the `ContentConnection` interface further, adding support for HTTP methods and header fields. Before we look into the additional methods in more detail, we need to describe the three possible states of this type of connection:

- Setup—The connection to the server has not yet been established.
- Connected—All request parameters have been set and sent to the server, and the client expects a response from the server.
- Closed—The connection is closed, and all methods invoked on the `HTTPConnection` will throw an `IOException`. Note that the connection and all streams obtained should be closed.

Depending on the state, only a subset of the methods provided by `HTTPConnection` may be called without causing a state transition or even an exception.

Note

A special feature of HTTP 1.1 that most HTTP 1.0 servers do not understand is chunked encoding. Basically, chunked encoding means that parts of a request are sent separately with additional length information. The additional length information is interpreted by HTTP 1.0 servers as content, letting the server fail in interpreting the content. Some MIDP implementations such as the SUN wireless toolkit switch to chunked encoding when the amount of data sent exceeds a fixed limit, or when the `flush()` method is called on the output stream. In many cases, the problems with HTTP 1.0 servers can be avoided by not calling `flush()`.

Request Properties

The `setRequestMethod()` can be used to set the connection to `GET`, `POST`, or `HEAD` only in the Setup state. The same holds for `setRequestProperty()`, which sets the request headers that need to be initialized before a connection is established. The following line contains an example for setting the `User-Agent` header:

```
setRequestProperty ("User-Agent", "Profile/MIDP-1.0
Configuration/CLDC-1.0");
```

This call causes the following header line to be sent with the request when the connection is established:

```
User-Agent: Profile/MIDP-1.0 Configuration/CLDC-1.0
```

The `HTTPConnection` will change to the Connected state when one of the methods for sending and receiving data, like `openInputStream()` or `openOutputStream()`, is called. The transition to the Connected state is also performed when the header fields of the server response are accessed. For that purpose, the `getHeaderField()` method can be used. The following line of code would return the `String` `"Apache/1.3.14 (Unix)"` for our example request:

```
String serverType = getHeaderField("Server");
```

The complete source code for creating the example request is as follows:

```
try {
    HttpConnection httpConnection =
        (HttpConnection) Connector.open ("http://www.leo.org/");
    httpConnection.setRequestProperty
```

```
        ("User-Agent", Profile/MIDP-1.0 Configuration/CLDC-1.0);
    InputStream is = httpConnection.openInputStream();

    // read data from server here

    is.close();
    httpConnection.close();
}
catch (IOException e) {
    // put exception handling here...
}
```

The most important request property is the Request method. The default Request method is `GET`. The `GET` request method is used by Web browsers for requesting HTML pages from Web servers. HTTP does not allow `GET` requests to have side effects on the server. Thus, if data will be submitted to the server, the `POST` method must be used. For example, Web browsers use `POST` requests for sending the content of HTML forms to the server. HTML editors can use `POST` requests to update HTML pages stored at the server. The `openOutputStream()` method can be used to open a stream for writing data to the server.

A complete application example for running a chat system over HTTP using the `GET` and `POST` methods is given in the section "A Simple HTTP Based Client-Server Chat Application." For full coverage of the HTTP protocol refer to RFC2616, which can be found at http://www.ietf.org/rfc.

Datagram Connections

Datagram connections provide a mechanism for transferring simple data packets between two applications. In contrast to TCP socket connections, UDP datagram connections are not reliable. Thus, `DatagramConnection`s can only be used for connections where packet losses are acceptable. Typical applications of datagram connections are streaming or real-time applications. The advantage of datagrams is that their transport produces less protocol overhead. Thus, the performance for datagrams can be higher than TCP performance.

Datagram connections can be used as server and client connections as well. The protocol name for both client and server datagram connections is `datagram://`. For client connections, the protocol name is followed by the host address and the port, separated by a colon. For server connections, the host is omitted and just the colon and the port are given. For example, a client datagram connection to port `1234` of the server `myserver.some.com` is set up by the following line of code:

```
DatagramConnection datagramConn =
    (DatagramConnection)Connector.open
("datagram://myserver.some.com:1234");
```

Each datagram that is sent over a datagram connection is a small data packet consisting of the destination address and a buffer containing the payload data. In J2ME, datagrams are encapsulated in the `Datagram` class. `Datagram` objects are created using the `newDatagram()` method of a datagram connection. The `newDatagram()` method takes the buffer containing the data and the size of the buffer to be transferred. It then returns a new `Datagram` object. A datagram that is created in this way automatically contains the receiver address of the connection.

The following snippet creates a `Datagram` object containing a `"Hello World"` string:

```
String helloWorldString = new String ("Hello World");
byte[] buffer = helloWorldString.getBytes();
Datagram myDatagram = datagramConn.newDatagram(buffer, buffer.length);
```

Datagrams are sent using the method `send()` of the `DatagramConnection`:

```
datagramConn.send(myDatagram);
```

To receive datagrams, you need to create a datagram server connection. For that purpose, you again use the `Connector.open()` method, but this time without specifying the hostname of the machine. Instead, you just give the local port number your server will listen to:

```
DatagramConnection serverDatagramConn =
    (DatagramConnection)Connector.open ("datagram://:1234");
```

In order to receive a datagram, you need to call the `receive()` method, which blocks until a datagram is received. Before you call the `receive()` method, you need to create a datagram object that is passed to this method. The `receive()` method fills the given object from the datagram received. Now let's assume that you expect the client to send a datagram containing a `"hello world"` string consisting of 11 bytes. To create a datagram that is able to hold the complete buffer, you need to create an empty datagram with a buffer size of at least 11 bytes using the following line of code:

```
Datagram receivedDatagram = serverDatagramConn.newDatagramConn (11);
```

Finally, you need to call the `receive()` method of the `DatagramConnection` to receive the datagram sent from the client and convert the contained buffer back to a `String`:

```
serverDatagramConn.receive(receivedDatagram);
String helloWorldString = new String(receivedDatagram.getData());
```

The `helloWorldString` variable should now contain the `"Hello World" String` that was sent from the client.

As a real-world example, we will again use the server `time-a.nist.gov`. It supports not only socket connections, but also the datagram time protocol specified in RFC868. This protocol returns a 32-bit binary number that represents the time in seconds since January 1, 1900.

If you want to compare the server time to the local time, you need to convert the server time to the usual Java format, which is measured in milliseconds since January 1, 1970. Thus, you must subtract the 2,208,988,800 seconds between 1.1.1900 and 1.1.1970 from the server time, and then multiply the result by 1,000 in order to convert the seconds to milliseconds.

The following code example establishes a corresponding `DatagramConnection` to the time server `datagram://time-a.nist.gov:37`:

```
try {
    Connection connection = Connector.open ("datagram://time-
a.nist.gov:37");

    // converting the Connection to a DatagramConnection
    DatagramConnection datagramConnection =
(DatagramConnection)connection;

    // creating a new datagram in order to request the time from the
server
    Datagram datagram = datagramConnection.newDatagram
        (datagramConnection.getNominalLength());

    // sends the datagram to the server
    datagramConnection.send (datagram);

    // for the response that is sent from the server
```

```
    // we need to create a new datagram that can hold 4 bytes
    Datagram respDatagram = datagramConnection.newDatagram (4);

    // receives the datagram containing the actual time
    datagramConnection.receive (respDatagram);

    // in order to convert the 4 bytes that are
    // received to a long we need the following lines
    byte[] received = respDatagram.getData();
    long time = (((((long) received [0]) & 0x0ff) << 24)
        + ((((long) received [1]) & 0x0ff) << 16)
        + ((((long) received [2]) & 0x0ff) << 8)
        + ((((long) received [3]) & 0x0ff)));

    // Convert seconds since 1.1.2000 to Milliseconds since 1.1.1970
    time = (time - 2208988800l) * 1000;

    // calculate deviation
    long difference = time – System.currentTimeMillis();

    // closes the datagram connection
    datagramConnection.close();
}
catch (IOException e) {
// Handle the exception that occurred while opening a datagram
connection
}
```

Serial Connections

Some devices running a CLDC KVM might be equipped with a serial port and support the protocol for serial connections. In order to transfer data over the serial port, the device needs to be connected to a desktop computer or to another device that supports serial communication—for example, a GPS receiver. In order to test serial communication, a simple terminal can be used.

PDAP specifies that a PDAP implementation can support the `CommConnection` if supported by the underlying operating system and hardware. Such a `CommConnection` can be established as shown in the following code snippet that opens a 9600 baud communication connection on comm port 0 with 8 bits per character and no parity

For example,

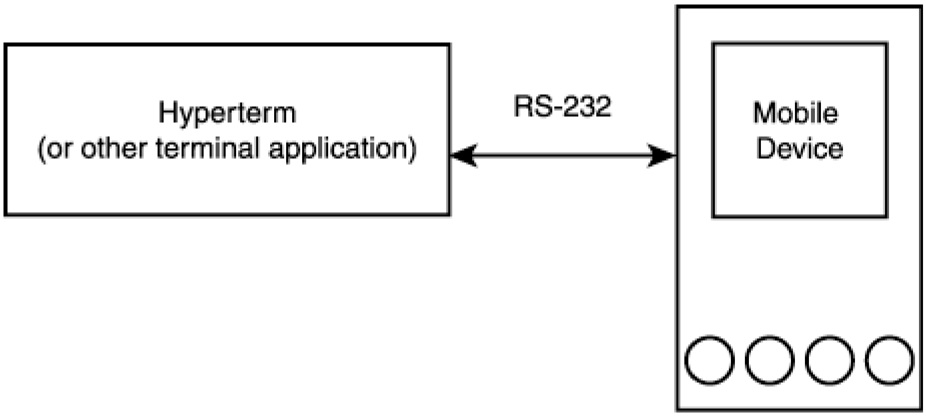
```
CommConnection connection =
   (CommConnection) Connector.open
   ("comm:0;baudrate=9600;bitsperchar=8;parity=none");
```

The actual serial port properties depend on the device's hardware. A comma-separated list of all available comm ports can be obtained from the system property "microedition.commports" using the `System.getProperty()` method. These comm ports might not only include physical RS-232 ports, but also IR or Bluetooth ports that are mapped to serial ports. Commonly supported parameters are

- Number of the comport: comm=0..n or IR=0..n
- baudrate=57600, 38400, 19200, 9600, 2400, 1200
- bitsperchar=8, 7
- parity=even, odd, none

They need to match the connection properties of the connected device. For debugging purposes, the mobile device can easily be connected to the desktop PC using a serial RS232 connection, as shown in Figure 6.3. Using a terminal program such as Hyperterm (included in MS Windows), the transferred data can be visualized on the host computer system.

Figure 6.3. The mobile device connected to a desktop PC for sending and receiving data using a terminal application such as Hyperterm.



Note

The CommConnection that is used in PDAP is also available in MIDP 2.0.

File Connections

A new kind of connection that is added to the Generic Connection Framework by PDAP is the capability to access file-based memory cards or the internal file system of a device. Currently, most of the new handheld devices are equipped with some kind of card reader supporting one or even two different brands of removable media cards such as

- CompactFlash cards
- MultiMedia cards
- Secure Digital cards
- SmartMedia cards
- and MemorySticks

Those cards support a hierarchical file system similar to the file system known from desktop computers. PDAs may also provide an internal file system.

For access to file systems, the three interfaces `FileConnection` , `FileSystemEvent` and `FileSystemListener` are included in PDAP. Those interfaces and the additional `FileSystemRegistry` class are described in detail in the following sections.

The `FileConnection` Interface

The `FileConnection` interface defines methods for handling files on removable media, similar to the functionality provided by the `java.io.File` class of the Java 2 Standard Edition. For opening a file connection using the `Connector` class, a URL consisting of the following parts needs to be passed:

```
file://<host>/<path>
```

The possible root strings can be queried using the `PushRegistry.listRoots()` method. In order to simplify path concatenation they have a trailing "/".

The following list shows examples of possible root strings:

- `CFCard/`
- `SDCard/`
- `MemoryStick/`
- `C:/`
- `usr/`
- `sda1/`

When compared with other GCF connections, the connection behavior of the file connection differs to some extent. The `Connector.open()` method can successfully return a `FileConnection` object although the referenced file or directory does not yet existent. This behavior is necessary for creating files and directories. The following code snipped shows how the `FileConnection` interface can be used to create a directory, given the device provides a "SDCard" file system root:

```
try {
    FileConnection fileConn =
        (FileConnection)Connector.open("file:///SDCard/MyDir/");
     if (!fileConn.exists())
         fconn.mkdir();
     fconn.close();
}
catch (IOException ioe) {
    // handle the error that occurred during directory creation
}
```

If a platform does not support file connections, it will throw a `javax.microedition.io.ConnectionNotFoundException` when an application tries to open a particular connection using `Connector.open()` method.

The `FileConnection` is derived from the `StreamConnection` interface and supports its methods for opening input and output streams. Furthermore, the interface defines methods such as `create()` and `delete()` for creating and deleting files, and methods to check the capability of reading and writing of a file such as `canRead()` and `canWrite()`. Also methods to query information about the memory space available and currently occupied are provided.

FileSystem Security

The security model supported for `FileConnections` is device and implementation dependent. Accessing a file connection is restricted to prevent unauthorized access and manipulation of data. Implementations must prohibit access to RMS databases, private data, and internal files of the operating system. The security model may be applied already at the invocation of the `Connector.open()` method. When the URL passed to the `Connector.open()` method points to a file or directory that is not allowed to be accessed, a `java.lang.SecurityException` may be thrown. The security model may also be applied later, when an application actually tries to open a stream.

Registering `FileSystemListeners` Using the `FileSystemRegistry`

Since the functionality of the `FileConnection` is defined in an interface and not in an abstract class that could hold static methods as well, the functionality to add and remove `FileSystemListeners` is contained in a separate class. The `FileRegistry` class contains three static methods. The `addFileSystemListener()` method is used to register a new `FileSystemListener`, and the `removeFileSystemListener()` method is used to deregister such a listener. Notifications from the `FileRegistry` class are important for applications like when file browsers that may want to notify the user or update the display when e.g. a memory card was inserted or removed. Whenever a file system is added or removed, the corresponding `rootAdded()` or `rootRemoved()` callback method is called. Both methods provide a `FileSystemEvent` parameter. The `FileSystemEvent` is used to identify the particular file system change that occurred. The type of the event can be retrieved using the `getID()` method returning one of the constants `FileSystemEvent.ROOT_ADDED` or `FileSystemEvent. ROOT_REMOVED`. The `getRootName()` method allows you to query the file system root the event is referring to.

IRDA Connections

Unfortunately, infrared connections corresponding to the IRDA standard are covered neither by the CLDC specification nor by the MIDP or PDAP profile, except from the serial port emulation. However, it might be possible that some devices allow mapping a serial connection to the infrared port. Also, even if IRDA communication is currently not specified in CLDC, it might be provided by device manufacturers, third parties or future versions of the J2ME standards.

Bluetooth Connections

A very new connection type for mobile devices is the upcoming Bluetooth technology. The Bluetooth standard defines data exchange up to 100 meters over the air at a maximum bandwidth of 723.2kB/s. However, in order to save battery power in mobile devices, usually the pico version of the standard is implemented, which reduces the nominal range of the Bluetooth Communication to 10 meters. Bluetooth might replace the IRDA communication that is often used between mobile phones, PDAs, and notebooks because it does not need to have a direct optical connection between sender and receiver. Thus, it would be possible to use a cell phone that is placed in the user's vest pocket to connect a PDA to the Internet.

JSR-82 covering Bluetooth connectivity for J2SE and J2ME includes a Bluetooth integration into the GCF. To get more information about JSR-82, it is available for public review under the following URL: http://www.jcp.org/jsr/detail/82.jsp.

GCF Examples

Now that you know about the connection types provided by CLDC, MIDP 1.0, and 2.0, you are ready to implement two example applications: a terminal program and a chat application. These example applications will be CLDC-based.

GCF Terminal Program

Prior to the PC era, it was quite common to have a single centralized computer and a set of simple hardware terminals providing text-based interfaces to the main computer. A *terminal program* is an application that simulates this kind of interface by providing a local text interface to a remote process. Before the World Wide Web became popular, terminal programs were widely used to access so-called mailboxes or *bulletin board systems (BBS)* over a serial modem connection. Today, terminal programs implementing the telnet protocol are still used for command line access; for example, for server configuration.

For the sample terminal application, we will take advantage of the fact that the GCF is indeed very generic. You will enter a URI, and your terminal program will connect to the given address and display what the other end of the connection sends. For example, if you connect to an HTTP address, the program will display the HTML code of the requested page. You may also use the example to view the raw positioning data read from a GPS receiver using a serial connection. An input line allows you to send data over the connection.

Here, we will only describe the network related parts of the application. The complete sources of the MIDP and PDAP versions are shown in Listings 6.2 and 6.3, respectively. The `MidpTerminal` and `PdapTerminal` application user interfaces consist of a widget in which you can enter a URL that is used to open a particular connection. The connection is established by activating the `Connect` command or by pressing the `Connect` button, depending on the platform in which the application is running.

The core of the Terminal implementation is contained in an inner class of the main application called `Handler`. This class "handles" establishing the connection and receiving new data in the background. For that purpose, the `Handler` stores the connection, as well as the corresponding input and output streams in member variables. It also contains a variable `leave` that determines whether the handler should leave the receive loop and terminate itself:

```
class Handler extends Thread {

    StreamConnection connection;
    InputStream in;
    OutputStream out;
    boolean leave;
```

The constructor of the `Handler` takes a URL as input and establishes a corresponding stream connection. It is called from the main application when the user enters an address and requests a corresponding connection. The `Handler` is not able to handle datagram or serversocket protocols. Trying to do so would cause a class cast exception.

When the connection is established, the `out` and `in` variables are set. Then, the establishment of the connection is reported using the `show()` method of the main class:

```
public Handler (String url) throws IOException {

    connection = (StreamConnection) Connector.open
    (url, Connector.READ_WRITE, true);

    out = connection.openOutputStream();
    in = connection.openInputStream();

    show ("opened: "+ url + "\r");
}
```

The main work is performed in the `run()` method of the `Handler` thread. The `run()` method is invoked automatically when the main application calls the `start()` method, which runs the handler as a separate thread in the background. In the `run()` method, incoming data is collected in a string buffer until the leave flag is set, no more data is available, or the buffered data reaches a limit of 1024 bytes. In that case, the collected data is shown to the user by handing it over to the `show()` method. Then, the buffer is cleared and a new iteration of the read loop is entered. If read encounters a `-1`, that means that the stream is closed remotely. In that case, `disconnect()` is called in order to terminate the `Handler` and to release the connection:

```
    public void run() {

        StringBuffer buf = new StringBuffer();

        try {
            while (!leave) {
```

```
                do {
                    int i = read();
                    if (i == -1) disconnect();
                   // ignore control characters except from cr
                    else if (i == '\r' || i >= ' ')
                        buf.append ((char) i);
                }
                while (!leave && in.available() > 0 && buf.length() <
1024);

                show (buf.toString());
                buf.setLength (0);   // clear buffer
            }
        }
        catch (Exception e) {
            if (!leave) show (e.toString() + "\r");
            disconnect();
        }
    }
}
```

You might have noticed that the method `Handler.read()` is called for reading data from the stream instead of just calling `in.read()`. The only difference is that `Handler.read()` implements telnet parameter negotiation, which allows you to use the Terminal sample application as a telnet client by connecting to port 23 of a corresponding host. The Telnet protocol is widely used to connect computer systems remotely with a command-line interface. For details of the Telnet protocol please refer RFC854.

The main application classes, `MidpTermial` and `PdapTerminal`, mainly handle the user interface. The only method that is independent from the user interface is `disconnect()`, which closes the connection if a connection exists and notifies the corresponding `Handler` to terminate.

An important difference of the PDAP implementation when compared to the MIDP implementation is that the `show()` method does not manipulate the user interface directly. Because the PDAP AWT is not thread safe, it is necessary to manipulate the user interface indirectly by calling `invokeAndWait()` with an instance implementing the `Runnable` interface. Here, you use the class `Appender` for that purpose. The `Appender` instance encapsulates the string to be appended to the list of incoming data. When the AWT calls the `run()` method of the `Appender`, AWT has made sure that it is currently safe to manipulate the user interface. Now, the `Appender` adds its payload to the list showing the data sent from the remote end of the connection.

Figure 6.4 shows an example session of the terminal program. Note that the terminal is only a minimal implementation for demonstration purposes. It does not handle any control sequences such as cursor control, except from carriage return characters (`\r`). Feel free to extend the sample as you like for your purposes. For example, for applications relying on binary data transfer, it might be better to use a hex format for sending and receiving data. Listing 6.2 contains the MIDP version of the terminal program, and Listing 6.3 shows the PDAP version.

Figure 6.4. An example PdapTerminal connection to an HTTP server using the socket protocol after sending "GET / HTTP 1.0" and an empty `MidpTerminal` and line.

Listing 6.2 MidpTerminal.java—The MIDP Terminal Application for Using Different Protocols in the Same Application

```
import java.io.*;

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


/** The MIDP version of a simple Terminal client */

public class MidpTerminal extends MIDlet implements CommandListener {

    /** The Handler class cares about establishing the connection and
        receiving and displaying data in the background. */

    class Handler extends Thread {

        StreamConnection connection;
        InputStream in;
        OutputStream out;
        boolean leave;


        /** Establishes a connection to the given URI */
        public Handler (String uri) throws IOException {

            connection = (StreamConnection) Connector.open
                (uri, Connector.READ_WRITE, true);

            out = connection.openOutputStream();
            in = connection.openInputStream();

            show ("opened: "+uri + "\r");
        }

        /* Like in.read(), but additional performs telnet parameter
           negotiations */

        public int read() throws IOException {
            while (true) {
                int i = in.read();
                if (i != 0x0ff) return i;
```

```
            int cmd = in.read();

            if (cmd == 0x0ff)
                return 0x0ff;

            int opt = in.read();

            if (cmd == 0xfd || cmd == 0x0fb) {
                out.write (0x0ff);
                out.write (cmd == 0xfd ? 252 : 254);
                out.write (opt);
                out.flush();
            }
        }
    }

    /** Main receive loop running in the background */

    public void run() {

        StringBuffer buf = new StringBuffer();

        try {
            while (!leave) {
                do {
                    int i = in.read();

                    if (i == -1) disconnect();
                    else if (i == '\r' || i >= ' ')
                        buf.append ((char) i);
                }
                while (!leave && in.available() > 0
                    && buf.length() < 1024);
                show (buf.toString());
                buf.setLength (0);
            }
        }
        catch (Exception e) {
            if (!leave) show (e.toString() + "\r");
            disconnect();
        }
    }
}

List incoming = new List ("MidpTerminal", Choice.IMPLICIT);

TextBox uriField = new TextBox
    ("Connect to:", "http://www.kawt.de/", 100, TextField.ANY);

TextBox sendField = new TextBox ("Send:", "", 100, TextField.ANY);

Command connectCmd = new Command ("Connect", Command.SCREEN, 1);
Command sendCmd = new Command ("Send", Command.SCREEN, 1);
Command okCmd = new Command ("Ok", Command.OK, 1);
Command abortCmd = new Command ("Abort", Command.CANCEL, 1);

Handler handler = null;
Display display;
```

```
/** Set up user interface */

public MidpTerminal() {
    uriField.addCommand (okCmd);
    uriField.addCommand (abortCmd);
    uriField.setCommandListener(this);
    sendField.addCommand (okCmd);
    sendField.addCommand (abortCmd);
    sendField.setCommandListener(this);

    incoming.addCommand (connectCmd);
    incoming.addCommand (sendCmd);
    incoming.setCommandListener(this);

    incoming.append ("", null);
}

/** Set display to the URI dialog */

public void startApp() {
    display = Display.getDisplay (this);
    display.setCurrent (uriField);
}


/** Shows the given string by appending it to the
    list with respect to contained line breaks. */


public void show (String data) {
    int i0 = data.indexOf ('\r');

    if (i0 == -1) i0 = data.length();

    incoming.set
        (incoming.size()-1,
         incoming.getString (incoming.size() - 1)
         + data.substring (0, i0), null);
    i0++;
    while (i0 <= data.length()) {
        int i = data.indexOf ((char) 13, i0);
        if (i == -1) i = data.length();
        incoming.append (data.substring (i0, i), null);
        i0 = i+1;
    }
}
/** Performs the action associated with the given command
    like showing dialogs, opening the connection or sending
    a string */

public void commandAction (Command c, Displayable d) {

    try {
        if (c == connectCmd)
            display.setCurrent (uriField);
        else if (c == sendCmd && handler != null)
            display.setCurrent (sendField);
        else if (c == abortCmd)
            display.setCurrent (incoming);
        else if (c == okCmd) {
            display.setCurrent (incoming);
```

```
                if (d == sendField && handler != null) {
                    handler.out.write
(sendField.getString().getBytes());
                    handler.out.write ('\r');
                    handler.out.write ('\n');
                    handler.out.flush();
                    sendField.setString ("");
                }
                else if (d == uriField) {

                    disconnect();
                    handler = new Handler (uriField.getString());
                    handler.start();
                }
            }
        }
        catch (Exception e) {
            show (e.toString());
            disconnect();
        }
    }


    public void pauseApp() {
    }
    public void disconnect() {
        if (handler != null) {
            handler.leave = true;
            show ("disconnected!\r");
            try {
                handler.connection.close();
                handler.in.close();
                handler.out.close();
            }
            catch (IOException e) {
            }
            handler = null;
        }
    }


    public void destroyApp (boolean unconditional) {
        disconnect();
    }
}
```

Listing 6.3 PdapTerminal.java—The PDAP Terminal Application for Using Different Protocols in the Same Application

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

import java.awt.*;
import java.awt.event.*;

public class PdapTerminal extends MIDlet implements ActionListener {


    /** The Handler class cares about establishing the connection and
```

```
    receiving and displaying data in the background. */

class Handler extends Thread {

    StreamConnection connection;
    InputStream in;
    OutputStream out;
    boolean leave;
    /** Establishes a connection to the given URI */

    public Handler (String uri) throws IOException {

        connection = (StreamConnection) Connector.open
            (uri, Connector.READ_WRITE, true);

        out = connection.openOutputStream();
        in = connection.openInputStream();

        show ("opened: "+uri + "\r");
    }


    /**  Like in.read(), but additional performs telnet
        parameter negotiations. */

    public int read() throws IOException {
        while (true) {
            int i = in.read();
            if (i != 0x0ff) return i;

            int cmd = in.read();

            if (cmd == 0x0ff)
                return 0x0ff;

            int opt = in.read();

            if (cmd == 0xfd || cmd == 0x0fb) {
                out.write (0x0ff);
                out.write (cmd == 0xfd ? 252 : 254);
                out.write (opt);
                out.flush();
            }
        }
    }


    /** Collects incoming data in the background and
        shows it if the buffer size reaches 1 k or
        no more data is available at the moment. */

    public void run() {

        StringBuffer buf = new StringBuffer();
        try {
            while (!leave) {

                do {
                    int i = in.read();

                    if (i == -1) disconnect();
```

```
                    else if (i == '\r' || i >= ' ')
                        buf.append ((char) i);
                }
                while (!leave && in.available() > 0
                    && buf.length() < 1024);

                show (buf.toString());

                buf.setLength (0);
            }
        }
        catch (Exception e) {
            if (!leave) show (e.toString() + "\r");
            disconnect();
        }
    }
}


/** Class for thread safe appending of information to
    the list of incoming data */

class Appender implements Runnable {
    String data;
    Appender (String data) {
        this.data = data;
    }

    public void run() {
        int i0 = data.indexOf ('\r');

        //System.out.println ("Adder: cr index is: "+i0);
        if (i0 == -1) i0 = data.length();

        incoming.replaceItem
            (incoming.getItem (incoming.getItemCount() - 1)
             + data.substring (0, i0), incoming.getItemCount()-1);
        i0++;
        while (i0 <= data.length()) {
            int i = data.indexOf ('\r', i0);
            if (i == -1) i = data.length();
            incoming.add(data.substring (i0, i));
            i0 = i+1;
        }
    }
}


Frame frame = new Frame();
TextField urlField = new TextField ("");
List incoming = new List();
TextField sendField = new TextField();

Button connectButton = new Button ("connect");
Button sendButton = new Button ("send");


Handler handler;


/** Initializes GUI */
```

```
    public PdapTerminal() {
        frame = new Frame ("GcfTerminal");

        frame.addWindowListener(new WindowAdapter() {
                public void windowClosing (WindowEvent e) {
                    destroyApp (true);
                    notifyDestroyed();
                }
            });

        connectButton.addActionListener(this);

        Panel topPanel = new Panel (new BorderLayout());
        //topPanel.add("West", protocolChoice);
        topPanel.add("Center", urlField);
        topPanel.add("East", connectButton);

        sendButton.addActionListener(this);

        Panel bottomPanel = new Panel (new BorderLayout());
        bottomPanel.add("Center", sendField);
        bottomPanel.add("East", sendButton);

        frame.add("North", topPanel);
        frame.add("Center", incoming);
        frame.add("South", bottomPanel);

        frame.pack();
    }


    /** Shows the given string thread safe by handing a new Appender
        to invokeLater */

    public void show (String s) {
        try {
            Toolkit.getDefaultToolkit().getSystemEventQueue()
                .invokeAndWait (new Appender (s));
        }
        catch (Exception e) {
            throw new RuntimeException (e.toString());
        }
    }


    /** Shows the main frame on the device screen */

    public void startApp() {
        frame.show();
    }


    /** Handles the buttons by opening a connection or sending text
*/

    public void actionPerformed (ActionEvent event) {

        try {
            if (event.getSource() == sendButton && handler != null) {
                handler.out.write (sendField.getText().getBytes());
```

```
                handler.out.write ('\r');
                handler.out.write ('\n');
                handler.out.flush();
                sendField.setText ("");
            }
            else if (event.getSource() == connectButton) {
                disconnect();
                handler = new Handler (urlField.getText());
                handler.start();
            }
        }
        catch (Exception e) {
            incoming.add(e.toString());
            incoming.add("");
            disconnect();
        }
    }


    /** Closes the connection if any */

    public void disconnect() {
        if (handler != null) {
            handler.leave = true;
            show ("disconnected!\r");
            try {
                handler.connection.close();
                handler.in.close();
                handler.out.close();
            }
            catch (IOException e) {
            }
            handler = null;
        }
    }


    public void pauseApp() {
    }
    public void destroyApp (boolean unconditional) {
        disconnect();
        frame.setVisible (false);
    }
```

A Simple HTTP-Based Client-Server Chat Application

In addition to the telnet client, we would like to show you how to build a "real" client-server application, where the server runs on a desktop computer, and the CLDC device takes over the role of the client. To keep things simple, we have chosen a chat application as an example. The idea is that you can connect to a server, see the messages from other people connected to the same server, and write your own messages that become visible to the other users. Because HTTP is the only protocol available for all devices, we will use HTTP as the communication protocol for our application. However, using HTTP includes a significant drawback: HTTP does not provide server initiated transmissions, so the clients need to connect to the server and to "poll" for new data from time to time. It might be possible to work around this limitation by keeping an HTTP connection open for each client and to forward data to all connected clients automatically. However, it might be possible that the gateway used by the device for HTTP access does not support this, so we will stick to polling here.

In order to allow the clients to receive only their new messages, all messages have an unique number, which is managed by a simple server-sided counter. Thus, the client can submit the number of the newest message it has already received, and the server will send only newer messages that have higher numbers assigned. If no number is given, the server will just send the 10 most recent messages.

Thus, we can define the following behaviors for reading:

- Client—Sends a request of type `GET`.
- Server—First it sends a line containing the number that will be assigned to the next message. Then, if the URL is of the form `/?start=N`, a list of all messages, starting with message number *N*, is transmitted. For all other URLs, the last 10 messages are submitted. All items are separated by a pair of carriage return and linefeed control characters (`\r\n`).

For submitting text, you will use the HTTP `POST` command with an identical URL, but you will also send the nickname and the text in the body of the request. In return, the server sends the same content as for the `GET` command:

- Client—Sends a request of type `POST`.
- Server—Sends the same reply as for the `GET` command. The text just sent by the client is included in the list of messages. Thus, at least one message is sent.

J2SE Chat Server

Now that we have defined the communication protocol, we can start with implementing a corresponding server.

The server depends on the `java.io` and `java.net` packages. It stores the number of the current message, a buffer for text, and a J2SE server socket in member variables:

```
import java.io.*;
import java.net.*;

public class ChatServer {
int current = 0;
String [] lines = new String [256];
ServerSocket serverSocket;
```

The constructor of the server gets a port number as input and creates a corresponding server socket:

```
public ChatServer (int port) throws IOException {
    serverSocket = new ServerSocket (port);
    System.out.println ("Serving port: "+port);
}
```

The `run()` method of the server contains a loop that waits for incoming connections. When a request is accepted, a buffered reader and a writer corresponding to input and output streams associated with the connection are handed over to the `handleRequest()` method. Usually, the actual handling of the request would be performed in a separate thread, enabling the server to handle new requests immediately. However, in order to keep the server as simple as possible, you handle the request in the current thread, blocking new requests for the corresponding amount of time:

```
public void run() {
    while (true) {
        try {
            Socket socket = serverSocket.accept();
            handleRequest (new BufferedReader
                (new InputStreamReader (socket.getInputStream())),
                           new OutputStreamWriter
(socket.getOutputStream()));
```

```
            socket.close();
        }
        catch (Exception e) {
            e.printStackTrace (System.err);
        }
    }
}
```

The main functionality of the chat server is performed in the `handleRequest()` method. It gets the socket reader and writer as input from the `run()` method. At first, it reads the HTTP request line from the client and prints it to system.out:

```
public void handleRequest (BufferedReader reader,
                           Writer writer) throws IOException {

    String request = reader.readLine();
    System.out.println ("handling: "+request);
```

The next step is to analyze the request line. For that purpose, it is divided into the method, the requested address, and the version part, which are separated by space characters:

```
int s0 = request.indexOf (' ');
int s1 = request.indexOf (' ', s0+1);

String method = request.substring (0, s0);
String url = request.substring (s0+1, s1);
```

Now, the first line to be submitted is determined by analyzing the request URL:

```
int start = -1;  // default;
int cut = url.indexOf ("?start=");

if (cut != -1)
    start = Integer.parseInt (url.substring (cut+7));
if (start < 0) start = count – 10;
```

Additional header lines are skipped by reading from the stream until an empty line or the end of the stream is reached. An empty line marks the end of the HTTP headers and the beginning of the content of the request:

```
while (true) {
    String s = reader.readLine();
    System.out.println ("header: "+s);
    if (s == null || s.length() == 0) break;
}
```

Now, if the HTTP-Request method is `POST`, the nickname and the sender are read from the HTTP content. A corresponding string is appended to the message ring buffer of the server:

```
if (method.equalsIgnoreCase ("post")) {

    String nick = reader.readLine().substring (5);
    String text = reader.readLine().substring (5);

    System.out.println ("nick="+nick);
    System.out.println ("text="+text);

    lines [(current++) % lines.length] = nick + ": "+ text;
```

```
    // skip possible additional crlf from bad http implementations
    if (reader.ready()) reader.readLine();
}
```

Finally, an HTTP OK status report is sent back to the client, together with the number that will be assigned to the next incoming messages, and the list of messages that was requested by the client:

```
    writer.write ("HTTP/1.0 200 OK\r\n");
    writer.write ("Content-Type: text/plain\r\n");

    writer.write ("Connection: close\r\n");

    // Header is separated from content by a blank line.
    writer.write ("\r\n");

    writer.write (""+current+"\r\n");

    if (start < current - lines.length) start = current - 
lines.length;
    if (start < 0) start = 0;

    for (int i = start; i < current; i++) {
        writer.write (lines [i % lines.length]);
        writer.write ("\r\n");
    }
            writer.close();
}
```

The main method of the chat server sets up the server listening to the port given as the command-line parameter. If no port number was given, it defaults to port `8080`:

```
    public static void main (String [] argv) throws IOException {

        if (argv.length == 0)
            new ChatServer (8080).run();
        else if (argv.length == 1)
            new ChatServer (Integer.parseInt (argv[0])).run();
        else
            System.out.println ("Usage: java ChatServer [port]");
    }
}
```

Note

The Java Servlet API provides better support for implementing HTTP-based server applications than using raw sockets. However, we did not want to introduce an additional dependency for this example application.

MIDP and PDAP Chat Clients

Now, you have developed a simple HTTP based chat server. You can test it using a simple Web browser. By starting the server on the local machine and pointing a Web browser to the address `http://localhost:8080`, you can get a list of the 10 most recent messages. This list will probably be empty, but you can use a very simple HTML page to send messages to the server:

```
<html><head><title>Chat Server Test</title></head>
<body>
 <form target="main" method="post"
       action="http://localhost:8080/"
       enctype="text/plain">

  <table>
    <tr><td>Nickname:</td><td><input name="nick" /></td></tr>
    <tr><td>Text:</td><td><input size="80" name="text" /></td></tr>
   </table>
   <input value="Submit Text" type="submit" />
  </form>
 </body>
</html>
```

However, the main idea is to use J2ME devices as clients for the chat server. Listings 6.4 and 6.5 contain the MIDP and PDAP versions of the chat client. Again, the main task is performed in the transmit method in both cases, which sends a new string to the server and updates the display of the client with the messages received from the server.

The transfer method is called from two points in the program:

- From the event handler when the user requests to send some text. In that case, the string to be submitted to the server is given as a parameter.
- Periodically from a refresh task every four seconds with `null` as a parameter, in order to keep the message display of the client updated.

The transfer method takes the string to be sent to the server as parameter, or `null` if the local list of messages should only be updated from the server without sending a new message. Depending on this parameter, a HTTP connection is opened in `READ` or `READ_WRITE` mode. The URI is constructed from the hostname that was queried by the user interface and stored in the host variable and the `start` parameter, denoting from which message number the server should start sending. The `count` variable is initialized with the value `-1`, so for the first request, the server will send back the 10 most recent messages. The `count` variable is updated later in this method from the response of the server. Because IO exceptions might be thrown during the connection, you include the whole method in a `try`-`catch` block. The Boolean return value indicates whether the transfer was performed successfully:

```
boolean transfer (String submit) { // if null, just read
    try {
        HttpConnection connection = (HttpConnection) Connector.open
            (host + "/?start="+count,
             submit != null ? Connector.READ_WRITE : Connector.READ);
```

If `submit` is not `null`, a corresponding writer is obtained from the HTTP connection, and the method is set to `POST`. Then, the message stored in the `submit` variable is submitted as content of the request:

```
Writer writer = null;

if (submit != null) {
    connection.setRequestMethod (HttpConnection.POST);
    writer = new OutputStreamWriter
        (((StreamConnection) connection).openOutputStream());
    writer.write ("nick="+nick + "\r\n");
    writer.write ("text="+submit+"\r\n");
        writer.close();
}
```

Now you open a reader in order to read the new messages submitted with the reply from the server:

```
Reader reader = new InputStreamReader
    (((StreamConnection) connection).openInputStream());
```

First, you read the new `count` value, denoting the number that will be assigned to the next message arriving at the server. This value is important in order to know where to start the next request. `readLine()` is a static method of this class that reads a line from the given reader:

```
count = Integer.parseInt (readLine (reader));
```

Now you read the messages submitted by the server until you reach the end of the stream:

```
while (true) {
    String s = readLine (reader);
    if (s == null || s.length() == 0) break;
    addLine (s);
}
```

Next, you close the readers and the corresponding connection. Also, you return `true` in order to indicate that the transfer was performed successfully:

```
                    reader.close();
    connection.close();
    return true;
}
```

Finally, if an exception occurred in the connection, you add the corresponding error string to the display, and then call the `disconnect()` method which stops the timer that ensures the display is updated periodically. A `false` value is returned in order to indicate that an exception has occurred:

```
    catch (Exception e) {
        addLine (e.toString());
        disconnect();
        return false;
    }
}
```

The only other part of the application relevant for communications is `RefreshTask`:

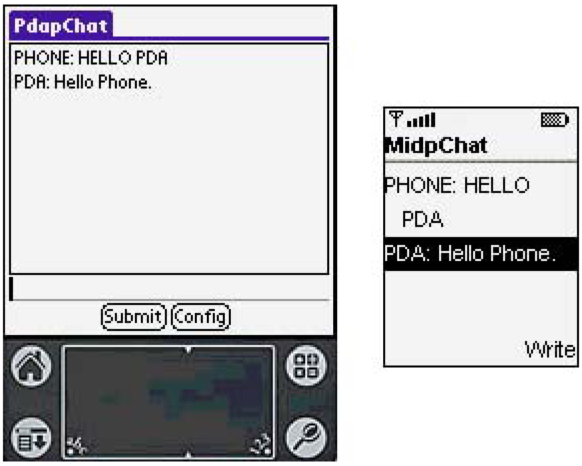
```
class RefreshTask extends TimerTask {
    public void run() {
      transfer (null);
    }
}
```

When a connection is established, it is launched with parameters to request an update of the messages from the server every four seconds:

```
timer = new Timer();
timer.schedule (new RefreshTask(), 0, 4000);
```

Figure 6.5 shows emulated MIDP and PDAP clients connected to a local chat server. Note that the resulting chat application is minimalistic. For example, it does not check if two different users are using the same nickname. The application is intended to show the basic HTTP functionality only. Feel free to extend or change the application as you like for your own purposes.

Figure 6.5. Emulated MIDP and PDAP clients connected to a local chat server.



Listing 6.4 MidpChat.java—A MIDP Chat Client Using the HTTP Protocol to Communicate with the Server

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

import java.util.*;
import java.io.*;

public class MidpChat extends MIDlet implements CommandListener {

    List list = new List ("MidpChat", Choice.IMPLICIT);
    TextBox text = new TextBox ("Chat Text", "", 100, TextField.ANY);

    int count = -1;
    Timer timer;

    String host = "http://localhost:8080";
    String nick = "guest";

    Display display;

    Command write = new Command ("Write", Command.OK, 1);
    Command submit = new Command ("Submit", Command.OK, 1);
    Command cancel = new Command ("Cancel", Command.BACK, 1);

    class RefreshTask extends TimerTask {

        public void run() {
            if (display.getCurrent() == list)
                transfer (null);
        }
    }

    class ConfigForm extends Form implements CommandListener {

        TextField hostField =
            new TextField ("Host:", host, 50, TextField.ANY);
        TextField nickField =
            new TextField ("Nickname:", nick, 50, TextField.ANY);
        Command connectCommand = new Command ("Connect", Command.OK,
1);
        Command abortCommand = new Command ("Abort", Command.BACK, 1);
```

```
    ConfigForm() {
        super ("Configuration");
        append (hostField);
        append (nickField);
        addCommand (connectCommand);
        addCommand (abortCommand);
        setCommandListener(this);
    }

    public void commandAction(Command c, Displayable d) {
        if (c == connectCommand) {
            host = hostField.getString();
            nick = nickField.getString();
            connect();
        }
        display.setCurrent (list);
    }
}

static String readLine (Reader reader) throws IOException {
    StringBuffer buf = new StringBuffer();
    while (true) {
        int c = reader.read();
        if (c == -1) {
            if (buf.length() == 0) return null;
            break;
        }
        if (c == 10) break;
        if (c != 13) buf.append ((char) c);
    }
    return buf.toString();
}

public MidpChat() {
    list.addCommand (write);
    list.setCommandListener(this);
    text.addCommand (submit);
    text.addCommand (cancel);
    text.setCommandListener(this);
}

public void startApp() {
    display = Display.getDisplay (this);
    if (timer == null)
        configure();
    else
        display.setCurrent (list);
}
void configure() {
    disconnect();
    display.setCurrent (new ConfigForm());
}

void connect() {
    disconnect();
    if (transfer (null)) {
        timer = new Timer();
        timer.schedule (new RefreshTask(), 0, 4000);
    }
}
```

```
    void addLine (String line) {
        list.append (line, null);
    }

    void disconnect() {
        if (timer != null) {
            timer.cancel();
            timer = null;
        }
    }

    boolean transfer (String submit) { // if null, just read
        try {
            HttpConnection connection = (HttpConnection)
Connector.open
                (host + "/?start="+count,
                 submit != null ? Connector.READ_WRITE :
Connector.READ);

            Writer writer = null;

            if (submit != null) {
                connection.setRequestMethod (HttpConnection.POST);
                writer = new OutputStreamWriter
                    (((StreamConnection)
connection).openOutputStream());
                writer.write ("nick="+nick + "\r\n");
                writer.write ("text="+submit+"\r\n");
                writer.close();
            }

            Reader reader = new InputStreamReader
                (((StreamConnection) connection).openInputStream());
            count = Integer.parseInt (readLine (reader));
            while (true) {
                String s = readLine (reader);
                if (s == null || s.length() == 0) break;
                addLine (s);
            }

            reader.close();
            connection.close();
            return true;
        }
        catch (Exception e) {
            addLine (e.toString());
            disconnect();
            return false;
        }
    }

    public void commandAction(Command c, Displayable d) {
        if (c == write) {
            display.setCurrent (text);
        }
        else {
            if (c == submit) {
                transfer (text.getString());
                text.setString ("");
            }
```

```
            display.setCurrent (list);
        }
    }

    public void pauseApp() {
    }

    public void destroyApp (boolean unconditional) {
        disconnect();
    }
}
```

Listing 6.5 PdapChat.java—A PDAP Chat Client Using the HTTP Protocol to Communicate with the Server

```
import java.io.*;
import java.awt.*;
import java.util.*;
import java.awt.event.*;
import javax.microedition.io.*;

import javax.microedition.midlet.*;


public class PdapChat extends MIDlet implements ActionListener,
Runnable {

    Frame frame = new Frame ("PdapChat");
    java.awt.List list = new java.awt.List();
    TextField text = new TextField();
    Button configButton = new Button ("Config");
    Button submitButton = new Button ("Submit");

    int count = -1;
    Timer timer;
    String host = "http://localhost:8080";
    String nick = "guest";

    class ConfigDialog extends Dialog implements ActionListener {
        TextField hostField = new TextField (host);
        TextField nickField = new TextField (nick);
        Button connectButton = new Button ("Connect");
        Button abortButton = new Button ("Abort");

        ConfigDialog() {
            super (frame, "Configuration", true);
            Panel labels = new Panel (new GridLayout (0, 1));
            Panel fields = new Panel (new GridLayout (0, 1));
            Panel buttons = new Panel();

            add("West", labels);
            add("Center", fields);
            add("South", buttons);


            labels.add(new Label ("host:"));
            labels.add(new Label ("nick:"));
            fields.add(hostField);
            fields.add(nickField);

            connectButton.addActionListener(this);
```

```
        buttons.add(connectButton);
        abortButton.addActionListener(this);
        buttons.add(abortButton);
        pack();
    }
    public void actionPerformed (ActionEvent ev) {
        if (ev.getSource() == connectButton) {
            host = hostField.getText();
            nick = nickField.getText();
            connect();
        }
        setVisible (false);
    }
}

class RefreshTask extends TimerTask {
    public void run() {
        try {
            Toolkit.getDefaultToolkit()
                .getSystemEventQueue()
                .invokeAndWait (new Runnable() {
                        public void run() {
                            transfer (null);
                        }
                    }
            );
        }
        catch (Exception e) {
        }
    }
}

static String readLine (Reader reader) throws IOException {
    StringBuffer buf = new StringBuffer();
    while (true) {
        int c = reader.read();
        if (c == -1) {
            if (buf.length() == 0) return null;
            break;
        }
        if (c == '\n') break;
        if (c != '\r') buf.append ((char) c);
    }
    return buf.toString();
}

public PdapChat() {
    frame.add("Center", list);
    Panel input = new Panel (new BorderLayout());
    frame.add("South", input);
    input.add("Center", text);
    Panel buttons = new Panel();
    input.add("South", buttons);
    buttons.add(submitButton);
    buttons.add(configButton);
    submitButton.addActionListener(this);
    configButton.addActionListener(this);

    frame.addWindowListener(new WindowAdapter() {
            public void windowClosing (WindowEvent e) {
                destroyApp (true);
```

```
                notifyDestroyed();
            }
        });
    frame.pack();
}

void configure() {
    disconnect();
    new ConfigDialog().show();
}

void connect() {
    disconnect();
    if (transfer (null)) {
        timer = new Timer();
        timer.schedule (new RefreshTask(), 0, 4000);
    }
}

void disconnect() {
    if (timer != null) {
        timer.cancel();
        timer = null;
    }
}

/** implementation of runnable */

public void run() {
    transfer (null);
}
boolean transfer (String submit) { // if null, just read
    try {
        HttpConnection connection = (HttpConnection)
              Connector.open(host + "/?start="+count, submit !=
              null? Connector.READ_WRITE : Connector.READ);
        Writer writer = null;
        if (submit != null) {
            connection.setRequestMethod (HttpConnection.POST);
            writer = new OutputStreamWriter
              (((StreamConnection)connection).openOutputStream());
            writer.write ("nick="+nick + "\r\n");
            writer.write ("text="+submit+"\r\n");
                                        writer.close();
        }

        Reader reader = new InputStreamReader
            (((StreamConnection) connection).openInputStream());
        count = Integer.parseInt (readLine (reader));
        while (true) {
            String s = readLine (reader);
            if (s == null || s.length() == 0) break;
            addLine (s);
        }
                                  reader.close();
        connection.close();
        return true;
    }
    catch (Exception e) {
        addLine (e.toString());
        disconnect();
```

```
            return false;
        }
    }

    public void addLine (String l) {
        list.add(l);
    }

    public void actionPerformed (ActionEvent event) {
        if (event.getSource() == configButton)

            configure();
        else {
            transfer (text.getText());
            text.setText ("");
        }
    }

    public void startApp() {
        frame.show();
        if (timer == null) configure();
    }

    public void destroyApp (boolean unconditional) {
        frame.setVisible (false);
        disconnect();
    }

    public void pauseApp() {
    }
}
```

MIDP 2.0 Additions to the javax.microedition.io Package

The Generic Connection Framework is extended by MIDP 2.0 with the following interfaces and classes:

- the `HTTPSConnection` interface supporting secure HTTP connections.
- the `PushListener` interface that needs to be used together with the new `PushRegistry` class.
- the `SecureConnection` in order to establish SSL or TLS connections.
- the `SecurityInfo` interface.
- the `ServerSocketConnection` interface, which has already been described in this chapter, is mandatory for MIDP 2.0.
- the `UDPDatagramConnection` interface that is derived from `DatagramConnection`.

`The UDPDatagramConnection` interface is derived from `DatagramConnection` and adds two new methods to retrieve information about the local machine such as `String getLocalAddress()` and `int getLocalPort()`.

Handling Inbound Connections

An application that needs to respond to inbound connections can be registered with the Application Management Systems (AMS) in two ways. Although it is possible to call the `registerConnection()` method of the `PushRegistry` dynamically, listeners for inbound connections can also be statically registered using a new entry in the JAD file:

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

The first entry `MIDlet-Push-<n>`is used for numbering the push MIDlets in a given suite. The `ConnectionUrl` defines the protocol and parameters of inbound connections the AMS should listen for. The `MIDletClassName` parameter contains the class name of the MIDlet that should be started if an inbound connection on the given URL was detected. The `AllowedSender` parameter may be used to restrict connections to the MIDlet. In case of IP connections the allowed senders may be specified by an IP number including wildcards. A single * allows access from any client.

If the client is allowed to connect to the MIDlet, the AMS starts the MIDlet by calling the `startApp()` method. After the MIDlet is started, it needs to handle the connection itself. For instance, in order to register a MIDlet called `MIDPHttpServer`, the corresponding jad entry allowing all clients to connect would look as follows:

```
MIDlet-Push-1: socket://:80, MIDPHttpServer, *
```

Another approach to register a MIDlet to inbound connection is to call the `registerConnection()` method from the `PushRegistry` class dynamically. The previously mentioned descriptor file entries are passed as `String` parameters to the method. In both cases the `startApp()` method is called if an inbound connection is detected by the AMS.

Since the `startApp()` method has no parameter for passing the inbound connection, the MIDlet needs to query the inbound connection. For querying the inbound connections currently waiting to be handled, the `PushRegistry.listConnctions()` method is provided. This method takes one boolean parameter as flag. `True` indicates that the `String` array returned by the method should contain connections with input data that requires handling only. If `false` is passed to the method, all connections registered to the given `MIDletSuite` are returned.

The following sample code snippet shows how to handle a previously registered inbound socket connection:

```
public void startApp() {

    String availableConnections[];
    availableConnections = PushRegistry.listConnections(true);

    if (availableConnections.size() > 0) {

    }

    try {
        // since the socket connection is the only connection
        // we want to listen to we can simply pass the result String
        // containing the Connector.open() method.
        StreamConnection sconn = (StreamConnection)Connector,open();
        InputStream is = sconn.opeInputStream();
        // read input data here
        is.close();
        sconn.close();
    }
    catch (IOEXception ioe) {
        // handle a possible error during establishing the connection
    }

}
```

Note

Please note, that not all generic connections will be appropriate for use as push application transport. Even if a GCF protocol is supported on a particular device it is not required to be enabled as a valid push mechanism.

The `PushRegistry` supports a listener model as well, which can be used by implementing the `PushListenerInterface` in a subclass of `MIDlet`. In order to register a class implementing the `PushListener`, the `PushRegistry.setPushListener()` method needs to be called, taking the connection as first parameter, and the class implementing the `PushListener` interface as second parameter. When the registered connection is established, the `notifyConnection()` method of the `PushListener` interface is called. The parameter passed to `notifyConnection()` is a String describing the incoming connection parameters.

Security

The new `SecureConnection` and `HttpsConnection` interfaces support the `getSecurityInfo()` method returning a class implementing the `SecurityInfo` interface.

The `SecurityInfo` interface provides the following methods in order to get information about a secure connection:

- `String getChipherSuite()`
- `String getProtocolName()`
- `String getProtocolVersion()`
- `Certificate getServerCertificate()`

Summary

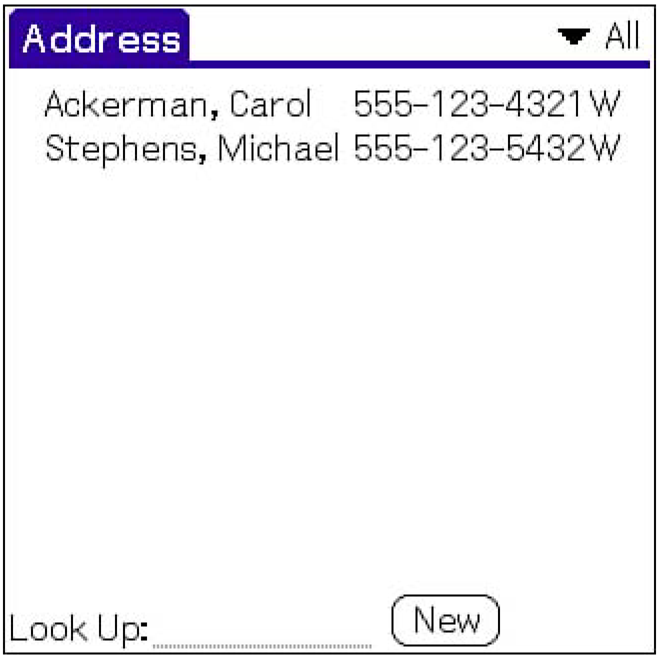
In this chapter you have learned how to integrate network capabilities into your MIDlets. You know the possible network protocols that might be supported in the CLDC profiles, especially the HTTP protocol. You have also seen how to integrate these protocols using the generic connection framework into applications such as the terminal and the chat client-server application.

Chapter 7. PIM: Accessing the Personal Information Manager

IN THIS CHAPTER

The traditional main purpose of PDAs is to serve as Personal Digital Assistants, providing access to a digital address book and calendar, as shown in Figure 7.1. The Personal Digital Assistant Profile (PDAP) contains a corresponding API, allowing you to access the device-specific address book, calendar database, and to-do lists from Java applications. This chapter discusses the Personal Information Manager (PIM) API that is available in PDAP.

Figure 7.1. The SONY CLIE Emulator showing the built-in address book containing two sample contacts.



For J2SE and PersonalJava the so called JavaPhone API is available (see http://java.sun.com/products/javaphone/). Because the PDAP implementation intends to have a very small footprint, which is necessary to fit in the small RAM of CLDC devices, adopting the JavaPhone API completely or just providing a JavaPhone subset would be inappropriate. In order to accomplish the request to minimize the PIM footprint, PDAP defines an optional PIM API in the package `javax.microedition.pim`.

In the following section, we will begin with the description of the general structure of the PIM API, and then go into the details of accessing the address book, calendar, and to-do lists. Finally, we'll show you how to create a demonstration MIDlet that is capable of accessing the address book of the device.

General PIM API Design

The basic idea of the API is to store entries such as contacts, events, and to-do elements in an appropriate database. In order to achieve this, the PIM API contains an interface `PIMList` providing methods to add, delete, and enumerate all the entries contained in the list. The concrete implementations of `PIMList` contained in the PIM API are `ContactList`s to store contacts, `EventList`s to store events, and `ToDoList`s to store to-do items. The PIM class provides static methods such as openContactList() to access the different list types.

`PIMList`s contain objects implementing the `PIMElement` interface. `PIMElement` encapsulates the common properties of the different PIM elements such as `Contact`s, `Event`s, and `ToDo`s. The interface provides methods for common functionality such as category access and vFormat import and export, as well as access to the fields of an entry. Access to some specialized fields is provided by the derived interfaces `Contact`, `Event`, and `ToDo`. For illegal PIM operations, a `PIMException` will be thrown. In the next sections, you will learn the details of the specialized address book, event, and to-do functionality.

In contrast to vCard, vCalendar, and the JavaPhone API, the PDAP PIM API does not support multiple fields with the same name and type combination. Although this restriction causes a slight loss of flexibility, it makes the API much less complex than the JavaPhone API. Fields with a single ID and multiple types are described in more detail in the following section.

Addressbook API

The functionality of the address book is achieved using the `ContactList` and `Contact` classes only. Contacts in the PIM API support a subset of the fields of the vCard format version 3.0 specified in IETF RFC 2426. The following example shows a vCard for John Smith in ASCII format:

```
BEGIN:VCARD
FN:John Smith
N:Smith;John;;;MD
TEL;TYPE=WORK:555-7352
TEL;TYPE=HOME:555-4321
END:VCARD
```

Note

More information about IETF RFC 2426 can be found at

http://www.nic.mil/ftp/rfc/rfc2426.txt

Contacts and Their Fields

As mentioned earlier, the interface Contact extending the `PIMElement` interface represents a single contact. A *contact* consists of several fields such as name, phone number, or birthday. These fields are of different data types, depending on their purpose. For example, the name and address information is stored in strings, whereas the birthday is stored in a date field. In the following subsections we will describe the different field types and fields available in the PIM API.

The individual fields of a contact are addressed by field IDs. The *field IDs* are integer values which are defined in the Contact interface. Examples for fields IDs are `Contact.FORMATTED_NAME` or `Contact.PUBLIC_KEY`.

String-Based Data

String data is added to a `Contact` using the `setString()` method, which takes two parameters. The first parameter is the field ID, and the second takes the string to be stored. For reading string data fields, the method `getString()` is provided. This method takes the field ID to be read as a parameter and returns the corresponding value. If the field has not been set, `null` is returned.

A contact containing one field, the formatted name, is created as follows:

```
ContactList myContacts = PIM.openContactList (PIM.READ_WRITE);
Contact myContact = myContacts.createContact();
myContact.setString(Contact.FORMATTED_NAME, "John, Smith");
```

Please note that a try-catch block for the `PIMException` is missing in the code above. We will omit the mandatory try-catch block from all following code snippets.

A special field is the UID field. It contains an identifier that is unique for each contact. The unique identifier is generated automatically and cannot be modified. It can be used to quickly retrieve a particular contact.

Table 7.1 shows all field IDs that can be used to add string data to a `Contact`.

Table 7.1. Field ID Constants for String-based Data Defined in the Contact Interface

Field Name	***Description***
`ORG`	The organization name
`FORMATTED_NAME`	The formatted name
`NAME_FAMILY`	The family name
`NAME_GIVEN`	The given name
`NAME_OTHER`	Undefined information
`NAME_PREFIX`	The name prefix
`NAME_SUFFIX`	The name suffix
`NICKNAME`	A nickname
`NOTE`	A note
`PUBLIC_KEY`	The public key, such as a PGP public key
`TITLE`	The title
`UID`	The contact's UID
`URL`	A URL associated with the contact

Date-Based Data

In addition to text fields, the PIM API supports fields for storing dates, such as a person's birthday. Similar to the `setString()` and `getString()` methods, the `Contact` class supports the `setDate()` and `getDate()` methods to store and retrieve dates in a `Contact`. The set method takes a field ID such as `BIRTHDAY` and a given `Date` object as parameters. The date can be retrieved by giving the corresponding field ID to the `getDate()` method.

Another field containing a date is `REVISION`. This entry stores the date when the contact was last modified. Setting this value is not recommended because it is automatically done by the implementation. `BIRTHDAY` and `REVISION` are the only date fields supported by PDAP.

The birthday of a contact can be set as shown in the following line of code:

```
myContact.setDate(Contact.BIRTHDAY, new Date());
```

Binary Data

Another simple field type is the byte array that can be added to a contact, for instance to store a photo of a person in a `Contact` object. Analogous to the string and date fields, methods for setting and retrieving the contents of that field are provided. The only supported field ID for a byte array type is the predefined `PHOTO` constant to store a person's photograph.

For setting a byte array, use the `setBinary()` method, which takes a field ID and the byte array to be stored in the contact. For retrieving the byte array, use `getBinary()`, which takes a field ID as the only parameter.

Multivalue Data

Multivalue fields are used to store multiple instances of the same field. For instance, a field of a `Contact` might contain different types (locations) of phone numbers, such as home or work phone numbers. All phone numbers are stored in the different subentries of the same `TEL` entry. Typed data is added to a contact using the set `setTypedString()` method that takes three parameters: the field ID, the type ID, and the value to be stored. `Revision` is a read-only field that is set by the implementation automatically.

Table 7.2 shows the fields that can take multiple values and the corresponding predefined type IDs.

Table 7.2. Multivalue Contact Fields and the Corresponding Type IDs

Field IDs	*Type IDs*
`EMAIL`, `FAX`, `TEL`	`TYPE_ASSISTANT`, `TYPE_AUTO`, `TYPE_HOME`, `TYPE_MOBILE`, `TYPE_OTHER`, `TYPE_PAGER`, `TYPE_WORK`
`ADDR_COUNTRY`, `ADDR_EXTRA`, `ADDR_LOCALITY`, `ADDR_POBOX`, `ADDR_POSTALCODE`, `ADDR_REGION`, `ADDR_STREET`	`TYPE_HOME`, `TYPE_OTHER`, `TYPE_WORK`

The following code snippet shows how two phone numbers, the home and work phone number, are assigned to a contact:

```
myContact.setTypedString(Contact.TEL, Contact.TYPE_WORK, "555-1177");
myContact.setTypedString(Contact.TEL, Contact.TYPE_HOME, "555-7711");
```

Device-Specific Meta Information

Because the currently available PDA devices have different native PIM implementations, the `PIMList` interface provides query methods for retrieving information about all supported fields and the supported types for a given field ID.

For obtaining this information, the `getSupportedFields()`, `isFieldSupported()` and `getSupportedTypes()` methods are provided. The `getSupportedFields()` method returns an integer array containing all field IDs that are supported by the device. The method `IsSupportedField()` returns a boolean determining whether the given field ID is supported. The `getSupportedTypes()` method takes a field ID as a parameter and returns an `int` array containing all possible subtypes for this field. For example, you can query the possible type IDs for the `Contact.TEL` field of a given `ContactList` using the following code snippet:

```
int [] possibleSubTypes =
    myContactList.getSupportedTypes(Contact.TEL);
```

On a device that is capable of storing only a home and a work phone number, the resulting int array contains the type IDs `Contact.TYPE_HOME` and `Contact.TYPE_WORK` only.

Extended Fields

A `Contact` may also provide support for so-called "extended" fields. Extended fields are supported to store device-dependent fields, where no field ID is predefined.

The IDs for extended fields are returned by the `getSupportedFields()` method of the `PIMList` interface. They are returned together with the standard fields in the same array. The method `isExtendedField()` can be used to determine whether a field is an extended field with a non-standard ID.

When working with extended fields, two additional methods of the `PIMList` interface are of special interest. The method `getFieldLabel()` returns a human-readable label for a given field ID. This method allows an application to display a label for an extended field, although the meaning of the ID is not known to the application. The method `PIMElement.getDataType()` returns one of the `PIMElement` constants `STRING`, `DATE`, `INT`, `BINARY`, or `TYPED_STRING`. This method allows the application to figure out the correct access methods for an extended field.

Categories

`Contact`s can be assigned to one or more categories using the `addToCategory()` method. The method takes the name of the category as `String` parameter. The maximum number of categories a contact may be assigned to can be queried using the `maxCategories()` method. A return value of -1 means that the number of categories is not limited. A `Contact` can be removed from a category using `removeFromCategory()`.

Please note that devices may limit the category names to those defined in the corresponding `PIMList`. If a `Contact` is added to an invalid category an `PIMListException` is thrown. `PIMList` provides the following methods to manage categories available: `addCategory()`, `deleteCategory()`, and `getCategories()`.

`ContactList`s: Creating, Updating, and Deleting `Contact`s

Now that you are familiar with the fields of the `Contact` interface, it is time to take a look at the `ContactList` interface, which encapsulates access to the persistent contact database.

The `ContactList` is derived from the `PIMList` interface, which offers the base functionality for handling collections of PIM elements. `ContactList` provides a `createContact()` method for creating new contacts.

When creating new contacts or modifying existing `contacts`, the `commit()` method of the Contact interface must be called in order to make the changes persistent. Without calling commit, a new contact will not be stored in the database.

The `PIMList` class supports a `deleteElement()` method, which takes a `PIMElement` as a parameter, can be used to delete the specified element from the database.

The `PIMList` class also provides two methods for retrieving an enumeration: an `elements()` method that returns all elements of the database, and a second `elements()` method that gets all elements matching the element passed as a parameter to this method. Finally, a `close()` method is provided.

The functionality to get access to the contact list(s) stored on the device is provided by the PIM class. The PIM class provides two static `openContactList()` methods and one method for retrieving the names of the non-default contact databases. The first `openContactList()` method takes the mode in which the contact database should be opened, either `PIM.READ_ONLY` or `PIM.READ_WRITE`. This method is used to open the default contact database. The second `openContactList()` method takes two parameters: the mode and the contact database name. In order to get an overview of all possible contact database names, the `listContactLists()` method is supported; it returns a string array containing the database names. The first entry in the array contains the name of the default database.

Now you are able to create a new `Contact` associated with the default `ContactList` and fill it with personal information, as shown in the code snippet below:

```
ContactList myContacts = PIM.openContactList (PIM.READ_WRITE);
Contact myContact = list.createContact;
myContact.setString(Contact.FORMATTED_NAME, "John Smith");
myContact.setString(Contact.TEL, TYPE_WORK, "555-1177");
myContact.setString(Contact.TEL, TYPE_HOME, "555-7711");
```

Once the contact is created, it can be added to the default contact database:

```
myContact.commit();
myContacts.close();
```

The second operation that is of importance when using a contact database is to update an already existing element. This might be necessary, for example, if the phone number of a given contact has changed. In order to update an element of the `ContactList`, you need to get the instance of the `Contact` first. The `ContactList` provides an `element()` method for searching existing `Contact`s, taking a `Contact` template as parameter. The template is filled with the information the search is based on. The `element()` method returns an enumeration containing all `Contact`s matching the template. A partially filled `Contact` object is used as template.

In the following code snippet, you can see how to retrieve a given contact, update a phone number, and then write it back to the database. There is no check whether the enumeration contains more than one elements. In a real application, a corresponding test would be appropriate:

```
ContactList myContacts = PIM.openContactList (PIM.READ_WRITE);
Contact matchContact = myContacts.create();
matchContact.setString(Contact.FORMATTED_NAME, "John Smith");
Enumeration enum = myContacts.elements(matchContact);
Contact myContact = (Contact)enum.nextElement();
myContact.setString(Contact.TEL, TYPE_HOME, "555-1177");
myContact.commit();
myContacts.close();
```

In order to make sure that only the desired contact is returned, the UID field could be used in the template instead of the formatted name. Another important operation provided by the `ContactList` is the ability to delete a previously added `Contact` from the database. This can be performed by passing the element to be deleted to the `removeElement()` method. So, you have to read the particular contact as you did when updating a contact. You can use the same code snippet used for updating a contact, except that you don't call `setString()` to modify a phone number; instead, you pass the element stored in `myContact` directly to the `removeContact()` method after you read it from the enumerator. This procedure is shown in the following code snippet:

```
ContactList myContacts = PIM.openContactList (PIM.READ_WRITE);
Contact matchContact = myContacts.create();
matchContact.setString(Contact.FORMATTED_NAME, "John Smith ");

Enumeration enum = myContacts.elements(matchContact);
Contact myContact =(Contact)enum.nextElement();
myContacts.removeContact(myContact);

myContact.commit();
myContacts.close();
```

Calendar API

Similar to the addressbook API, the calendar API supports most of the fields specified by the vCalendar in IETF RFC 2445. The calendar API supports two classes analogous to those in the addressbook API.

The following code snippet shows a vCalendar entry:

```
BEGIN:VCALENDAR
VERSION:2.0
PRODID:-//hacksw/handcal//NONSGML v1.0//EN
BEGIN:VEVENT
DTSTART:19970714T170000Z
DTEND:19970715T035959Z
SUMMARY:Bastille Day Party
END:VEVENT
END:VCALENDAR
```

As you can see, the format is very similar to the vCard format used in the addressbook API.

Note

More information about IETF RFC 2445 can be found at

http://www.nic.mil/ftp/rfc/rfc2445.txt

The calendar API supports two interfaces analogous to those in the addressbook API (one interface extending the `PIMElement` interface and one interface derived from the `PIMList` interface, providing access to an event database).

Here the corresponding interfaces are the `Event` interface for a particular element and the `EventList` interface, providing the necessary methods to access an event database.

`Contact`s and `Event`s support different field IDs. The calendar API supports two interfaces analogous to those in the addressbook API (one interface extending the `PIMElement` interface and one interface derived from the `PIMList` interface, providing access to an event database).

Here the corresponding interfaces are the `Event` interface for a particular element and the `EventList` interface, providing the necessary methods to access an event database.

Repetition of Events

The EventRepeat class is an encapsulation of the RRULE field in a vCalendar element. It is used to determine how often an event occurs. The repetition details of the event are set using the setInt() method, taking a field ID and an int value as parameter. The valid parameter combinations are shown in Table 7.3. Additionally, an end date for the repetition can be set using the setDate() method, taking the END field constant and a valid date as parameters.

Table 7.3. Field ID Constants and valid values for the setInt() method of the EventRepeat class

Field IDs	*Valid Values*
COUNT	any positive int
FREQUENCY	DAILY, WEEKLY, MONTHLY, YEARLY
INTERVAL	any positive int
MONTH_IN_YEAR	JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
DAY_IN_WEEK	SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
WEEK_IN_MONTH	FIRST, SECOND, THIRD, FOURTH, FIFTH, LAST, SECONDLAST, THIRDLAST, FOURTHLAST, FIFTHLAST
DAY_IN_MONTH	1-31
DAY_IN_YEAR	1-366

The following snippet shows how a repeat pattern is added to an event:

```
EventList myEvents = PIM.openEventList(PIM.READ_WRITE);
Event myEvent = myEvents.createEvent();

Date startDate = new Date();
myEvent.setString(Event.SUMMARY, "Weekly developer meeting.");
myEvent.setDate(Event.START, startDate);
myEvent.setDate(Event.ALARM, new Date(startDate.getTime() - 60000));
EventRepeat repeat = new EventRepeat();
repeat.setInt(EventRepeat. DAY_IN_WEEK, EventRepeat.MONDAY);

myEvent.setRepeat(repeat);

myEvent.commit();
myEvents.close();
```

In the following section, we will describe the list related calls used in this code snippet in more detail.

EventLists for Handling Events

For access to the event database, the PIM class provides openEventList() and listEventLists() methods analogous to the methods available for obtaining contact lists. The behavior of those methods is as described in the previous section about contact lists, except that the openEventList() methods both return instances of EventList. Like ContactList, EventList extends the general PIMList interface. In addition to the PIMList functionality, it provides a new elements() method, returning an enumeration containing all Event elements ranging from a start date to a specified end date.

ToDo API

The to-do API supports a subset of the vToDo fields that are defined in the vCalendar RFC 2445 specification. Analogous to the addressbook and calendar API, the to-do API contains specializations of the `PIMElement` and `PIMList` interfaces, namely the `ToDo` and `ToDoList` interfaces.

Field IDs supported in the `ToDo` interface are `COMPLETED`, `DUE`, `NOTE`, `PRIORITY`, `SUMMARY`, and `UID`. The following code snippet illustrates the usage of the ToDo API:

```
TodoList myTodoList = PIM.openTodoList(PIM.READ_WRITE);

Todo myTodo = myTodoList.createTodo();

myTodo.set(Todo.SUMMARY, "Buy a book covering MIDP and PDAP");
myTodo.set(Todo.DUE, new Date());
myTodo.set(Todo.NOTE, "Perhaps Sams offers a good one."
        + "Take a look on their website first.");
myTodo.set(Todo.PRIORITY, 1);

myTodo.commut();
myTodoList.close();
```

Like the `EventList`, the `ToDoList` supports an `elements()` method that returns an enumeration containing all `ToDo` elements ranging from a start date to an end date. Repeat patterns are not supported for `ToDo`s.

Contact Sample Application

In order to become familiar with the PDAP PIM API, you will now build a simple PIM sample application. Although the sample will focus on the address book part of the API, it will be designed in a structured way that allows you to reuse many parts as building blocks for your own, more powerful PIM applications.

For the PIM sample application, you will first design a simple dialog that allows you to edit a single contact, including fields with subtypes. The second building block of the application is the main window, showing the list of contacts stored in the default `ContactList` of the device.

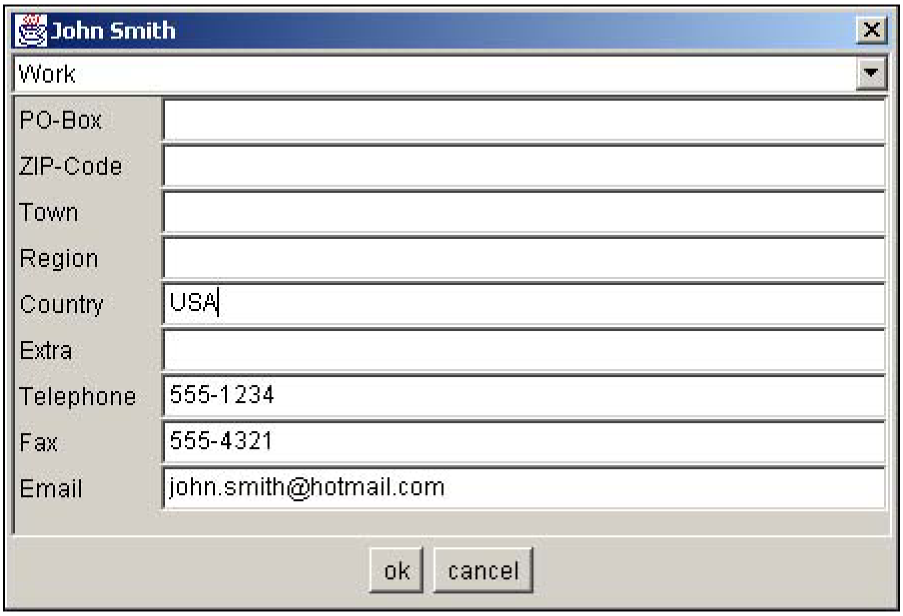
An Edit Dialog for Contacts

For the regular contact fields, it is obvious how to present them to the user in a graphical user interface. You just put them in two grid layouts contained in the right and center areas of a border layout; the left grid shows the labels and the center one the input fields, as in many other examples. However, for the fields supporting multityped values such as `TEL` or `FAX`, the task becomes a bit more complex.

One option to make all subtypes accessible in a convenient way is to create separate panels for each subtype such as `WORK`, `HOME`, and so on using a `CardLayout`. A `Choice` element can then be used to switch between the panels.

This approach is shown in Listing 7.1. Figure 7.2 shows a screenshot of the contact dialog. The panels are represented by the inner class `FieldPane`. `FieldPane` contains an integer variable `type` representing the subtype of all contained fields. The method `addField()` adds a field with the given label and ID. In order to be able to synchronize the user interface with a concrete `Contact` instance, the `FieldInfo` helper class is used to store the link between the `TextField` in the user interface and the ID and type of the corresponding field. All `FieldInfo` objects are stored in the vector `fieldList`. When the dialog is filled from a contact in the edit method, all field info objects are iterated, and the TextFields are filled according to the corresponding content of the contact. If the user confirms the dialog, the FieldInfo objects are iterated again, and the contents of the TextFields, possibly altered by the user, are transferred back to the contact. Note that the transfer to a contact object always needs to be committed to the database in order to become persistent.

Figure 7.2. The ContactDialog showing a sample contact.



The addField() method of the FieldPane inner class is not accessed in the ContactDialog directly, but ContactDialog has its own addField() method. Depending on the given type parameter, the STRING fields are added directly to the main FieldPane, whereas fields with MULTIPLE subtypes are distributed to all corresponding panels. For this purpose, the subtypes supported by the platform are queried using the getSupportedTypes() method. Then, for each type, addSub() is called. In addSub(), the addField() method of the corresponding FieldPane is called. Additional panels and choice entries are created as needed.

Note that the given add method supports string and multitype fields only, but no date or binary fields. However, it should not be a problem to support those types by adding corresponding constants and user interface elements.

Listing 7.1 ContactDialog.java—A Dialog for a Single Contact

```
import java.awt.*;
import java.awt.event.*;
import javax.microedition.pim.*;
import java.util.Hashtable;
import java.util.Vector;

public class ContactDialog extends Dialog
    implements ItemListener, ActionListener {

    Hashtable cards = new Hashtable();
    Contact contact;
    ContactList contactList;
    Panel cardPane = new Panel(new CardLayout());
    Choice typeChoice = new Choice();
    Button okButton = new Button("ok");
    Button cancelButton = new Button("cancel");
```

```
boolean result = false;
Vector fieldList = new Vector();

class FieldInfo {
    int id;
    int type;
    TextField field;
}

class FieldPane extends Panel {
    Panel labels = new Panel(new GridLayout(0, 1));
    Panel fields = new Panel(new GridLayout(0, 1));
    int type;

    FieldPane(int type) {
        super(new BorderLayout());
        this.type = type;
        add(labels, BorderLayout.WEST);
        add(fields, BorderLayout.CENTER);
    }

    void addField(int id, int type) {
        FieldInfo info = new FieldInfo();
        info.id = id;
        info.field = new TextField(30);
        info.type = type;
        labels.add(new Label(contactList.getFieldLabel(id)));
        fields.add(info.field);
        fieldList.addElement(info);
    }
}

public void actionPerformed(ActionEvent ev) {
    result = ev.getSource() == okButton;
    hide();
}

void addSub(int id, String type, int typeId) {
    FieldPane fieldPane = (FieldPane) cards.get(type);
    if (fieldPane == null) {
        fieldPane = new FieldPane(typeId);
        typeChoice.add(type);
        cards.put(type, fieldPane);
        Panel compact = new Panel(new BorderLayout());
        compact.add(fieldPane, BorderLayout.NORTH);
        ScrollPane sp = new ScrollPane();
        sp.add(compact);
        cardPane.add(sp, type);
    }
    fieldPane.addField(id, typeId);
}

void addField(int id) {
    if (id == Contact.UID || id == Contact.REVISION)
        return;
    int dataType = contact.getDataType(id);
    if (dataType == PIMElement.STRING)
        addSub(id, "Main", -1);
    else if (dataType == PIMElement.STRING_TYPED) {
        int[] types = contactList.getSupportedTypes(id);
        for (int i = 0; i < types.length; i++) {
```

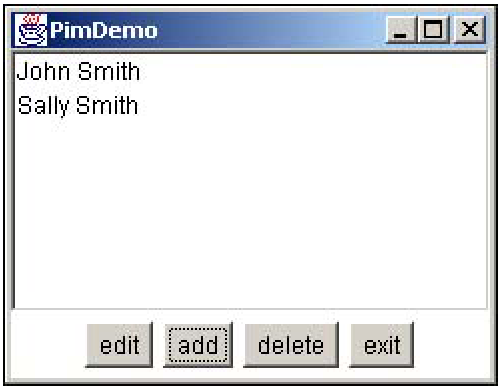
```
                switch (types[i]) {
                    case Contact.TYPE_HOME :
                        addSub(id, "Home", types[i]);
                        break;
                    case Contact.TYPE_WORK :
                        addSub(id, "Work", types[i]);
                        break;
                                    // add other types here
                }
            }
        }
        // other data types are ignored
    }

    public void itemStateChanged(ItemEvent e) {
        CardLayout cl = (CardLayout) (cardPane.getLayout());
        cl.show(cardPane, (String) e.getItem());
    }

    public ContactDialog(Frame frame, ContactList contactList) {
        super(frame, "Edit Contact", true);
        this.contactList = contactList;
        contact = contactList.createContact();
        add(typeChoice, BorderLayout.NORTH);
        add(cardPane, BorderLayout.CENTER);
        typeChoice.addItemListener(this);
        int[] ids = contactList.getSupportedFields();
        for (int i = 0; i < ids.length; i++)
            addField(ids[i]);
        Panel buttonPane = new Panel();
        buttonPane.add(okButton);
        buttonPane.add(cancelButton);
        okButton.addActionListener(this);
        cancelButton.addActionListener(this);
        add(buttonPane, BorderLayout.SOUTH);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent ev) {
                result = false;
                hide();
            }
        } );
        pack();
    }

    public boolean edit(Contact contact, String title) {
        setTitle(title);
        this.contact = contact;
        result = false;
        for (int i = 0; i < fieldList.size(); i++) {
            FieldInfo info = (FieldInfo) fieldList.elementAt(i);
            String text =
                contact.getDataType(info.id) == PIMElement.STRING
                    ? contact.getString(info.id)
                    : contact.getTypedString(info.id, info.type);
            info.field.setText(text == null ? "" : text);
        }
        show();
        if (result) {
            for (int i = 0; i < fieldList.size(); i++) {
                FieldInfo info = (FieldInfo) fieldList.elementAt(i);
                String text = info.field.getText();
```

```
                if (text.equals(""))
                    text = null;
                if (contact.getDataType(info.id) == PIMElement.STRING)
                    contact.setString(info.id, text);
                else
                    contact.setTypedString(info.id, info.type, text);
                info.field.setText(text == null ? "" : text);
            }
            contact.commit();
        }
        return result;
    }
}
```

The PimDemo MIDlet

Having implemented the dialog for editing a single contact, the main application is relatively simple. You just need to display a list of all entries contained in the address book, obtained from the corresponding `ContactList`. Listing 7.2 shows a corresponding sample application. Figure 7.3 shows the PimDemo MIDlet.

Figure 7.3. The PIMDemo application showing a list of all contacts added to the contact database.



Because the displayed names of the entries might not be unique, it makes sense to store the `UID`s in a separate vector. Of course, it would be possible to keep all the `Contact` objects in the vector directly, but this would demand a lot of heap memory, which might not be available on the PDA. The stored `UID`s are used in the `getContact()` method, where a given UID is translated back to a `Contact` object using the filtered contact enumeration.

The application displays buttons to add, edit, or remove an entry. When the add or edit button is pressed, a `ContactDialog` is shown, and the user can edit the content of the fields displayed. When the program control returns from the dialog, and the dialog was not cancelled, the changes are made persistent by calling the `addElement()` or `updateElement()` method of the `ContactList`.

Listing 7.2 `FieldDialog.java`—A Dialog, Based on the `FieldDescription` Class, for Editing Single Item Fields

```
import java.awt.*;
import java.awt.event.*;
import javax.microedition.midlet.*;
import javax.microedition.pim.*;
import java.util.*;

public class PimDemo extends MIDlet implements ActionListener {
```

```
    ContactList contactList = PIM.openContactList(PIM.READ_WRITE);
    Frame frame = new Frame("PimDemo");
    ContactDialog contactDialog = new ContactDialog(frame,
contactList);
    java.awt.List list = new java.awt.List();
    Vector uids = new Vector();
    Button editButton = new Button("edit");
    Button addButton = new Button("add");
    Button deleteButton = new Button("delete");
    Button exitButton = new Button("exit");

    public PimDemo() {
        for (Enumeration e = contactList.elements();
e.hasMoreElements();) {
            Contact c = (Contact) e.nextElement();
            uids.addElement(c.getString(Contact.UID));
            list.add(getLabel(c));
        }
        Panel buttonPane = new Panel();
        buttonPane.add(editButton);
        buttonPane.add(addButton);
        buttonPane.add(deleteButton);
        buttonPane.add(exitButton);
        editButton.addActionListener(this);
        addButton.addActionListener(this);
        deleteButton.addActionListener(this);
        exitButton.addActionListener(this);
        frame.add(buttonPane, BorderLayout.SOUTH);
        frame.add(list, BorderLayout.CENTER);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent ev) {
                destroyApp(true);
                notifyDestroyed();
            }
        } );
        frame.pack();
    }

    Contact getContact(String uid) {
        Contact template = contactList.createContact();
        template.setString(Contact.UID, uid);
        Enumeration e = contactList.elements(template);
        return (Contact) e.nextElement();
    }

    String getLabel(Contact c) {
        String label = c.getString(Contact.FORMATTED_NAME);
        if (label == null)
            label =
                "" + c.getString(Contact.NAME_FAMILY) + ", " +
                 getString(Contact.NAME_GIVEN);
        return label;
    }

    public void actionPerformed(ActionEvent ev) {
        System.out.println("ev: " + ev);
        if (ev.getSource() == addButton) {
            Contact c = contactList.createContact();
            if (contactDialog.edit(c, "New Contact")) {
                list.add(getLabel(c));
                uids.addElement(c.getString(Contact.UID));
```

```
            }
        }
        else if (ev.getSource() == exitButton) {
            destroyApp(true); // clean up
            notifyDestroyed();
        }
        else if (ev.getSource() == editButton || ev.getSource() ==
          deleteButton) {
            int index = list.getSelectedIndex();
            if (index == -1)
                return;
            Contact c = getContact((String) uids.elementAt(index));
            if (ev.getSource() == deleteButton) {
                contactList.deleteElement(c);
                list.remove(index);
                uids.removeElementAt(index);
            }
            else if (contactDialog.edit(c, getLabel(c))) {
                list.replaceItem(getLabel(c), index);
            }
        }
    }

    public void pauseApp() {
    }

    public void startApp() {
        frame.show();
    }

    public void destroyApp(boolean uncond) {
        contactList.close();
    }
}
```

What Is Missing?

The PimDemo sample application shows basic access to the PIM API, but for a real PIM application, a lot of functionality is missing. For example, categories are not supported, and the dialog shows only a fraction of the fields available. Event and to-do entries are not supported at all. Field types other than string and multityped fields are not supported. The contact list is not sorted, and a search function is missing. The delete button lacks a confirm dialog.

Use the example as a starting point for your own ideas and extensions, and feel free to reuse the code in your own applications.

Summary

In this chapter, you learned about the general design of the PIM API that is included in PDAP. You know about the different kinds of `PIMLists` and `PIMElements`. Moreover, you are familiar with the concept of the vCard and the vCalendar, you are able to create contact cards and calendar entries and you know how to store them in a list, and how to use the PIM API to create a PDAP application using this specific API.

Chapter 8. Size Does Matter: Optimizing J2ME Applications

IN THIS CHAPTER

Because of several computing and memory restrictions of mobile devices, you need to pay particular attention to the size and performance of J2ME applications.

The most important hint is probably to keep everything as simple as possible. If an application will be ported from the desktop, you may want to consider a re-implementation instead of trying to downsize the existing program until it fits into J2ME.

Although most optimization possibilities depend on the individual application, in this chapter we'll show you some general approaches and hints for saving resources. Unfortunately, there is often a tradeoff between execution speed and memory consumption. Thus, not all of our hints will make sense in all application scenarios.

Usually, memory in J2ME devices is divided into three different categories: program storage space, persistent memory, and heap memory. The heap memory is used at runtime only and holds all volatile objects. Depending on the device, there may be different limits on each of the types, making trade-off decisions even more complicated.

Reducing Class File Sizes

When importing libraries, the fully qualified name—including the class and package name of each imported function—is stored in the class file. Thus, limiting the number of imports may reduce the size of a class file significantly. One large class will probably consume less memory than a set of smaller classes, and the import overhead of additional convenience methods will often weigh more than the code actually saved.

Freeing Unused Variables and Resources

The Java garbage collection mechanism usually frees unused objects automatically. "Unused" from the point of view of the garbage collector means that the objects are not referenced from somewhere else. If objects are no longer needed in a program, but are still referenced by a variable or indirectly by another object, the garbage collector can't determine that the object can be removed, and the corresponding memory is not reclaimed. Thus, if an object is allocated and then no longer needed, but the variable holding the object is still in the valid scope, it may make sense to set the variable to `null` explicitly. For example, if a buffer is allocated at the beginning of a method, but then no longer needed in the method, the garbage collector cannot reclaim the buffer because the variable still points to the buffer. If the variable is explicitly set to `null`, the buffer is no longer referenced and can be reclaimed by the garbage collection.

For J2ME, it is also very important to always dispose resources such as record stores or connections when they are no longer needed. J2ME does not support finalization, which means that system resources cannot be closed automatically during garbage collection. Thus, if the reference to a system resource is removed without closing the resource, the resource will stay allocated until system cleanup, when the program is terminated completely.

Loop Condition Checking

When you're trying to optimize execution speed, loops are the most important pieces of code to look at: statements inside a loop are executed several times. Thus, even small optimizations can have a significant effect.

A standard method to speed up loops is to move constant expressions out of the loop. For example, the following simple `for` loop iterates all elements of the `Vector vector`:

```
for (int i = 0; i < vector.size(); i++) ...
```

In this line, the `size()` method of `vector` is called at each iteration. This repetition can be avoided by storing the size in a local variable:

```
int size = vector.size();
for (i for (int i = 0; i < size; i++) ...
```

Please note that access to local variables is generally less expensive than access to instance or class variables. Here, the `size()` method is called only once and stored in a local variable for fast access inside the loop. If the direction the vector is iterated is not important, counting down may be a reasonable alternative to introducing a new variable:

```
for (int i = vector.size()-1; i >= 0; i--) ...
```

The same technique can be applied to all expressions that are calculated inside the loop, but do not change with the iterations.

Avoiding Recursion

Another common optimization technique that may speed up program execution and that definitely saves memory is to transform recursions into iterations. You can usually do so if there is only one simple recursion call. For example, the factorial of a number *n* is defined as *n* multiplied by the factorial of (*n*-1). The corresponding recursive function is

```
public static void fact (int n) {
    return n == 1 ? 1 : n * fact (n – 1);
}
```

The problem with recursion is that it consumes stack space: The return address and local variables are stored on the program stack when a method is called. Also, the program state changes that correspond to a method call might take more time than a simple loop.

Thus, for the function defined previously, the following iterative function may be more efficient:

```
public static void fact (int n) {
    int result = 1;
    while (n > 1) result *= n--;
    return result;
}
```

Obviously, the tradeoff is that the iterative constructs are often less readable than the recursive versions.

Using Arrays Instead of Vectors

`Vector`s are powerful and flexible containers for all kinds of objects. Their main advantage over arrays is that their size grows dynamically as needed. The disadvantages are

- Many type casts may be needed because a `Vector` can hold any kind of objects.
- `Vector`s carry some overhead compared to a corresponding array. Actually, `Vector`s use an array internally to store their data. Thus, using a `Vector` requires an additional indirection step and an additional object when compared to using an array.
- The `Vector` access methods are synchronized, which also results in some performance tradeoff. The fact that the size of an array cannot change may help to achieve thread safety without synchronization.

For these reasons, in some cases it may make sense to use arrays instead of `Vector`s, especially if the size of the structure does not change frequently. However, using an array instead of a `Vector` is not free of costs. You can't change the size of an array. If an array must be extended, you have to allocate a new array and copy its entire contents. Thus, for structures that frequently change size, sticking to a `Vector` may be a better choice. An alternative that is also used internally by the `Vector` class is to allow that the array is larger than necessary. The actual size is stored in a separate variable in that case.

Table 8.1 shows some `Vector` operations and the corresponding counterparts for arrays.

Table 8.1. `Vector` Operations and the Corresponding Counterparts for Array Access

`Vector`	***Array***
`(MyObject) v.elementAt (i);`	`v[i];`

`v.addElement (o);`	`v2 = new MyObject[v.length+1];`
	`System.arraycopy (v, 0, v2, 0, v.length);`
	`v2 [v.length] = o;`
	`v = v2;`

`v.removeElementAt (i);`	`v2 = new MyObject [v.length-1];`
	`System.arraycopy (v, 0, v2, 0, i-1);`
	`System.arraycopy (v, i+1,  v2, i, v.length - i);`
	`v = v2;`

`v.insertElementAt (o, i);`	`v2 = new MyObject [v.length+1];`
	`System.arraycopy (v, 0, v2, 0, i-1);`
	`System.arraycopy (v, i+1, v2, i+1, v.length);`

	`v2[i] = o;` `v = v2;`
`v.setElementAt (o, i);`	`v[i] = obj;`

Using Record Stores Instead of Heap Memory

An additional opportunity for saving heap space is to store application data in a record store instead of consuming heap memory. The space available for persistent storage is often significantly larger than the heap memory, so in situations where heap memory is really rare, it may make sense to shift some of the memory consumption from the heap to persistent storage. However, accessing persistent storage may be significantly slower than heap access. Thus, the price for having more heap may be a significant performance tradeoff.

In order to demonstrate how to store objects in a record store instead of a `Vector`, let's create a sample implementation of a `StringVectorRms` class that stores a list of `String`s in a record store. In contrast to an ordinary `Vector`, only a small amount of heap is consumed. The `StringVectorRms` implementation provides the access methods listed in Table 8.2.

Table 8.2. Methods of the `StringVectorRms` Class

Method	***Description***
`StringVectorRms()`	Constructs an instance and creates the underlying record store if needed.
`addString (String text)`	Adds a `String` to the end of the `Vector`.
`String stringAt (int index)`	Returns the `String` at the given index.
`removeStringAt (int index)`	Removes the `String` at the given index and shifts the entries starting at `index`+1 down in order to fill the resulting gap.
`SetStringAt (String newText, int newIndex)`	Replaces the `String` at position `index` with the new `String` that is contained in the variable `newText`.
`int size()`	Returns the number of `String`s that are currently stored.

Listing 8.1 shows the corresponding `StringVectorRms` implementation. It handles the conversation between `Strings` and the `byte` arrays stored in the record store. It also maps the `Vector` indices starting with 0 to RMS indices starting with 1 and maps all `RmsException`s to `RuntimeException`s. Finally, it takes care of shifting the remaining elements to their new index if an element is deleted.

Note

Especially for flash memory, write operations might take seconds. Thus, the `removeStringAt()` method should be used with special caution; all remaining items are moved to their new index position, requiring a lot of write operations.

Listing 8.1 `StringVectorRMS.java`—The `StringVectorRMS` Class for Storing `String`s in a Record Store During Application Runtime

```
import javax.microedition.rms.*;
```

```
public class StringVectorRms {
    RecordStore store;
    String storeName;

    int size;

    static int openedStores = 0;


    public StringVectorRms() {
        storeName = "StringVectorRms"+(openedStores++);
        size = 0;
        try {
            store = RecordStore.openRecordStore (storeName, true);
            if (store.getNumRecords() > 0) {
            store.closeRecordStore();
            RecordStore.deleteRecordStore (storeName);
            store = RecordStore.openRecordStore (storeName, true);
            }
        }
        catch (Exception e) {
            throw new RuntimeException (e.toString());
        }
    }


    public void addString (String text) {
        try {
            byte[] data = text.getBytes();
            if (size < store.getNumRecords())
                store.setRecord (size+1, data, 0, data.length);
            else
                store.addRecord (data, 0, data.length);
            size++;
        }
        catch (Exception e) {
            throw new RuntimeException (e.toString());
        }
    }


    public String stringAt (int index) {
        try {
            return new String (store.getRecord (index+1));
        }
        catch (Exception e) {
            throw new RuntimeException (e.toString());
        }
    }


    public void removeStringAt (int index) {
        try {
            for (int i = index; i < size-1; i++) {
                byte[] data = store.getRecord (i+2);
                store.setRecord (i+1, data, 0, data.length);
            }
            size--;
        }
        catch (Exception e) {
            throw new RuntimeException (e.toString());
```

```
        }
    }

    public void setStringAt (String newText, int newIndex) {
        try {
            byte[] newData = newText.getBytes();
            store.setRecord (newIndex+1, newData, 0, newData.length);
        }
        catch (Exception e) {
            throw new RuntimeException (e.toString());
        }
    }

    public int size() {
        return size;
    }

    public void close() {
        try {
            store.closeRecordStore();
            RecordStore.deleteRecordStore (storeName);
        }
        catch (Exception e) {
            throw new RuntimeException (e.toString());
        }
    }
}
```

Distributing Functionality over Several Small MIDlets

Another way to save memory is to distribute the functionality of an application to two or more smaller applications in the same MIDlet suite. This approach seems especially feasible if the application consists of mostly independent building blocks, which only share their persistent data. An example could be any loosely connected client application, mainly operating on a local database, but synchronizing data with a server from time to time. Instead of integrating the synchronization in the main application, it can be moved to a separate application. Thus, the size of the main application can probably be reduced significantly. Other parts of the application, such as configuration dialogs, can probably be spun off as well.

Please note that you can split MIDlets only if all the applications are stored in the same suite; otherwise, they can't share their persistent data.

Fragmentation Problems

A frequent problem for J2ME applications that are running on a KVM implementation without compacting garbage collection is a *memory paradox:* `Runtime.freeMemory()` reports lots of free memory, but a memory allocation following immediately fails and causes an `OutOfMemoryException`.

This issue was especially problematic with early versions of the SUN KVM. The SUN KVM now provides a compacting garbage collector, so this problem seems less important. However, some KVM implementations may not provide a compacting garbage collector for some reason, so you should at least know about the characteristics of the problem.

The reason for the discrepancy between the return value of `Runtime.freeMemory()` and the ability to allocate a certain block of memory is usually a fragmentation problem: Lots of memory is available, but only in very small pieces. `Runtime.freeMemory()` reports the total amount of memory available, but the largest continuous block of memory may be much smaller.

The garbage collection for J2SE is usually implemented using a *compacting* algorithm—all memory blocks that can be reclaimed are compacted to a single large block of memory. In the KVM, free memory blocks are reclaimed, but the compaction step may be omitted for performance and complexity reasons. Thus, a fragmentation problem may occur when a lot of small memory blocks are allocated. Even if most of the small blocks can be reclaimed during garbage collection, the remaining blocks may fragment the free memory into several small pieces.

Although no general solution exists for the fragmentation problem, it may help to call the garbage collector explicitly at some points in the program. Explicit calls of `Runtime.gc()` will force a garbage collection before all the memory is used up. Thus, only part of the memory is fragmented. New memory allocations are served from the small chunks before the remaining block is touched. Please note that explicit calls to `Runtime.gc()` may help, but will not do so in all cases, depending on the concrete memory consumption behavior of the application and on the total amount of memory available on the device.

User Interface Issues

Compared to the desktop, the screen space available on MIDP devices or even PDAs is very limited. This limited space requires a careful user interface design. Also, the situations in which mobile devices are used are different. Whereas desktop applications are normally used for a longer period of time, mobile applications are usually used for just a few seconds, but more frequently. Thus, access to the desired information should be fast and navigation as simple as possible.

It should also be possible to leave the application at any point without loss of data. The reason is that in mobile application scenarios, the user may want to switch or leave applications quickly—for example, to answer an incoming call, when the subway reaches the destination station, or when the airplane pilot asks passengers to switch off all electronic devices during landing. Thus, nested dialogs should be avoided.

The built-in Palm Pilot applications provide very good examples for appropriate PDA user interface design. Before designing your own applications, it makes sense to look at some other typical PDA or MIDP applications, in order to get a feel for their design.

MIDP

The MIDP UI was designed for limited screens from the beginning, so it's difficult to give general hints for improvements that don't depend on the individual application.

Because of the limited screen size, you'll often need to distribute the user interface to several screens. Although commands provide one possibility to switch between forms, an important alternative for selecting actions is to use a `List` in the `IMPLICIT` mode. Selecting a command may require going to a submenu if the key mapping is not sufficient, but the list is always displayed directly on the main screen. Also, `List Item`s can hold an `Image`, which is not possible for commands.

PDAP

The PDAP user interface is based on the Abstract Windows Toolkit (AWT) that was originally developed for the desktop. It provides more design freedom than the MIDP UI elements, but it also loads more responsibility on the developer: Screen formats and sizes may differ significantly from device to device, and the application should try to adopt to these characteristics. For example, a configuration panel may fit completely on the screen of one device, but on another device a split may be necessary. You could achieve such a division by putting both parts in panels and switching between a `CardLayout` and a `BorderLayout`, depending on the actual screen size of the device.

The `Choice` class is a space-saving, comfortable widget. For PDA applications, it can often be used as a space-saving alternative to a set of radio buttons or a number of other buttons. The combination of a `Choice` with a `Cardlayout` may be used to simulate tab panes for PDAP.

Summary

In this chapter you learned about some possible program optimizations, allowing enhancement of your J2ME applications with respect to execution speed and memory consumption. You now know how to shift application data from the heap to record stores. You have seen some cases in which it may make sense to split the functionality of an application into a few separate programs in the same MIDlet or PDAlet suite. Finally, you saw some ways to optimize the user interface of MIDP and PDAP applications.

The next chapter presents an advanced application example that illustrate integrated usage of the MIDP and PDAP APIs, which have been described separately in the previous chapters. We will show how to build an application for both MIDP and PDAP, sharing most of the classes, and differing only in the implementation of the user interface.

Chapter 9. Advanced Application: Blood Sugar Log

IN THIS CHAPTER

This chapter demonstrates building integrated software that utilizes various CLDC APIs and provides interfaces for both MIDP and PDAP while using identical classes for data handling. Further, the application demonstrates the combination of the low-level and high-level MIDP API.

Here, we use blood sugar logging as an example application. People suffering from diabetes often keep a daily log of their blood sugar level in order to keep track of their blood sugar values and adjust the amount of insulin for their daily injection accordingly. They keep a record of their blood sugar levels at several times every day. Our application can help those people by enabling them to use their PDA or MID for keeping their daily log.

The purpose of this application is to log blood sugar values and graphically display the blood sugar measurement values that have been taken during one day. However, the main purpose of this application is to demonstrate the separation of the application logic from the user interface and combine various CLDC APIs in a single program. Parts of the program can be reused for building your own applications. For example, the graphics display could be used for building a stock quotes display.

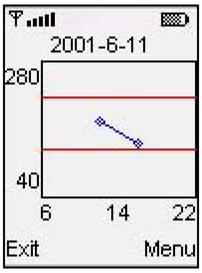
We will also utilize the example to show efficient binary search on record stores.

Requirement Analysis

The first step in building our application is to do a requirement analysis. What functionality will the application perform? Here, we want to provide the following functionality to the user:

- Display the blood sugar values of the current day as a diagram, as shown in Figure 9.1. The x-axis is the time, ranging from 6 to 22 o'clock. The y-axis is the blood sugar level in mg/dl ranging from 40 to 280.

Figure 9.1. The running BloodSugarMidp application showing the logged blood sugar values of the current day.



- Track hypoglycemic values that are lower than 80 mg/dl and hyperglycemic values that are higher than 160 by drawing red lines on the chart at 80 and 160 mg/dl.

- Enable the user to enter new values.
- Enable the user to delete erratic values.
- Enable the user to switch to previous logs.

In accordance with the design guidelines, persistent storage shall be performed in the background.

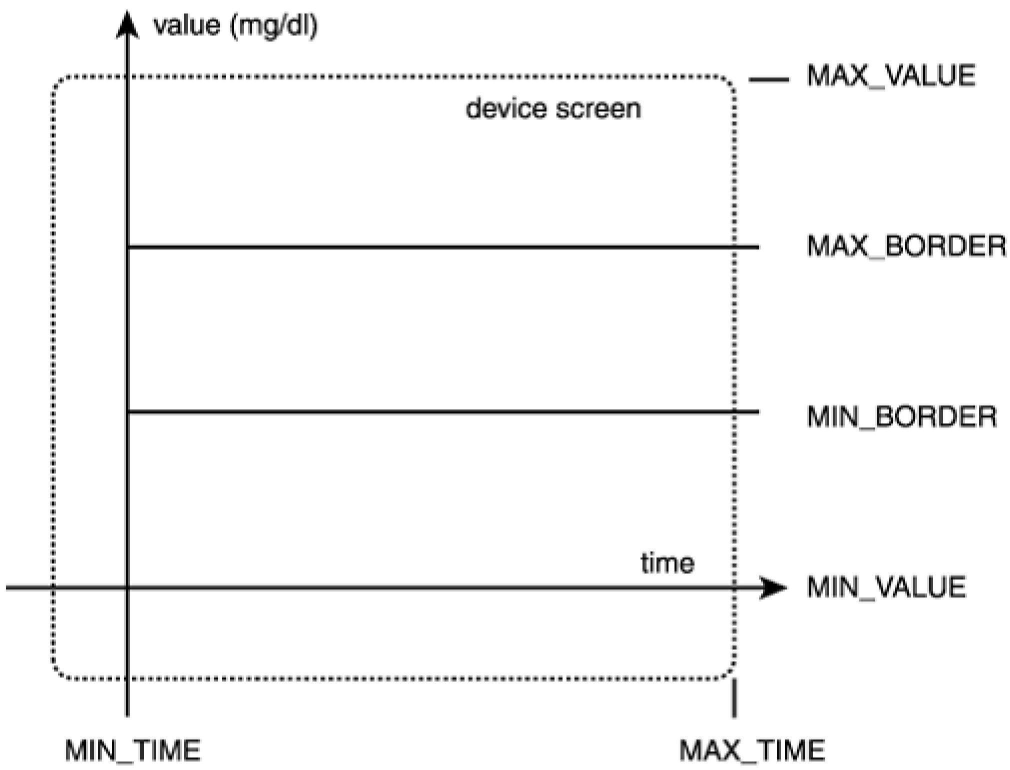
Day Log

In order to provide the desired functionality, we need two main data structures:

- A data structure for the current day, holding the date and the values and times for the day's measurements
- A data structure storing all the days persistently

Thus, the day log data structure needs a field holding the date and a list of the time and value pairs. The date is stored in the integer variable date, holding the day of the month in the lowest byte, the month in the second-lowest byte, and the year in the two upper bytes. We are using this format instead of the `Date` or `Calendar` object for simplified comparison, required for keeping the log entries in the right order. To store the pairs, a local class `Entry` is used, consisting of two integer values: `time` and `value`. Similar to the date field, we use an integer to represent the time, measured in minutes since midnight. Again, the main reason is simpler comparison. We also use the `DayLog` class to define the minimum and maximum value constants for the graphics display (see Figure 9.2).

Figure 9.2. The DayLog constants.



Our basic `DayLog` class looks as follows:

```
import java.io.*;
import java.util.*;
```

```
public class DayLog {

    public static final int MIN_TIME = 6 * 60;
    public static final int MAX_TIME = 22 * 60;
    public static final int MIN_VALUE = 40;
    public static final int MAX_VALUE = 280;
    public static final int MIN_BORDER = 70;
    public static final int MAX_BORDER = 160;

    int date; // yymd per byte
    Vector entries = new Vector();

    class Entry {
        int time; // minutes since midnight
        int value;
    }
}
```

For creating new `DayLog`s, a constructor, initializing the structure with the given date, is required:

```
public DayLog (int date) {
    this.date = date;
 }
```

In order to fill the structure with data, a `set` method, adding a new entry, is needed. The following method adds a new entry with respect to the correct time ordering of the entries. If an entry for the given point of time already exists, it is overwritten:

```
public void set (int minutes, int value) {
    Entry entry = new Entry();
    entry.time = minutes;
    entry.value = value;

    for (int i = getCount()-1; i >= 0; i--) {
        int minutesI = getTime (i);

        if (minutes <= minutesI) {
            if (minutes == minutesI)
                entries.setElementAt (entry, i);
            else
                entries.insertElementAt (entry, i+1);
            return;
        }
    }

    entries.insertElementAt (entry, 0);
}
```

Storing information in the day log data structure does not make sense if the information cannot be read back. Thus, we add access methods for the date, the number of entries, and the time and value of an entry at a given index:

```
public int getDate() {
    return date;
}

public int getCount() {
    return entries.size();
```

```
}

public int getTime (int index) {
    return ((Entry) entries.elementAt (index)).time;
}

public int getValue (int index) {
    return ((Entry) entries.elementAt (index)).value;
}
```

Finally, we add a method for deleting an entry. The method takes a point of time as input and removes the best matching entry:

```
public void remove (int time) {

    int bestDelta = 24;
    int bestIndex = -1;

    for (int i = 0; i < entries.size(); i++) {
        int delta = Math.abs (time - getTime (i));
        if (delta < bestDelta) {
            bestDelta = delta;
            bestIndex = i;
        }
    }

    if (bestIndex != -1) {
        entries.removeElementAt (bestIndex);
    }
}
```

Serialization and Deserialization for Persistent Storage

Because the `DayLog` is intended to be stored persistently, conversion methods from and to byte arrays are necessary. As described in Chapter 5, "Data Persistency," we use a `ByteArrayInputStream` for deserialization and a `ByteArrayOutputStream` for serialization:

```
public DayLog (byte [] data) throws IOException {

    DataInputStream dis = new DataInputStream
        (new ByteArrayInputStream (data));

    date = dis.readInt();
    int count = dis.readInt();

    for (int i = 0; i < count; i++) {
        Entry entry = new Entry();
        entry.time = dis.readInt();
        entry.value = dis.readInt();
        entries.addElement (entry);
    }

    dis.close();
}


public byte [] getByteArray() throws IOException {
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream (bos);
```

```
    dos.writeInt (date);
    int size = entries.size();
    dos.writeInt (size);
    for (int i = 0; i < size; i++) {
        dos.writeInt (getTime (i));
        dos.writeInt (getValue (i));
    }

    dos.close();

    return bos.toByteArray();
}
```

Helper Methods for the User Interface

Although the user interface itself is specific to the target profile, we can simplify the user interface code by providing some helper methods in the `DayLog` class. Here, we add a set of methods for two purposes:

- Converting the date of the `DayLog` to a readable string
- Scaling the entries to a given display size for simplified drawing

In order to convert the date to a `String`, we just extract the year, month, and day of month using the corresponding shift and mask operations. Those parts are concatenated with hyphens:

```
public String getTitle() {
    return "" + (date >> 16)
        + "-" + ((date >> 8) & 0x0ff)
        + "-" + (date & 0x0ff);
}
```

In order to scale a value to the size of the screen, we would usually normalize it by subtracting the `MIN_VALUE` and dividing by the difference of `MAX_VALUE` and `MIN_VALUE`, and then scale it up to the space available by multiplying by the size available. Because we do not have floating-point numbers in CLDC, we swap the normalization and scaling steps. Thus, we first multiply by the screen size and then divide by the value range. By first multiplying and then dividing, we make sure that we do not leave the scope of the integer data type:

```
public static int getY (int value, int size) {
    return - (value - MIN_VALUE) * size / (MAX_VALUE - MIN_VALUE);
}
```

In the `getYPoints()` method, we build an array of all values of the `DayLog`, scaled to the given screen size, utilizing the `getY()` method:

```
public int [] getYPoints (int size) {

    int [] y = new int [getCount()];

    for (int i = 0; i < getCount(); i++) {
        y [i] = getY (getValue (i), size);
    }

    return y;
}
```

The `getXPoints()` method is analogous to `getYPoints()`, except that we do not define a `getX()` helper method but perform the scaling step immediately in the loop:

```
public int [] getXPoints (int size) {

    int [] x = new int [getCount()];

    for (int i = 0; i < getCount(); i++) {
        x [i] = (getTime (i)-MIN_TIME) * size / (MAX_TIME-MIN_TIME);
    }

    return x;
}
```

Finally we add a `parseTime()` method in order to convert a time string such as `12:00`, consisting of an hour and a minute value separated by a colon, to our internal time format. For MIDP, instead of this conversion we could also use the time selector provided by `DateField`.

```
public static int parseTime (String time) {
    int cut = time.indexOf (':');
    if (cut==-1)
        return Integer.parseInt (time) * 60;

    return Integer.parseInt (time.substring (0, cut)) * 60
        + Integer.parseInt (time.substring (cut+1));
}
```

Persistent Storage: The LogStorage Class

Beneath the `DayLog` class for storing the log of a single day, we need another class to store all the `DayLog`s persistently. The `LogStorage` class uses a record store for that purpose.

In the constructor, the record store with the name `"BloodSugarLog"` is opened and assigned to the `days` object variable:

```
import javax.microedition.rms.*;
import java.util.*;
import java.io.*;
import java.util.*;


public class LogStorage {

    RecordStore days;
    public LogStorage() throws RecordStoreException {
        days = RecordStore.openRecordStore ("BloodSugarLog", true);
    }
}
```

The most important functionality of the persistent storage is to provide efficient access to the day log of a specific date. If the logs are stored and put in order by time in the record store, we do not need to iterate all records in order to find a specific date. Instead, we can take advantage of the ordering and perform a so-called *binary search*. We just pick the middle element and compare it to our target date. Now we know in which half we need to continue the search. Thus, we reduce the questionable records by half with each iteration.

For our binary search, we implement a helper method that just reads the date of a record without building the complete `DayLog` data structure:

```
int getDate (int index) throws RecordStoreException {
    byte[] buf = days.getRecord (index);
    return ((((int) buf [0]) & 0x0ff) << 24)
        | ((((int) buf [1]) & 0x0ff) << 16)
        | ((((int) buf [2]) & 0x0ff) << 8)
        | ((((int) buf [3]) & 0x0ff));
}
```

The `getIndex()` method performs the binary search, returning the index of the day log of the given date. If a day log for the given date does not exist, the negative index where the day log for the given date should be inserted is returned:

```
public int getIndex (int date) throws RecordStoreException {

    int i = 1;
    int j = days.getNumRecords();
    int k;
    int dateK;

    while (i <= j) {
        k = (i + j) / 2;
        dateK = getDate (k);
        if (date == dateK) return k;
        else if (date > dateK) i = k+1;
        else j = k-1;
    }

    return -i;
}
```

The method for reading a `DayLog` of a given date is quite simple. First, the index for the given date is calculated by calling `getIndex()`. Then, whether the index is negative or not, a new `DayLog` is created or loaded from the record store:

```
public DayLog getDayLog (int date)
        throws RecordStoreException, IOException  {
    int index = getIndex (date);

    return (index == -1)
        ? new DayLog (date)
            : new DayLog (days.getRecord (index));
}
```

Storing a `DayLog` is a bit more complicated. If an entry for the given date does not yet exist, all following records need to be shifted in order to obtain space for the new record:

```
    public void storeDayLog (DayLog dayLog)
            throws RecordStoreException, IOException {
        if (!dayLog.isDirty()) return;

        int index = getIndex (dayLog.getDate());

        byte [] target = dayLog.getByteArray();

        if (index < 0) {
            index = -index;

            int num = days.getNumRecords();
```

```
            if (index > num) {
                days.addRecord (target, 0, target.length);
                return;
            }

            byte [] buf = days.getRecord (num);
            days.addRecord (buf, 0, buf.length);

            for (int i = num; i > index; i++) {
                buf = days.getRecord (i-1);
                days.setRecord (i, buf, 0, buf.length);
            }
        }

        days.setRecord (index, target, 0, target.length);
    }
}
```

We also need a method for closing the record store. However, this is quite straightforward:

```
public void close() throws RecordStoreException {
   days.closeRecordStore();
}
```

Finally, we add a static method for converting Java `Date` objects to our internal format. For this purpose, we create a calendar, set it to the given date, and then read the year, month, and day of month fields:

```
public static int dateToInt (Date date) {
    Calendar c = Calendar.getInstance();
    c.setTime (date);
    return (c.get (Calendar.YEAR) << 16)
        | ((c.get (Calendar.MONTH)-Calendar.JANUARY+1) << 8)
        | (c.get (Calendar.DAY_OF_MONTH));
}
```

The User Interface

At this stage, we have finished the classes needed for data management. Using these classes, it is an easy task to create a platform-dependent user interface. We need to create user interfaces for MIDP and PDAP. The core of both application interfaces is the chart that is used to display the daily logs.

In the MIDP user interface, the chart class is realized using the low-level `javax.microedition.lcdui.Canvas`. The corresponding PDAP class is derived from `java.awt.Component`, the PDAP base class for components.

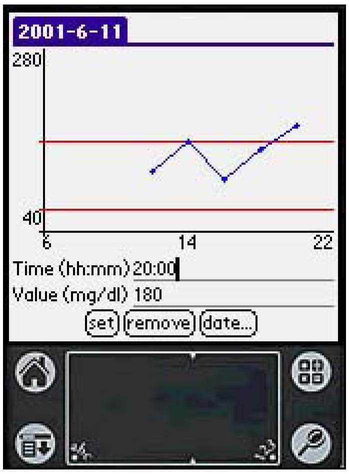
In both cases, the first step is to create the coordinate system and to draw borderlines for critical values.

To draw the actual data points on the chart, we can use the `DayLog` methods `getXPoints()` and `getYPoints()` previously implemented for that purpose.

The remaining UI components for both platforms are quite straightforward. For a general discussion of the MIDP UI components, refer to Chapter 3, "MIDP Programming." PDAP UI components are described in Chapter 4, "PDAP Programming."

The user interfaces for MIDP and PDAP are shown in Listings 9.1 and 9.2. In addition, Figure 9.3 shows a `BloodSugarPDAP` screenshot.

Figure 9.3. The running `BloodSugarPdap` application showing the logged blood sugar values of the current day.



Listing 9.1 `BloodSugarMidp.java`—The `BloodSugarMidp` MIDlet

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


import java.util.*;
import java.io.*;

public class BloodSugarMidp extends MIDlet implements CommandListener
{

    LogStorage log;
    DayLog dayLog;
    Display display;

    Chart chart;
    ValueForm valueForm;
    DateForm dateForm;

    class Chart extends Canvas {

        public void paint (Graphics g) {
            g.setColor (0x00FFFFFF);
            g.fillRect (0, 0, getWidth(), getHeight());
            g.setColor (0x00000000);

            int charWidth = g.getFont().charWidth ('0');
            int xSpace = 3*charWidth;
            int ySpace = g.getFont().getHeight();

            int h = getHeight() - 2*ySpace;
            int w = getWidth() - xSpace;

            String title = dayLog.getTitle();
            int tw = g.getFont().stringWidth (title);
```

```
            g.drawString (title, getWidth() / 2 - tw / 2, 0,
g.TOP|g.LEFT);

            g.drawString (""+DayLog.MAX_VALUE, 0, ySpace,
g.TOP|g.LEFT);
            g.drawString (""+DayLog.MIN_VALUE, charWidth, h,
g.TOP|g.LEFT);

            g.drawRect (xSpace, ySpace, w-1, h);

            g.translate (xSpace, h);
            g.drawString (""+(DayLog.MIN_TIME/60), 0, ySpace,
g.TOP|g.LEFT);
            g.drawString (""+(DayLog.MAX_TIME+DayLog.MIN_TIME)/120,
                              w/2-charWidth, ySpace, g.TOP|g.LEFT);
            g.drawString (""+(DayLog.MAX_TIME/60),
                              w-2*charWidth, ySpace, g.TOP|g.LEFT);

            int mnb = DayLog.getY (DayLog.MIN_BORDER, h);
            int mxb = DayLog.getY (DayLog.MAX_BORDER, h);

            g.setColor (0x00FF0000);
            g.drawLine (0, mnb, w + 2, mnb);
            g.drawLine (0, mxb, w + 2, mxb);
            g.setColor (0x000000FF);


            int[] x = dayLog.getXPoints (w);
            int[] y = dayLog.getYPoints (h);


            int count = dayLog.getCount();

            if (count == 1) {
                    g.drawArc (x[0] - 2, y[0] - 2, 4, 4, 0, 360);
            }
            else {
                for (int i = 0; i < count - 1; i++) {
                    g.drawLine(x[i], y[i],
                               x[i+1], y[i+1]);

                    g.drawArc (x[i] -2 , y[i] - 2, 4, 4, 0, 360);
                    g.drawArc (x[i+1] -2 , y[i+1] - 2, 4, 4, 0, 360);
                }
            }
        }
    }

    class DateForm extends Form {

        DateField dateField = new DateField ("Log date:",
DateField.DATE);

        public DateForm (CommandListener cmdListener) {
            super ("Select");
            append (dateField);
            addCommand (okCommand);
            setCommandListener(cmdListener);
        }
        public Date getDate() {
            return dateField.getDate();
```

```
        }
    }

    class ValueForm extends Form {
        TextField measuredMgDl =
            new TextField ("mg/dl", "", 3, TextField.NUMERIC);
        TextField measurementTime =
            new TextField ("Measured at:", "", 5, TextField.ANY);

        public ValueForm (CommandListener cmdListener) {
            super ("Enter Value");
            append (measuredMgDl);
            append (measurementTime);
            addCommand (okCommand);
            setCommandListener(cmdListener);
        }

        public String getValue() {
            return measuredMgDl.getString();
        }

        public int getTime() {
            return DayLog.parseTime (measurementTime.getString());
        }
    }

    static final Command okCommand = new Command ("OK",
Command.SCREEN, 1);
    static final Command exitCommand = new Command ("Exit",
Command.SCREEN, 1);
    static final Command setCommand = new Command ("Set",
Command.SCREEN, 2);
    static final Command removeCommand =
        new Command ("Remove", Command.SCREEN, 2);
    static final Command dateCommand =
        new Command ("Date...", Command.SCREEN, 2);

    public BloodSugarMidp() throws RecordStoreException, IOException
{
        log = new LogStorage();
        dayLog = log.getDayLog (LogStorage.dateToInt (new Date()));
        valueForm = new ValueForm (this);
        dateForm = new DateForm (this);
        chart = new Chart();

        chart.addCommand (exitCommand);
        chart.addCommand (setCommand);
        chart.addCommand (removeCommand);
        chart.addCommand (dateCommand);

        chart.setCommandListener(this);
    }

    public void startApp() {
        display = Display.getDisplay (this);
        display.setCurrent (chart);
```

```
    }

    public void pauseApp() {
    }

    public void destroyApp (boolean unconditional) {
    }

    public void commandAction (Command c, Displayable d) {

        if (c == exitCommand) {
            notifyDestroyed();
        }
        else if (c == setCommand) {
            display.setCurrent (valueForm);
        }
        else if (c == dateCommand) {
            display.setCurrent (dateForm);
        }
        else {
            try {
                if (c == okCommand && d == valueForm) {
                    dayLog.set (valueForm.getTime(),
                                Integer.parseInt
(valueForm.getValue()));
                    log.storeDayLog (dayLog);
                    display.setCurrent (chart);
                    chart.repaint();
                }
                else if (c == okCommand && d == dateForm) {
                    dayLog = log.getDayLog (LogStorage.dateToInt
                                            (dateForm.getDate()));
                    display.setCurrent (chart);
                }
                else if (c == removeCommand) {
                    dayLog.remove (valueForm.getTime());
                    log.storeDayLog (dayLog);
                }
            }
            catch (Exception e) {
                Alert inputError = new Alert ("Invalid Value:");
                inputError.setString (e.toString());
                display.setCurrent (inputError);
            }
        }
    }
}
```

Listing 9.2 `BloodSugarPdap.java`—The `BloodSugarPdap` Application

```
import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;

import javax.microedition.midlet.*;
import javax.microedition.rms.*;

public class BloodSugarPdap extends MIDlet implements ActionListener
{
```

```
    public class ErrorDialog extends Dialog implements ActionListener
{

        private Button okButton = new Button ("Ok");

        public ErrorDialog(String error) {
            super (frame, "Error", true);
            add("Center", new Label (error));
            Panel buttonPanel = new Panel();
            buttonPanel.add(okButton);
            okButton.addActionListener(this);
            add("South", buttonPanel);

            addWindowListener(new WindowAdapter() {
                public void windowClosing (WindowEvent ev) {
                    setVisible (false);
                    dispose();
                }
            });

            pack();
        }

        public void actionPerformed (ActionEvent e) {
            setVisible (false);
            dispose();
        }
    }

    Frame frame = new Frame();
    LogStorage log;
    DayLog dayLog;

    Button setButton = new Button ("set");
    Button removeButton = new Button ("remove");
    Button dateButton = new Button ("date...");

    TextField timeField = new TextField();
    TextField valueField = new TextField();

    Chart chart = new Chart();

    class Chart extends Canvas {

        public void paint (Graphics g) {

            FontMetrics fm = g.getFontMetrics();
            Dimension size = getSize();

            int cw = fm.stringWidth ("0");
            int lw = 3*cw;
            int lh = fm.getHeight();
            int w = size.width - lw;
            int h = size.height - lh;

            g.drawString (""+DayLog.MAX_VALUE, 0, fm.getAscent());
            g.drawString (""+DayLog.MIN_VALUE, cw, h -
fm.getDescent());

            g.translate (lw, h);
```

```
            g.drawString (""+(DayLog.MIN_TIME/60), 0, fm.getAscent());
            g.drawString (""+(DayLog.MAX_TIME+DayLog.MIN_TIME)/120,
                          w/2-cw, fm.getAscent());
            g.drawString (""+(DayLog.MAX_TIME/60), w-2*cw,
fm.getAscent());

            g.drawLine (-2, 0, w + 2, 0);
            g.drawLine (0, 2, 0, - (h + 2));

            int mnb = DayLog.getY (DayLog.MIN_BORDER, h);
            int mxb = DayLog.getY (DayLog.MAX_BORDER, h);

            g.setColor (Color.red);
            g.drawLine (-2, mnb, w + 2, mnb);
            g.drawLine (-2, mxb, w + 2, mxb);

            g.setColor (Color.blue);

            int[] x = dayLog.getXPoints (w);
            int[] y = dayLog.getYPoints (h);

            for (int i = 0; i < dayLog.getCount(); i++)
                g.drawOval (x [i]-1, y[i]-1, 2, 2);

            g.drawPolyline (x, y, dayLog.getCount());
        }

        public Dimension getPreferredSize() {
            return new Dimension (200, 200);
        }
    }



    public BloodSugarPdap() throws RecordStoreException, IOException
{
        log = new LogStorage();
        dayLog = log.getDayLog (LogStorage.dateToInt (new Date()));
        frame.setTitle (dayLog.getTitle());
        Panel inputPane = new Panel (new BorderLayout());

        Panel labelPane = new Panel (new GridLayout (0,1));
        labelPane.add(new Label ("Time (hh:mm)"));
        labelPane.add(new Label ("Value (mg/dl)"));
        inputPane.add("West", labelPane);

        Panel fieldPane = new Panel (new GridLayout (0,1));
        fieldPane.add(timeField);
        fieldPane.add(valueField);
        inputPane.add("Center", fieldPane);

        Panel buttonPane = new Panel();
        buttonPane.add(setButton);
        setButton.addActionListener(this);
        buttonPane.add(removeButton);
        removeButton.addActionListener(this);
        buttonPane.add(dateButton);
        dateButton.addActionListener(this);
        inputPane.add("South", buttonPane);
```

```
        frame.add("Center", chart);
        frame.add("South", inputPane);
        frame.pack();

        frame.addWindowListener(new WindowAdapter() {
                public void windowClosing (WindowEvent ev) {
                    frame.dispose();
                    notifyDestroyed();
                }
            });
    }

    public void startApp() {
        frame.show();
    }


    public void pauseApp() {
    }
    public void destroyApp (boolean conditional) {
        frame.dispose();
    }


    public void actionPerformed (ActionEvent action) {
        try {
            if (action.getSource() == setButton) {
                dayLog.set (DayLog.parseTime (timeField.getText()),
                            Integer.parseInt (valueField.getText()));

                log.storeDayLog (dayLog);
            }
            else if (action.getSource() == removeButton) {
                dayLog.remove (DayLog.parseTime
(timeField.getText()));
                log.storeDayLog (dayLog);
            }
            else if (action.getSource() == dateButton) {
                dayLog = log.getDayLog
                    (LogStorage.dateToInt
                     (new DateDialog (frame).getDate (new Date())));

                    frame.setTitle (dayLog.getTitle());
            }
        }
        catch (Exception e) {
            new ErrorDialog (e.toString()).show();
        }

        chart.repaint();
    }
}
```

The source codes of the classes `CalendarComponent` and `CalendarCanvas` shown in Listings 9.3, 9.4 and 9.5 contain helper components providing a date dialog and the correspondig helper classes.

Listing 9.3 `DateDialog.java`—The `DateDialog` Class Source Code

```
import java.util.*;
import java.awt.*;
```

```
import java.awt.event.*;

public class DateDialog extends Dialog implements ActionListener {
    CalendarComponent calendarComponent = new CalendarComponent();
    Button okButton = new Button ("Ok");
    Button cancelButton = new Button ("Cancel");
    boolean ok;


    public DateDialog (Frame owner) {
        super (owner, "Set Date", true);
        Panel buttons = new Panel();
        buttons.add(okButton);
        buttons.add(cancelButton);
        okButton.addActionListener(this);
        cancelButton.addActionListener(this);
        add("Center", calendarComponent);
        add("South", buttons);

        addWindowListener(new WindowAdapter() {
                public void windowClosing (WindowEvent e) {
                    setVisible (false);
                }
            });

        pack();
    }


    public Date getDate (Date d) {
        show();
        return ok ? calendarComponent.getDate() : d;
    }


    public void actionPerformed (ActionEvent e) {
        ok = e.getSource() == okButton;
        setVisible (false);
    }
}
```

Listing 9.4 `CalendarComponent.java`—The `CalendarComponent` Class Source Code

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class CalendarComponent extends Panel implements
ActionListener, ItemListener {

    Calendar calendar =  Calendar.getInstance();
    Choice monthChoice = new Choice();
    Label yearLabel = new Label ("9999");
    Button buttonPlus = new Button ("+");
    Button buttonMinus = new Button ("-");
    CalendarCanvas calendarCanvas = new CalendarCanvas();

    public CalendarComponent() {

        super (new BorderLayout());

        monthChoice.add("Jan");
```

```
        monthChoice.add("Feb");
        monthChoice.add("Mar");
        monthChoice.add("Apr");
        monthChoice.add("May");
        monthChoice.add("Jun");
        monthChoice.add("Jul");
        monthChoice.add("Aug");
        monthChoice.add("Sep");
        monthChoice.add("Oct");
        monthChoice.add("Nov");
        monthChoice.add("Dec");
        monthChoice.addItemListener(this);

        Panel top = new Panel();
        top.add(monthChoice);
        top.add(yearLabel);
        top.add(buttonMinus);
        top.add(buttonPlus);
        buttonPlus.addActionListener(this);
        buttonMinus.addActionListener(this);

        add("North", top);
        add("Center", calendarCanvas);

        propagate();
    }
    public void setDate (Date date) {
        calendar.setTime (date);
        propagate();
    }


    void propagate() {
        calendar.set
            (Calendar.DAY_OF_MONTH,
             calendarCanvas.getCalendar().get
             (Calendar.DAY_OF_MONTH));

        calendarCanvas.setDate (calendar.getTime());

        yearLabel.setText (""+calendar.get(Calendar.YEAR));

        monthChoice.select
            (calendar.get (Calendar.MONTH)-Calendar.JANUARY);
        repaint();
    }


    public Date getDate() {

        calendar.set (Calendar.DAY_OF_MONTH,
                      calendarCanvas.getCalendar().get
                      (Calendar.DAY_OF_MONTH));

        return calendar.getTime();
    }



    public void itemStateChanged (ItemEvent ev) {
```

```
        calendar.set (Calendar.MONTH,
                      monthChoice.getSelectedIndex() +
Calendar.JANUARY);

        propagate();
    }


    public void actionPerformed (ActionEvent ev) {
        if (ev.getSource() == buttonPlus)
            calendar.set (Calendar.YEAR, calendar.get (Calendar.YEAR)
+ 1);
        else if (ev.getSource() == buttonMinus)
            calendar.set (Calendar.YEAR, calendar.get (Calendar.YEAR)
- 1);

        propagate();
    }
}
```

Listing 9.5 `CalendarCanvas.java`—The `CalendarCanvas` Class Source Code

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class CalendarCanvas extends Component implements
MouseListener {

    Calendar calendar = Calendar.getInstance();

    private static String[] days
        = {"Su", "Mo", "Tu", "We", "Th", "Fr", "Sa"} ;

    private static int daysInMonth[]
        = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31} ;


    int cellWidth;
    int cellHeight;
    int totalWidth;
    int totalHeight;
    int left;
    int top;
    int firstDay;


    public CalendarCanvas() {
        addMouseListener(this);
    }


    public void paint(Graphics g) {

        cellWidth = (getWidth() - 1) / 7;
        cellHeight = (getHeight() - 1) / 7;
        totalWidth = 7 * cellWidth + 1;
        totalHeight = 7 * cellHeight + 1;

        left = (getWidth() - totalWidth) / 2;
        top = (getHeight() - totalHeight) / 2;
```

```
        g.translate (left, top);

        for (int i = 0; i < 8; i++)
            g.drawLine
                (cellWidth * i, cellHeight,
                 cellWidth * i, 7 * cellHeight - 1);

        for (int i = 1; i < 8; i++)
            g.drawLine
                (0, cellHeight * i,
                 totalWidth - 1, cellHeight * i);

        FontMetrics font = g.getFontMetrics();
        int textHeight = font.getHeight();
        int ascent = font.getAscent();

        for (int i = 0; i < 7; i++) {
            String name = days [i];
            int textWidth = font.stringWidth(name);
            g.drawString
                (name,  cellWidth * i + (cellWidth - textWidth) / 2,
                 cellHeight - 1);
        }


        int day = calendar.get (Calendar.DAY_OF_MONTH);
        calendar.set(Calendar.DAY_OF_MONTH, 1);
        firstDay = calendar.get(Calendar.DAY_OF_WEEK) - 1;
        calendar.set(Calendar.DAY_OF_MONTH, day);

        int dayOfWeek = firstDay;

        int days = daysInMonth (calendar);
        int weekOfMonth = 1;

        for (int i = 1; i <= days; i++) {
            String s = String.valueOf(i);
            int textWidth = font.stringWidth(s);
            int cellLeft = cellWidth * dayOfWeek;
            int cellTop = cellHeight * weekOfMonth;

            if (i == day) {
                g.fillRect (cellLeft, cellTop,
                            cellWidth, cellHeight);
                g.setColor (Color.white);
            }

            g.drawString (s,
                          cellLeft + (cellWidth - textWidth) / 2,
                          cellTop + ascent + (cellHeight - textHeight)
/ 2);

            g.setColor (Color.black);

            dayOfWeek++;
            if (dayOfWeek == 7) {
                weekOfMonth++;
                dayOfWeek = 0;
            }
        }
```

```
    }


    public static int daysInMonth (Calendar calendar) {
        int year = calendar.get (Calendar.YEAR);
        int month = calendar.get (Calendar.MONTH);
        int days = daysInMonth [month-Calendar.JANUARY];
        if ((month == Calendar.FEBRUARY) && (year % 4 == 0)
            && !(year % 100 == 0) || (year % 400 == 0))
            days++;

        return days;
    }


    public void setDate (Date date) {
        calendar.setTime (date);
        repaint();
    }
    public Calendar getCalendar() {
        return calendar;
    }


    public Date getDate() {
        return calendar.getTime();
    }


    public Dimension getMinimumSize() {
      return getPreferredSize();
    }


    public Dimension getPreferredSize() {

        FontMetrics fm = getFontMetrics (getFont());

        return new Dimension ((fm.stringWidth ("88") + 1) * 7 + 1,
                              (fm.getHeight() + 1) * 7 + 1);
    }


    public void mouseEntered (MouseEvent ev) {}
    public void mouseExited (MouseEvent ev) {}
    public void mousePressed (MouseEvent ev) {}
    public void mouseReleased (MouseEvent ev) {}

    public void mouseClicked (MouseEvent ev) {
        int x = (ev.getX() - left) / cellWidth;
        int y = (ev.getY() - top) / cellHeight - 1;

        int index = x + 7 * y - firstDay + 1;

        if (index > 0 && index <= daysInMonth (calendar))
            calendar.set (Calendar.DAY_OF_MONTH, index);

        repaint();
    }

}
```

Summary

In this chapter, you learned how to create an advanced application using most of the APIs that are available in MIDP and PDAP. You have learned to split the functionality of one application into a user interface independent base part in order to use it in an MIDP and PDAP implementation. Finally, you have learned to create a user interface for the Blood Sugar Logger in order to run it on MIDP and the PDAP devices.

Chapter 10. Third-Party Libraries

IN THIS CHAPTER

- XML
- Simple Object Access Protocol: SOAP
- MathFP
- The Bouncy Castle Crypto API
- User Interface Extensions

At this point we have discussed the whole CLDC API, including all core packages and the profile-specific extensions of MIDP and PDAP. You might have noticed that some important APIs aren't provided in CLDC. However, for several purposes such as parsing XML or fixed-point integer arithmetic, third-party libraries are available for CLDC.

In this chapter, we will discuss a few important libraries that can be downloaded from the Internet. In most cases, we will demonstrate their usage by creating a small sample application.

XML

Currently, three different libraries are available for XML parsing. Table 10.1 shows an overview of the features and limitations of the different APIs. In general, a trade-off exists between the package size and the features available. None of the XML parsers available for CLDC is a validating XML parser.

Table 10.1. XML Parsers Available for CLDC

Parser	***JAR Size***	***License***	***URL and Remarks***
NanoXML KVM port	9.7KB	Libpng	http://nanoxml.sourceforge.net No support for mixed content. The whole document is parsed to a memory structure. Optional SAX interface available. XML writing support.
TinyXML KVM port	7.9KB	GPL	http://www.microjava.com SAX-like callback interface. No XML writing support.
KXML	19.9KB	Enhydra License	http://www.kxml.org Namespace support. Optional WBXML/WML support. Optional kDOM support. XML writing support.

NanoXML

NanoXML is a very small XML parser developed by Marc De Scheemaecker. Eric Giguere has provided a CLDC port of version 1.6.4 of the parser. The parser parses the whole XML document to a memory structure that is accessible via the NanoXML API. The advantage of this approach is that access to the document content is extremely simple. The trade-off is that the device needs enough heap memory to hold the whole document structure. It isn't possible to access the XML document until it has been read completely, so progressive display isn't possible.

The NanoXML API consists of only two classes: `kXMLElement` and `XMLParseException` (an exception class). An XML document is parsed by creating an `kXMLElement` and then invoking `parseFromReader()` with a reader as parameter. A `kXMLElement` object can be written to a writer or a string using the `write()` method. Further important methods are `enumerateChildren()`, which returns an `Enumeration` for iterating through all child elements, and `getContents()`, which returns the text content of the given `kXMLElement`. For complete NanoXML reference, refer to the NanoXML JavaDOC, available at http://nanoxml.sourceforge.net.

As an illustrative example for parsing XML, we will use the NewsForge XML format. NewsForge is a news Web site with a focus on open source development, and it provides a URL where you can download the current content in XML format. The NewsForge XML code consists of a `<backslash>` element containing a number of `<story>` elements. The child elements of `<story>` contain information about the story such as title, the author, and possibly a short description. Listing 10.1 shows an example of the NewsForge XML code. If you would like to run the demo but cannot access NewsForge, you can use the example XML code for testing. For more details about NewsForge, refer to http://www.newsforge.com.

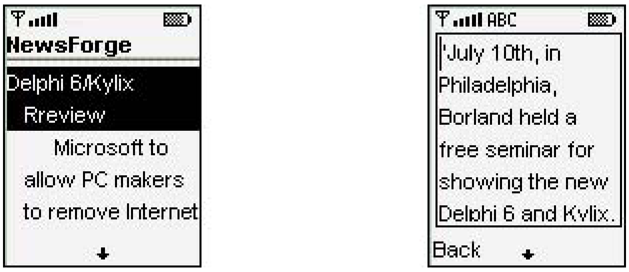
Listing 10.1 `newsforge.xml`—NewsForge XML Code Example

```
<?xml version="1.0" encoding="iso-8859-1"?>
<backslash xmlns:backslash="http://www.newsforge.com/backslash.dtd">
  <story>
        <title>Linux 2.4.6-ac2</title>

<url>http://www.newsforge.com/article.pl?sid=01/07/07/169250</url>
        <time>2001-07-07 16:08:37</time>
        <author>tina</author>
        <department></department>
        <topic>gnulinux</topic>
        <comments>0</comments>
        <section>newsvac</section>
        <image>topicnews.gif</image>
        <description>&quot;Drop out various bits that
        are 2.5 stuff...&quot;</description>
  </story>
  <story>
        <title>Millions are shut out of Microsoft's
        instant-messaging service</title>

<url>http://www.newsforge.com/article.pl?sid=01/07/07/0854238</url>
        <time>2001-07-07 12:04:23</time>
        <author>cdlu</author>
        <department></department>
        <topic>closedsrc</topic>
        <comments>0</comments>
        <section>newsvac</section>
        <image>topicnews.gif</image>
        <description>The Seattle Post-Intelligencer reports that
Microsoft's
        instant-messaging service has been inaccessible
```

```
        to a third of its users for
        three days. </description>
  </story>
</backslash>
```

Listing 10.2 contains the code of the NanoNewsreader example MIDlet. When the MIDlet is started, it reads the XML code from NewsForge and displays a list of the story titles included. When a title is selected from the list, the description of the story is displayed in a TextBox, if available, otherwise only the title is displayed. The interesting methods are readNews() and readStory(). readNews() connects to the server using an HttpConnection and constructs a kXMLElement. Then it opens a reader on the InputStream provided by the connection and passes it to the parseFromReader() method of the kXMLElement. Now the root element is parsed to the given kXMLElement. The method enumerateChildren() is used to enumerate all <story> child elements of the <backslash> root element. For each child, the readStory() method is called. readStory() again iterates through the child elements, now those of the <story> element. The content of the <title> and <description> elements is extracted using the getContents() method of kXMLElement. Figure 10.1 shows the NanoNewsreader.

Figure 10.1. The NanoNewsreader MIDlet.



Listing 10.2 NanoNewsreader.java—NanoXML Newsreader Example

```
import java.io.*;
import java.util.Vector;
import java.util.Enumeration;

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

import nanoxml.*;

public class NanoNewsreader extends MIDlet implements CommandListener
{

    static final String URL = "http://newsforge.com/newsforge.xml";
    static final String TITLE = "NewsForge";

    Vector descriptions = new Vector();
    List newsList = new List (TITLE, Choice.IMPLICIT);
    TextBox textBox = new TextBox ("", "", 256, TextField.ANY);
    Display display;

    Command backCmd = new Command ("Back", Command.BACK, 0);

    public void startApp() {
        display = Display.getDisplay (this);
        display.setCurrent (newsList);
        newsList.setCommandListener(this);
```

```
        textBox.setCommandListener(this);
        textBox.addCommand (backCmd);
        if (descriptions.size() == 0)
            readNews();
    }


    public void readNews() {
        try {
            HttpConnection httpConnection =
                (HttpConnection) Connector.open (URL);

            kXMLElement xml = new kXMLElement();
            xml.parseFromReader (new InputStreamReader
                (httpConnection.openInputStream()));

            for (Enumeration e = xml.enumerateChildren();
                 e.hasMoreElements();)
                readStory ((kXMLElement) e.nextElement());
        }
        catch (IOException e) {
            newsList.append ("Error", null);
            descriptions.addElement (e.toString());
        }
    }


    /** Read a story and append it to the list */

    public void readStory (kXMLElement story) {
        String title = null;
        String description = null;
        for (Enumeration e = story.enumerateChildren();
             e.hasMoreElements(); ) {
            kXMLElement field = (kXMLElement) e.nextElement();

            if (field.getTagName().equals ("title"))
                title = field.getContents();
            else if (field.getTagName().equals ("description"))
                description = field.getContents();
}

            if (title !=null){
                descriptions.addElement
(description !=null ?description :title);
                newsList.append (title,null);
            }
}


    public void pauseApp() {
    }


    public void commandAction (Command c, Displayable d) {

        if (c == List.SELECT_COMMAND) {

            String text = (String) descriptions.elementAt
                (newsList.getSelectedIndex());
```

```
            if (textBox.getMaxSize() < text.length())
                textBox.setMaxSize (text.length());

            textBox.setString (text);
            display.setCurrent (textBox);
        }
        else if (c == backCmd)
            display.setCurrent (newsList);
    }

    public void destroyApp (boolean really) {
    }
}
```

TinyXML

TinyXML was designed by Tom Gibara to be used in applets and other situations in which code size is important. TinyXML has been ported to CLDC by Christian Sauer. Detailed information about the port is available at http://www.microjava.com/news/techtalk/tinyxml. In contrast to NanoXML, TinyXML provides a callback interface similar to the *Simple Access Interface to XML (SAX)*, the de facto Java parsing standard. In order to parse an XML document, you need to create an `XMLParser` object, register a `DocumentHandler` using the `setDocumentHandler()` method, and assign an `XMLInputStream` using `setInputStream()`. The `DocumentHandler` provides the callback interface mentioned earlier. It contains methods such as `elementStart()`, `charData()`, and `elementEnd()`, which are called when the parser encounters the corresponding XML code.

You will again use Newsforge XML to build an example application (see Listing 10.3). Unfortunately, you cannot just use any `InputStream` or `Reader` with TinyXML; you must provide an `XMLInputStream` that is constructed from a `String`. Thus, you need to build a string from the `InputStream` obtained from the `HttpConnection` before you can invoke the parser.

Listing 10.3 `TinyNewsreader.java`—TinyXML Newsreader Example

```
import java.io.*;
import java.util.Vector;
import java.util.Hashtable;

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


public class TinyNewsreader extends MIDlet implements CommandListener
{

    static final String URL = "http://newsforge.com/newsforge.xml";
    static final String TITLE = "NewsForge";
    Vector descriptions = new Vector();
    List newsList = new List (TITLE, Choice.IMPLICIT);
    TextBox textBox = new TextBox ("", "", 256, TextField.ANY);
    Display display;

    Command backCmd = new Command ("Back", Command.BACK, 0);

    class NewsHandler extends HandlerBase {
        StringBuffer title;
        StringBuffer description;
        String currentElement = "";
```

```
        public void elementStart (String name, Hashtable attributes)
{
            currentElement = name;
            if (name.equals ("story")) {
                title = new StringBuffer();
                description = new StringBuffer();
            }

        }

        public void charData (String s) {
            if (currentElement.equals ("title"))
                title.append (s);
            else if (currentElement.equals ("description"))
                description.append (s);
}

        public void elementEnd (String name){
            currentElement ="";
            if (name.equals ("story ")){
                newsList.append (title.toString(),null);
                String d =description.toString();
                descriptions.addElement
(d.length()==0 ?title.toString():d);
            }
}


    public void readNews() {
        try {
            HttpConnection httpConnection =
                (HttpConnection) Connector.open (URL);
            InputStream is = httpConnection.openInputStream();

            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            byte [] buf = new byte [256];

            while (true) {
                int cnt = is.read (buf, 0, 256);
                if (cnt < 0) break;
                bos.write (buf, 0, cnt);
            }

            XMLParser parser = new XMLParser();
            parser.setInputStream
                (new XMLInputStream (new String (bos.toByteArray())));
            parser.setDocumentHandler (new NewsHandler());
            parser.parse();
        }
        catch (ParseException e) {
            newsList.append ("Error", null);
            descriptions.addElement (e.toString());
        }
        catch (IOException e) {
            newsList.append ("Error", null);
            descriptions.addElement (e.toString());
        }
    }

    public void startApp() {
```

```
        display = Display.getDisplay (this);
        display.setCurrent (newsList);
        newsList.setCommandListener(this);
        textBox.setCommandListener(this);
        textBox.addCommand (backCmd);
        if (descriptions.size() == 0) readNews();
    }

    public void pauseApp() {
    }

    public void commandAction (Command c, Displayable d) {

        if (c == List.SELECT_COMMAND) {

            String text = (String) descriptions.elementAt
                (newsList.getSelectedIndex());

            if (textBox.getMaxSize() < text.length())
                textBox.setMaxSize (text.length());

            textBox.setString (text);
            display.setCurrent (textBox);
        }
        else if (c == backCmd)
            display.setCurrent (newsList);
    }

    public void destroyApp (boolean really) {
    }
}
```

The most interesting part of the example is the inner class `NewsHandler`. Instead of implementing the full `DocumentHandler` interface, you derive your class from the convenience class `HandlerBase`, which provides empty implementations for all `DocumentHandler` methods. Thus, you only need to implement the callback methods for events you are actually interested in. For parsers with a callback (push) interface, you need some representation of the current state in order to know what to do with the incoming events. For that purpose, you save the name of the element in `currentElement` when the start of an element is indicated by a call to `elementStart()`. If a new `<story>` element starts, the title and description buffer variables are initialized. When a text event is received, it is ignored or appended to the title or description, depending on the current element.

Note that the implemented handling of the `currentElement` name isn't sufficient for nested structures; it works only for elements that have no further sub-elements. However, in this case it is sufficient. Finally, when an element end is detected and the name of the element is *story*, the title and description are appended to the corresponding lists.

kXML

kXML is larger than NanoXML and TinyXML, but it is the only XML parser for CLDC that provides XML namespace support. In addition, optional kDOM and WBXML packages are available for kXML.

In contrast to the other parsers, kXML is a *pull-based* parser. The motivation for pull parsers is to provide an easier handling mechanism than a centralized callback interface without needing to build an explicit XML memory structure. Push parsers such as TinyXML; push all XML events to a few centralized callback methods. Inside the callback methods, the application needs to look up its internal state before being able to handle the event correctly. For the Newsforge XML example, it is relatively simple to keep track of the state; but for highly nested structures, doing so becomes quite a problem.

For that reason, many implementers prefer to have a complete object tree such as the NanoXML `kXMLElement` structure before actually processing an XML document.

A pull parser such as kXML works similar to a reader, where the application is in control of reading data. The advantage is that parsing can be performed in recursive functions, following the tree structure of the document. Instead of having an explicit global state object, the program state reflects the parsing state in a natural way.

The kXML pull parser is implemented in the `XmlParser` object. The next parse event can be queried with the `peek()` method or read with the `read()` method. Both methods return a `ParseEvent` object. `ParseEvent` provides access methods such as `getType()`, `getName()`, and `getText()`. The `getType()` method returns the type of the event—for example, `Xml.START_TAG`, `Xml.END_TAG`, or `Xml.TEXT`. The `getName()` method returns the name of the element if the event is an element start or element end event. The corresponding namespace can be obtained with `getNamespace()`. For text events, `getText()` returns the corresponding text strings. The parser also provides some convenience methods. For example, `XmlParser.skip()` skips over events that are ignorable in most cases, such as whitespace or comments. Special versions of `peek()` and `read()` also test for a certain type of event. For example, `peek(Xml.START_TAG, null, "story")` returns `true` if the next event is a start tag with the name *story* in any namespace.

As you did for the other two parsers, you will implement an MIDP client for the Newsforge XML service (see Listing 10.4). kXML accepts any reader as a source for the XML input, so you just need to wrap an `InputStreamReader` around the `InputStream` obtained from the `HttpConnection`. This feature enables you to begin processing the XML code while still receiving data. The advantage is that you can display the first news titles while still reading data from the server, reducing the time users need to wait before they can start reading. In order to use this feature, you just need to put the parsing process in a separate thread. Note that this approach would make sense for the TinyXML example, too, if the CLDC port would accept a regular `InputStream` for reading.

Note

This section handles kXML version 1.x. Version 2.0 will have a slightly different interface, similar to the XML parser available from kobjects.org. The kobjects parser is a pull parser as well, but it does not support XML namespaces.

Listing 10.4 `KxmlNewsreader.java`—kXML Newsreader Example

```
import java.io.*;
import java.util.Vector;

import org.kxml.*;
import org.kxml.parser.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;


public class KxmlNewsreader extends MIDlet implements CommandListener
{

    static final String URL = "http://newsforge.com/newsforge.xml";
    static final String TITLE = "NewsForge";

    Vector descriptions = new Vector();
    List newsList = new List (TITLE, Choice.IMPLICIT);
    TextBox textBox = new TextBox ("", "", 256, TextField.ANY);
```

```
    Display display;

    Command backCmd = new Command ("Back", Command.BACK, 0);

    class ReadThread extends Thread {

        public void run() {
            try {
                HttpConnection httpConnection =
                    (HttpConnection) Connector.open (URL);

                XmlParser parser = new XmlParser (new
InputStreamReader
                    (httpConnection.openInputStream()));

                parser.skip();
                parser.read (Xml.START_TAG, null, "backslash");

                while (true) {
                    parser.skip();

                    if (parser.peek (Xml.START_TAG, null, "story"))
                        readStory (parser);
                    else if (parser.peek (Xml.END_TAG, null,
"backslash")) {
                        parser.read();
                        break;
                    }
                    else throw new RuntimeException
                        ("XML error: Unexpected event: "+
parser.peek());
                }
            }
            catch (Exception e) {
                newsList.append ("Error", null);
                descriptions.addElement (e.toString());
            }
        }


        /** Read a story and append it to the list */

        void readStory (XmlParser parser) throws IOException {
            parser.read (Xml.START_TAG, "", "story");

            String title = null;
            String description = null;

            while (true) {
                parser.skip();
                ParseEvent event = parser.peek();

                if (event.getType() == Xml.START_TAG) {
                    String name = event.getName();
                    parser.read();
                    String text = parser.readText();
                    parser.read (Xml.END_TAG, "", name);

                    if (name.equals ("title"))
                        title = text;
                    else if (name.equals ("description"))
```

```
                        description = text;
                }
                else if (event.getType() == Xml.END_TAG) {
                    parser.read (Xml.END_TAG, "", "story");
                    break;
                }
                else throw new RuntimeException ("unexpected event:
"+event);
            }

}

                if (title !=null){
                    descriptions.addElement
(description !=null ?description
                    :title);
                    newsList.append (title,null);
        }
    }

    public void startApp() {
        if (display == null) {
            display = Display.getDisplay (this);
            newsList.setCommandListener(this);
            textBox.setCommandListener(this);
            textBox.addCommand (backCmd);
            new ReadThread().start();
        }
        display.setCurrent (newsList);
    }


    public void pauseApp() {
    }


    public void commandAction (Command c, Displayable d) {

        if (c == List.SELECT_COMMAND) {

            String text = (String) descriptions.elementAt
                (newsList.getSelectedIndex());

            if (textBox.getMaxSize() < text.length())
                textBox.setMaxSize (text.length());

            textBox.setString (text);
            display.setCurrent (textBox);
        }
        else if (c == backCmd)
            display.setCurrent (newsList);
    }


    public void destroyApp (boolean really) {
    }
}
```

In the example, the `run()` method of the `ReadThread` class connects to the server and creates an `XmlParser` object from the corresponding input stream. Then the XML processing instruction and whitespace at the beginning of the document are skipped using the `skip()` method. After the

`<backslash>` is read, the code enters the main loop for reading the `<story>` elements. Depending on the event type, the program descends into the `readStory()` method for reading a story or leaves the loop.

Simple Object Access Protocol: SOAP

The *Simple Object Access Protocol (SOAP)* is an XML-based protocol for remote method invocation. The SOAP protocol allows you to access Web services such as weather forecasts, stock quotes, or flight-booking information in a machine readable manner. Possible applications are clients adding value by combining different related services, or providing an user interface appropriate for mobile devices.

SOAP calls consist of a request sent from a client to a server and a response from the server. Listing 10.5 shows an example SOAP request, taken from the delayed stock quote example available at http://www.xmethods.org. The `<getQuote>` element in the body of the message represents the method to be called on the server, and the embedded symbol element contains the only method parameter: the name of the symbol to be queried. The corresponding server response example is shown in Listing 10.6. The `<getQuoteResponse>` element contains the return value in the `<return>` element.

Note

The XMethods Web site (http://www.xmethods.org) also provides a nice overview of SOAP resources available on the Net.

Listing 10.5 `SampleSoapRequest.xml`—Sample SOAP Stock Quote Request

```
<SOAP-ENV:Envelope
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <SOAP-ENV:Body>
    <ns1:getQuote
         xmlns:ns1="urn:xmethods-delayed-quotes"
         SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

        <symbol xsi:type="xsd:string">IBM</symbol>
    </ns1:getQuote>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 10.6 `SampleSoapResponse.xml`—Sample Server Response to the Stock Quote Request

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <SOAP-ENV:Body>
    <ns1:getQuoteResponse
```

```
        xmlns:ns1="urn:xmethods-delayed-quotes"
        SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:float">133.625</return>
    </ns1:getQuoteResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Serialization

In addition to an XML call envelope, the SOAP specification includes a simple XML-based data serialization format. The serialization formats standardized transmission of complex objects via SOAP remote method calls. It serializes the top-level object to an XML element corresponding to the classname, with embedded elements representing the properties. The following shows an example SOAP serialization of a `Person` class. The `nsp:` prefix is an XML namespace prefix, determining the namespace of the `Person` element. (XML namespaces can be compared to Java package names to some extent. For more details about XML namespaces, refer to the W3C recommendation found on http://www.w3.org/TR/1999/REC-xml-names-19990114/ or *to Sams Teach Yourself XML in 21 Days, Second Edition*.)

Java class:

```
class Person {
   String familyName;
   String givenName;
}
```

SOAP serialized instance:

```
<nsp:Person>
   <familyName>Turing</familyName>
   <givenName>Adam</givenName>
</nsp:Person>
```

The SOAP serialization format is able to serialize complex object graphs. Related objects can be serialized in a linked or embedded form. Single-referenced objects can be embedded in the referenced object, whereas multi-referenced objects should be linked. The following shows a simple example of a referenced object, serialized in both embedded and linked forms. Note that for embedded objects, there is no separate element for the classname, reducing the nesting depth and keeping the format human readable. However, for polymorphic properties, the classname must be contained in a `type` attribute of the corresponding element. In SOAP, the remote method call and the response are also serialized as if they were objects. For more details about SOAP, refer to Sams Publishing's *Understanding SOAP: The Authoritative Solution*. The specification is available at http://www.w3.org/TR/SOAP/.

Embedded objects:

```
<nsp:Person>
   <familyName>Douglas</familyName>
   <givenName>Mike</givenName>
   <father>
      <familyName>Douglas</familyName>
      <givenName>Kirk</givenName>
   </father>
</nsp:Person>
```

Linked objects:

```
<nsp:Person>
```

```
   <familyName>Turing</familyName>
   <givenName>Adam</givenName>
   <father href="#P02" />
</nsp:Person>

<nsp:Person>
   <familyName>Turing</familyName>
   <givenName>Adam</givenName>
   <father href="#P02" />
</nsp:Person>
```

Note

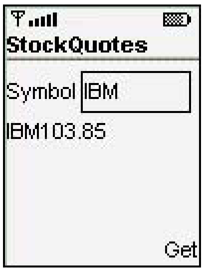
A simple alternative to SOAP is XML-RPC. An XML-RPC package for CLDC is recently available from http://kxmlrpc.enhydra.org.

kSOAP

The `kSOAP` API, available at http://www.ksoap.org, contains a kXML-based implementation of the SOAP protocol and serialization format for CLDC. Because CLDC doesn't provide reflection capabilities, existing classes must implement the interface `KvmSerializable` to add SOAP serialization capabilities. kSOAP already includes serialization support for primitive types and `Vector`s. `KvmSerializable` objects must be registered with a `ClassMap` object, providing a mapping between XML namespaces and names and Java classnames. For classes that are needed in the SOAP call only, you can use the convenience class `SoapObject`.

Listing 10.7 shows an example MIDlet for querying delayed stock quotes, corresponding to the request and response code given in Listings 10.5 and 10.6. The SOAP call is performed in the `commandAction()` method. First, the `SoapObject rpc` is created, modeling the method object `getQuote`. Then the `symbol` property is added. The symbol name is retrieved from the corresponding UI field. Finally, an `HttpTransport` object is created, and the remote method invocation is performed by the `call()` method with the `rpc` object as a parameter. The returned value is displayed in the `resultItem`. Figure 10.2 shows the `StockQuoteDemo` MIDlet.

Figure 10.2. The `StockQuoteDemo` MIDlet.



Listing 10.7 `StockQuoteDemo.java`—A Stock Quote MIDlet Using kSOAP

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

import org.ksoap.*;
import org.ksoap.transport.HttpTransport;
```

```
public class StockQuoteDemo extends MIDlet implements CommandListener
{

    Form mainForm = new Form ("StockQuotes");
    TextField symbolField = new TextField ("Symbol", "IBM", 5,
TextField.ANY);
    StringItem resultItem = new StringItem ("", "");
    Command getCommand = new Command ("Get", Command.SCREEN, 1);


    public StockQuoteDemo() {
        mainForm.append (symbolField);
        mainForm.append (resultItem);
        mainForm.addCommand (getCommand);
        mainForm.setCommandListener(this);
    }

    public void startApp() {
        Display.getDisplay (this).setCurrent (mainForm);
    }

    public void pauseApp() {
    }

    public void destroyApp (boolean unconditional) {
    }

public void commandAction (Command c, Displayable d) {
    try {
        // build request string

        String symbol = symbolField.getString();
        resultItem.setLabel (symbol);

        SoapObject rpc = new SoapObject
            ("urn:xmethods-delayed-quotes", "getQuote");

        rpc.addProperty ("symbol", symbol);

        HttpTransport transport = new HttpTransport
            ("http://services.xmethods.net:9090/soap",
             "urn:xmethods-delayed-quotes#getQuote");

        Object result = transport.call (rpc);
        resultItem.setText (""+result);

    }
    catch (Exception e) {
        resultItem.setLabel ("Error:");
        resultItem.setText (e.toString());
    }
}
```

MathFP

Because CLDC doesn't provide the primitive Java types `float` and `double` or the corresponding classes `Float` and `Double`, it seems that you cannot write a J2ME CLDC application that performs mathematical calculations beyond integer calculations. As a substitute for the missing floating-point arithmetic, Onno Hommes created a library called MathFP providing 32-bit fixed-point integer math functions. The MathFP library and the corresponding documentation can be downloaded at http://www.jscience.net.

To perform calculations in the MathFP format, you need to convert an integer or a string representing a floating-point value to the MathFP format. Those conversions are performed using the static methods of the class `MathFP` given in Table 10.4.

Table 10.4. Conversion Methods to the MathFP Format

Method Name	*Description*
`toFP(int l)`	Converts a normal Java `int` to a fixed-point integer.
`toFP(String s)`	Converts a string input to a fixed-point integer.

The following code snippet shows the conversion on two small examples:

```
// conversion of an integer to a MathFP integer
int mathFPInt1 = MathFP.toFP (1234);

// conversion of a String to a MathFP integer
int mathFPInt2 = MathFP.toFP("12.9881");
```

After successful conversion to the MathFP format, you are able to perform mathematical calculations using the methods supported in the MathFP library. The MathFP functionality ranges from simple addition, subtraction, multiplication, and division to powerful operations such as sine, cosine, and tangent of radian values.

To add or subtract two MathFP integers, you can use the methods `add()` and `sub()` or use the plus (`+`) or minus (`-`) operators directly:

```
// adding two MathFP integers using methods
int mathFPResult1 = MathFP.add(mathFPInt1, mathFPInt2);

// adding two MathFP integers using operators
int mathFPResult2 = mathFPInt1 + mathFPInt2;

// subtracting two MathFP integers using methods
int mathFPResult3 = MathFP.sub (mathFPInt1, mathFPInt2);

// subtracting two MathFP integers using operators
int mathFPResult4 = mathFPInt1 - mathFPInt2;
```

Simplifying the code by using the standard `int` operators is only possible for the `add()` and `sub()` methods. You can convert the result back to a Java integer or `String` type (for example, to display the result) using one of the three methods listed in Table 10.5.

Table 10.5. Conversion Methods from the MathFP Format Back to the Java Standard Types

Method Name	*Description*
`toInt(int x)`	Converts a MathFP integer back to a normal integer.
`toString(int x)`	Returns the MathFP integer as `String`.
`toString(int x, int d)`	Returns the MathFP integer as `String` with rounding.

The following snippet of code demonstrates how the result of a particular MathFP operation can be converted to a `String`:

```
int resultOfMathFPMul = MathFP.mul (MathFP.toFP ("1.231"),
                                    MathFP.toFP ("3.14"));
System.out.println (MathFP.toString (resultOfMathFPMul));
```

In order to show how the MathFP library can be used in a real-life application, let's implement a small MIDP calculator. The main functionality of the application is done in the `commandAction()` method, where the arithmetic operations take place. MathFP functions are used after the user activates the equals (=) command, depending on the operator selected for the current calculation.

The complete source code of the `CalculatorMidp` MIDlet is shown in Listing 10.8. Figure 10.3 shows the application in action.

Figure 10.3. The `CalculatorMidp` MIDlet.

Listing 10.8 `CalculatorMidp.java`—The `CalculatorMidp` Sample Source Code

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

import net.jscience.math.kvm.*;

public class CalculatorMidp extends MIDlet implements CommandListener
{

    char operator = '=';
    StringItem operand1Item = new StringItem ("", "");
    TextField  operand2Field = new TextField ("", "", 7,
TextField.NUMERIC);
    TextField operand2FField = new TextField (".", "", 5,
TextField.NUMERIC);

    Display display;
    Form list;

    public CalculatorMidp() {
        display = Display.getDisplay (this);
        list = new Form ("Calculator");
        list.append (operand1Item);
        list.append (operand2Field);
        list.append (operand2FField);

        list.addCommand (new Command ("+", Command.SCREEN, 1));
        list.addCommand (new Command ("=", Command.SCREEN, 2));
        list.addCommand (new Command ("-", Command.SCREEN, 3));
        list.addCommand (new Command ("*", Command.SCREEN, 3));
        list.addCommand (new Command ("/", Command.SCREEN, 2));
        list.addCommand (new Command ("CLR", Command.SCREEN, 4));

        list.setCommandListener(this);
    }

    public void startApp() {
        display.setCurrent (list);
    }

    public void pauseApp() {
    }
```

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


    public void commandAction(Command c, Displayable d) {

        if (c.getLabel().equals ("CLR")) {
            operator = '=';
            operand1Item.setText ("");
            operand2Field.setLabel ("");
            operand2Field.setString ("");
            operand2Field.setString ("");
        }
        else {
            // 1. apply op

            int op2 = MathFP.toFP
                (operand2Field.getString()
                 + "." + operand2FField.getString());

            if (operator != '=') {
                int op1 = MathFP.toFP (operand1Item.getText());
                switch (operator) {
                case '+': op2 = MathFP.add(op1, op2); break;
                case '-': op2 = MathFP.sub (op1, op2); break;
                case '*': op2 = MathFP.mul (op1, op2); break;
                case '/': op2 = MathFP.div (op1, op2); break;
                }
            }

            String result = MathFP.toString (op2);
            operator = c.getLabel().charAt (0);

            if (operator != '=') {
                operand1Item.setText (MathFP.toString (op2));
                operand2Field.setLabel (c.getLabel());
                operand2Field.setString ("");
                operand2FField.setString ("");
            }
            else {
                operand1Item.setText ("");
                operand2Field.setLabel ("");
                int cut = result.indexOf (".");
                if (cut == -1) {
                    operand2Field.setString (result);
                    operand2FField.setString ("");
                }
                else {
                    operand2Field.setString (result.substring (0,
cut));
                    operand2FField.setString (result.substring
(cut+1));
                }
            }
        }
    }
}
```

The Bouncy Castle Crypto API

An additional feature that is currently unavailable in J2ME CLDC is the *Java Cryptography Extension (JCE).* The JCE is a set of packages providing a framework for encryption, key generation and key agreement, and *Message Authentication Code (MAC)* algorithms. It supports encryption for symmetric, asymmetric, block, and stream ciphers.

The Bouncy Castle Crypto API, which is available from http://www.bouncycastle.org, supports the following features:

- A clean-room implementation of the JCE 1.2.1
- A lightweight cryptography API in Java
- A provider for the JCE and JCA
- Generators for Version 1 and Version 3 X.509 certificates

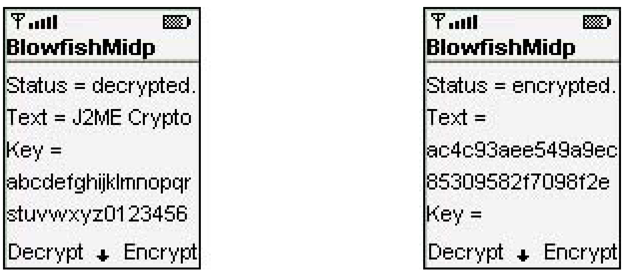
Because a complete implementation of the JCE is too big for the J2ME CLDC platform, we will focus on the lightweight API available for the J2ME CLDC platform. It is specially developed for circumstances in which the complete API and the integration of the JCE aren't required.

The lightweight API provides the following subset of the JCE:

- `AsymmetricBlockCipher`
- `BlockCipher`
- `BufferedBlockCipher`
- `BufferedAsymmetricBlockCipher`
- `BufferedStreamCipher`
- `Digest`
- `KeyAgreement`
- `Mac`
- `PBE`
- `StreamCipher`

For example, these tools let you cipher a password that needs to be transferred over a network for user authorization. Listing 10.9 shows an example of how a `String` can be ciphered using a private key with the Blowfish encryption algorithm that is part of the Bouncy Castle Crypto API. Encryption and decryption are performed entirely in the methods `encryptText()` and `decryptText()` in order to providing an easy mechanism for adopting to your own implementations. Figure 10.4 shows the original and the encrypted text in the MIDlet.

Figure 10.4. The Blowfish MIDlet showing decrypted text on the left and the encrypted on the right.



Listing 10.9 `BlowfishMidp.java`—The `BlowfishMidp` Sample Source Code

```
import java.io.*;
import java.lang.*;
```

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

import org.bouncycastle.util.encoders.*;

import org.bouncycastle.crypto.*;
import org.bouncycastle.crypto.engines.*;
import org.bouncycastle.crypto.modes.*;
import org.bouncycastle.crypto.params.*;

public class BlowFishMidp extends MIDlet implements CommandListener {

    private Display display = null;
    private boolean encrypted = false;

    private String key = "abcdefghijklmnopqrstuvwxyz0123456789";
    private String text = "J2ME Crypto";

    private BufferedBlockCipher bfCipher = null;

    private Form mainForm = null;

    private StringItem statusItem = new StringItem ("Status = ",
"init");
    private StringItem keyItem = new StringItem ("Key = ", key);
    private StringItem textItem = new StringItem ("Text = ", text);

    public static Command encrypt = new Command ("Encrypt",
Command.SCREEN, 1);
    public static Command decrypt = new Command ("Decrypt",
Command.SCREEN, 2);


    public BlowFishMidp() throws CryptoException {

        mainForm = new Form("BlowfishMidp");

        mainForm.append (statusItem);
        mainForm.append (textItem);
        mainForm.append (keyItem);

        mainForm.addCommand (encrypt);
        mainForm.addCommand (decrypt);
        mainForm.setCommandListener(this);
    }


    public void startApp() {
        display = Display.getDisplay(this);
        display.setCurrent(mainForm);
    }


    public void pauseApp() {}


    public void destroyApp(boolean unconditional) {}


    public void commandAction (Command c, Displayable d) {
```

```
        if (c == encrypt && !encrypted) {
            try {
                textItem.setText (encryptText(key, text));
                statusItem.setText ("encrypted.");
                encrypted = true;
            }
            catch (CryptoException ce) {
                throw new RuntimeException (ce.toString());
            }
        }
        else if (c == decrypt && encrypted) {
            try {
                textItem.setText (decryptText(key,
textItem.getText()));
                statusItem.setText ("decrypted.");
                encrypted = false;
            }
            catch (CryptoException ce) {
                throw new RuntimeException (ce.toString());
            }
        }
    }


    private String encryptText (String key, String text)
        throws CryptoException {

        byte[] keyBytes= Hex.decode(key.getBytes());
        byte[] ptBytes = text.getBytes();
        bfCipher = new PaddedBlockCipher
            (new CBCBlockCipher
                (new BlowfishEngine()));

        bfCipher.init(true, new KeyParameter(keyBytes));
        byte[] result = new
byte[bfCipher.getOutputSize(ptBytes.length)];
        int len = bfCipher.processBytes(ptBytes, 0, ptBytes.length,
result, 0);
        bfCipher.doFinal(result, len);

        return new String(Hex.encode(result));
    }


    private String decryptText(String key, String cipherText)
        throws CryptoException {

        byte[] keyBytes = Hex.decode(key.getBytes());
        byte[] textBytes = Hex.decode(cipherText.getBytes());

        bfCipher.init(false, new KeyParameter(keyBytes));

        byte[] result = new
byte[bfCipher.getOutputSize(textBytes.length)];
        int len = bfCipher.processBytes(textBytes, 0,
                                        textBytes.length, result, 0);
        bfCipher.doFinal(result, len);
        return new String(result).trim();
    }
}
```

User Interface Extensions

Although PDAP and MIDP provide user interface APIs appropriate to the corresponding devices, some extensions are available for both profiles.

For PDAP, kAWT provides some additional widgets such as a tabbed pane, a progress bar, and an option dialog to overcome some of the limitations in the widget set of AWT when compared to SWING.

Three different user interface extensions are available for MIDP. The *Open Windowing Toolkit (OWT)* provides a set of simple lightweight widgets based on the MIDP Canvas class. OWT is developed by DigitalFocus for Nextel and Motorola. Motorola itself provides a similar widget toolkit called the *Lightweight Windowing Toolkit (LWT).* Unfortunately, LWT is available for Motorola phones only. Finally, the kAWT toolkit provides an AWT subset on top of the MIDP canvas toolkit. Table 10.6 contains a brief overview of the APIs available. As with the XML parsers, there is a tradeoff between size and functionality. Figure 10.5 shows the OWT Grass Seed sample application; Figure 10.6 shows the KawtDemo MIDlet.

Figure 10.5. The OWT Grass Seed example application.

Figure 10.6. The KawtDemo MIDlet.

Table 10.6. User Interface APIs Available for MIDP

Toolkit	***JAR File Size***	***Remarks***
KAWT for MIDP	75.4KB	http://www.kawt.de AWT subset Optional additional widgets such as `TabbedPane` and `ProgressBar` available
LWT	(native)	http://developers.motorola.com/developers/wireless/tools/index.html#lwt Native implementation available for Motorola Accompli phones only

OWT	52.7KB	http://nextel.sourceforge.net

Summary

In this chapter, you have gained an overview of third-party libraries that overcome some limitations of the CLD Configuration and its profiles, including various libraries for XML parsing, kSOAP for remote method invocation, the bouncycastle library for cryptography, and mathFP for fixed point mathematics.

Appendix A. Class Library: CLDC Packages

IN THIS APPENDIX

This appendix gives an overview of the API provided in the *Connected Limited Device Configuration (CLDC) version 1.0 and version 1.1 also called CLDC Next Generation (NG).* It also lists profile-related additions to the configurations.

The java.io Package

The `java.io` package contains all the interfaces, classes, and exceptions that provide a mechanism for system input and output through data streams.

Interfaces

`DataInput`	The `DataInput` interface provides methods for reading binary streams and converting them into Java primitive data types.
`DataOutput`	The `DataOutput` interface provides methods for writing/converting Java primitive data types to a binary stream.

Classes

`ByteArrayInputStream`	The `ByteArrayInputStream` encapsulates an `InputStream` that uses a byte array as a buffer that might be read from a stream.
`ByteArrayOutputStream`	The `ByteArrayOutputStream` provides an `OutputStream` that might be used to write data to a byte array.
`DataInputStream`	The `DataInputStream` implements the `DataInput` interface and provides methods to read primitive Java data types from a stream.
`DataOutputStream`	The `DataOutputStream` implements the `DataOutput` interface and provides methods to write primitive Java data types to a stream.
`InputStream`	The `InputStream` is the abstract superclass of all byte-related input streams.
`InputStreamReader`	The `InputStreamReader` closes the gap between byte-related input streams and character-related input streams. Bytes that are read using this class are internally converted to characters.
`OutputStream`	The `OutputStream` is the abstract superclass of all byte-related output streams.

`OutputStreamWriter`	The `OutputStreamWriter` closes the gap between byte-related output streams and character-related output streams. Characters that are written using this class are internally converted to bytes.
`PrintStream`	The `PrintStream` is an extension to an `OutputStream`. It adds functionality that lets you print representations of various data types.
`Reader`	The `Reader` is the abstract superclass of all character-oriented `InputStreams`.
`Writer`	The `Writer` is the abstract superclass of all character-oriented `OutputStreams`.

Exceptions

`EOFException`	The `EOFException` signals that an unexpected end of the stream has been reached during input.
`InterruptedIOException`	The `InterruptedIOException` signals that an I/O operation has been interrupted.
`IOException`	The `IOException` signals that an exception of some kind has occurred.
`UnsupportedEncodingException`	The `UnsupportedEncodingException` signals that the character encoding is not supported.
`UTFDataFormatException`	The `UTFDataFormatException` signals that a malformed UTF-8 String has been read by a class implementing the `DataInput` interface.

The java.lang Package

This package provides classes that are fundamental to the design of the Java programming language.

Interface

The `Runnable` interface needs to be implemented by a class when its instances are intended for execution by a thread.

Classes

`Boolean`	The `Boolean` class is a wrapper class for a value of the primitive Java data type `boolean`.
`Byte`	The `Byte` class is a wrapper class for a value of the primitive Java data type `byte`.
`Character`	The `Character` class is a wrapper class for a value of the primitive Java data type `char`.
`Class`	The instances of the class `Class` represent Java classes and interfaces of a running Java application.
`Double`	The `Double` class is a wrapper class for a value of the primitive type `double`. This class is part of CLDC since version 1.1.
`Float`	The `Float` class is a wrapper class for a value of the primitive type `float`. This class is part of CLDC since version 1.1.
`Integer`	The `Integer` class is a wrapper class for a value of the primitive Java data type `int`.
`Long`	The `Long` class is a wrapper class for a value of the primitive Java data type `long`.
`Math`	The `Math` class provides some basic mathematical operations on the Java primitive data types `int`,`long`, `float` and `double`.
`Object`	The `Object` class is the superclass of every Java class in a class hierarchy.
`Runtime`	The `Runtime` class provides every Java application with an interface for interacting with the application environment where it is executed. Only one instance of the Runtime class is available in a running application.
`Short`	The `Short` class is a wrapper class for a value of the primitive Java data type `short`.
`String`	The `String` class represents character-based strings.
`StringBuffer`	The `StringBuffer` class represents a mutable sequence of characters.
`System`	The `System` class provides system-related methods.
`Thread`	The `Thread` class represents a running thread of a Java application.
`Throwable`	The `Throwable` class is the superclass of every exception and error in the Java language.

Exceptions

`ArithmeticException`	The `ArithmeticException` is thrown if an unexpected arithmetic exception occurs.
`ArrayOutOfBoundsException`	The `ArrayOutOfBoundsException` is thrown when an illegal index of an array is accessed.
`ArrayStoreException`	The `ArrayStoreException` is thrown to indicate

	that the code has tried to store an incorrect object type in an array of objects.
`ClassCastException`	The `ClassCastException` is thrown to indicate that the code has tried to convert the class into a subclass that it is not an instance of.
`ClassNotFoundException`	The `ClassNotFoundException` is thrown to indicate that the code has used the `forName()` method to try to load a class that could not be found.
`Exception`	The `Exception` class and all its subclasses indicate conditions that might be watched by an application.
`IllegalAccessException`	The `IllegalAccessException` indicates that an application tried to load in a class, but the currently executing method did not have access to the definition of the specified class, because the class was not public and was in another package.
`IllegalArgumentException`	The `IllegalArgumentException` is thrown to indicate that a method has passed an illegal parameter.
`IllegalMonitorStateException`	The `IllegalMonitorStaeException` is thrown to indicate that a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor.
`IllegalStateException`	The `IllegalStateException` is an MIDP extension to CLDC and is available only in MIDP-conforming Java Virtual Machines. It indicates that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an appropriate state for the requested operation.
`IllegalThreadStateException`	The `IllegalThreadStateException` is thrown to indicate that a method has been invoked on a thread that is not in an appropriate state.
`IndexOutOfBoundsException`	The `IndexOutOfBoundsException` is thrown in order to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.
`InstantiationException`	The `InstantiationException` is thrown to indicate that an application has tried to create an instance using the `newInstance()` method for an instance of an interface or abstract class.
`InterrruptedException`	The `InterruptedException` is thrown to indicate when a thread is interrupted while waiting, sleeping, or otherwise pausing for a long period of time.
`NegativeArraySizeException`	The `NegativeArraySizeException` is thrown to indicate that the application has tried to create an array of negative size.
`NullPointerException`	`The NullPointerException` is thrown to indicate that an application tried to use `null` in a case where an `Object` is required.
`NumberFormatException`	The `NumberFormatException` is thrown to indicate that the application has attempted to

	convert a string to one of the numeric types, but the string did not have the appropriate format.
`RuntimeException`	The `RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.
`SecurityException`	The `SecurityException` is thrown if the security manager discovers a security violation.
`StringIndexOutOfBoundsException`	The `StringIndexOutOfBoundsException` is thrown by the charAt method in the class `String` and by other `String` methods to indicate that an index is either negative, greater than, or equal to the size of the string.

Errors

`Error`	The `Error` class is a subclass of the `Throwable` class indicating serious problems that an application should not try to catch.
`NoClassDefFoundError`	The `NoClassDefFoundError` occurs, if the Java Virtual Machine tries to load in the definition of a class and no definition of the class could be found. This error is part of CLDC since version 1.1.
`OutOfMemoryError`	The `OutOfMemoryError` is thrown in order to indicate that the Java Virtual Machine cannot allocate an object, because it is out of memory, and no more memory could be made available by the garbage collector.
`VirtualMachineError`	The `VirtualMachineError` is thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating.

The java.lang.ref Package

This package contains support for weak references and is part of CLDC since version 1.1.

Classes

`Reference`	The `Reference` class is the abstract base class for all reference objects and may not be sub classed directly.
`WeakReference`	The `WeakReference` class provides support for weak references which are most often used to implement canonicalizing mappings.

The java.util Package

This package contains date and time facilities and miscellaneous utility classes.

Interface

The `Enumeration` interface provides methods for accessing a series of elements in a class implementing this interface.

Classes

`Calendar`	The `Calendar` class is an abstract class for setting and getting dates using a set of integer fields.
`Date`	The `Date` class represents a specific point of time in millisecond precision.
`Hashtable`	The `Hashtable` class implements a hashtable in order to map keys to values.
`Random`	The `Random` class implements a stream of pseudorandom numbers.
`Stack`	The `Stack` class implements a last-in-first-out (LIFO) stack of objects.
`Timer`	The `Timer` class is an MIDP extension to CLDC and is available only in MIDP-conforming Java Virtual Machines. The class provides a mechanism for threads to schedule tasks for future executions in a background thread.
`TimerTask`	The `TimerTask` class is an MIDP extension to CLDC and is available only in MIDP-conforming Java Virtual Machines. A `Timer` can schedule a `TimerTask` for an one-time or a repeated execution.
`TimeZone`	The `TimeZone` class represents a time zone offset, and also calculates daylight savings time changes.
`Vector`	The `Vector` class implements a mutable array of objects.

Exceptions

`EmptyStackException`	The `EmptyStackException` is thrown by the methods of the `Stack` class to indicate that the stack is empty.
`NoSuchElementException`	The `NoSuchElementException` is thrown by the `nextElement()` method of an `Enumeration` to indicate that there are no more elements in the enumeration.

The javax.microedition.io Package

This package contains all interfaces, classes, and exceptions of the generic connection framework.

Interfaces

CommConnection	The CommConnection interface is a PDAP extension to CLDC and is only available in PDAP-conforming Java Virtual Machines. The CommConnection interface defines all necessary methods for a logical serial port connection.
Connection	The Connection interface is the basic type of generic connection and a superclass of all connections.
ContentConnection	The ContentConnection interface defines methods for a stream connection over which content is passed.
Datagram	The Datagram interface defines generic methods for a datagram that is used by the DatagramConnection.
DatagramConnection	The DatagramConnection interface defines all necessary methods for datagram connections.
FileConnection	The FileConnection interface is a PDAP extension to CLDC and is available only in PDAP-conforming Java Virtual Machines. The FileConnection interface defines all necessary methods to access files that are stored on removable media.
FileSystemEvent	The FileSystemEvent interface is a PDAP extension to CLDC and is available only in PDAP-conforming Java Virtual Machines. The FileSystemEvent interface defines all necessary methods needed for an event used to detect when a file system is added and removed on a device.
HttpConnection	The HttpConnection interface is an MIDP extension to CLDC and is available only in MIDP-conforming Java Virtual Machines. The HttpConnection interface defines all necessary methods and constants for a HTTP connection.
InputConnection	The InputConnection interface defines all necessary methods for an input connection.
OutputConnection	The OutputConnection interface defines all necessary methods for an output connection.
StreamConnection	The StreamConnection interface is the combination of the InputConnection and the OutputConnection.
StreamConnectionNotifier	The StreamConnectionNotifier interface defines the methods that a stream connection notifier must have.

Classes

Connector	The Connector class provides a set of static methods for handling all kinds of connections contained in the generic connection framework.
FileSystemListener	The FileSystemListener is a PDAP extension to CLDC and is available only in PDAP-conforming Java Virtual Machines. The FileSystemListener is used for receiving FileSystemEvents while adding or removing a file system root.

Exception

The `ConnectionNotFoundException` is thrown to indicate that a particular connection passed to the `Connector.open()` methods can not be found.

MIDP-Specific Packages

The MIDP-specific packages `javax.microedition.lcdui`, `javax.microedition.midlet`, and `javax.microedition.rms` are described in the following sections.

The `javax.microedition.lcdui` Package

This package provides a set of features for implementation of user interfaces for MIDP applications.

Interfaces

`Choice`	The `Choice` interface defines methods for UI components implementing the capability of selecting elements from a predefined number of elements.
`CommandListener`	The `CommandListener` interface defines methods that are used by applications that want to receive high-level commands from the UI implementation.
`ItemStateListener`	The `ItemStateListener` interface defines methods that are used by applications that want to receive events that indicate changes in the internal state of the interactive items within a `Form` screen.

Classes

`Alert`	The `Alert` class provides a screen that shows data to the user and waits for a specified period of time before proceeding to the next screen.
`AlertType`	The `AlertType` class provides an indication of the behavior of alerts.
`Canvas`	The `Canvas` class is the base class for writing applications that need to handle low-level events and to issue graphics calls for drawing to the display at low-level.
`ChoiceGroup`	The `ChoiceGroup` class is a group of selectable elements that needs to be appended to a `Form`.
`Command`	The `Command` class encapsulates the semantic information of an action.
`DateField`	The `DateField` class provides a UI component for editing date and time information. This UI component needs to be appended to a `Form`.
`Display`	The `Display` class represents the manager of the display and input devices of the system.
`Displayable`	The `Displayable` class encapsulates an object that has the capabilities to be placed on the `Display`.
`Font`	The `Font` class represents a font and font metrics.
`Form`	The `Form` class is a `Screen` that contains an arbitrary mixture of all available high-level UI components.
`Gauge`	The `Gauge` class represents a bar graph display of a value that needs to be placed in a `Form`.
`Graphics`	The `Graphics` class provides simple 2D geometric rendering in the low-level API.
`Image`	The `Image` class is used to hold image data.
`ImageItem`	The `ImageItem` class provides layout functionality for images that need to be placed in a `Form` or `Alert`.
`Item`	The `Item` class is the superclass of all UI components that can be appended

	to Forms and Alerts.
List	The List class provides a Screen containing a list of choices.
Screen	The Screen class is the superclass of all high-level user interface classes.
StringItem	The StringItem class is an Item containing a String.
TextBox	The TextBox class provides a Screen that is capable of entering and editing text.
TextField	The TextField class provides an Item that is capable of entering and editing text that needs to be appended to a Form.
Ticker	The Ticker class implements a UI component where the text scrolls continuously across the display.

The javax.microedition.midlet Package

This package defines MIDP applications and the interactions between the application and the environment in which the application is executed.

Class

The MIDlet class provides an application for the MID profile.

Exception

The MIDletStateChangeException signals that a requested MIDlet state change has failed.

The javax.microedition.rms Package

This package provides a mechanism to persistently store data and later retrieve it.

Interfaces

RecordComparator	The RecordComparator interface defines methods for a comparator that compares two records, depending on the application-defined criteria, in order to see if they match or to determine their relative sort order.
RecordEnumeration	The RecordEnumeration interface defines methods for a bidirectional record store record enumerator.
RecordFilter	The RecordFilter interface defines methods for a filter that examines a record to see if it matches depending on the application-defined criteria.
RecordListener	The RecordListener interface defines methods for receiving events indicating that a Record was changed, added, or deleted from a record store.

Class

The RecordStore class represents one record store.

Exceptions

InvalidRecordIDException	The InvalidRecordIDException is thrown to indicate that an operation could not be completed, because the record ID was invalid.

`RecordStoreException`	The `RecordStoreException` is thrown to signal that a general exception occurred in a record store operation.
`RecordStoreFullException`	The `RecordStoreFullException` is thrown to indicate that an operation could not be completed, because the record store system storage was full.
`RecordStoreNotFoundException`	The `RecordStoreNotFoundException` is thrown to indicate that an operation could not be completed, because the record store could not be found.
`RecordStoreNotOpenException`	The `RecordStoreNotOpenException` is thrown to indicate that an operation was attempted on a closed record store.

PDAP-Specific Packages

The `java.awt`, `java.awt.event`, `java.awt.image`, `javax.microedition.pim` and the additions to the `java.io`, `java.net`, `java.util` packages are available in PDAP only.

The `java.awt` Package

This package contains all the classes for creating user interfaces and for painting graphics and images for implementation of user interfaces for PDAP applications.

Interfaces

`ActiveEvent`	`ActiveEvent` provides an interface for events that know how dispatch themselves. By implementing this interface, an event can be placed upon the event queue and its `dispatch()` method will be called when the event is dispatched, using the `EventDispatchThread`.
`Adjustable`	The `Adjustable` interface is used for objects having an adjustable numeric value contained within a bounded range of values.
`ItemSelectable`	The `ItemSelectable` provides an interface for objects containing a set of items for which zero or more can be selected.
`LayotManager`	The `LayoutManager` interface defines methods for classes knowing how to layout Containers.
`LayoutManager2`	`LayoutManager2` provides an interface for classes that know how to layout Containers based on a layout constraints object. It extends the `LayoutManager` interface to handle layouts explicitly in terms of constraint objects that specify how and where components should be added to the layout.
`MenuContainer`	The `MenuContainer` interface provides methods for all menu related containers.
`Shape`	The `Shape` interface provides definitions for graphical objects that represent some kind of geometric shapes.

Classes

`AWTEvent`	The `AWTEvent` class is the root event for all AWT related events.
`AWTEventMulticaster`	The `AWTEventMulsticaster` class provides a mechanism for efficient and thread-safe multi-cast event dispatching for the AWT events included in the `java.awt.event` package.
`BorderLayout`	The `BorderLayout` lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center.
`Button`	The `Button` class represents a labeled button.
`Canvas`	The `Canvas` class represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user.
`CardLayout`	The `CardLayout` lays out a container, arranging each component in the container as a card where only one card is visible at a time.
`Checkbox`	The `Checkbox` class represents a graphical component that can be in either an "on" (true) or "off" (false) state.

CheckboxGroup	The CheckboxGroup class is used to combine a set Checkboxes together in a group.
CheckboxMenuItem	The CheckboxMenuItem represents a Checkbox that can be added to a menu.
Choice	The Choice class represents a graphical pop-up menu component of choices.
Color	The Color class is used to encapsulate colors in the RGB color space.
Component	The Component class is the abstract super class of objects having a graphical representation that can be displayed on the screen and are able to interact with the user.
Container	The Container class is a component that can contain other AWT components.
Cursor	The Cursor class provides the bitmap representation of the mouse cursor.
Dialog	A Dialog class is a top-level window with a title and a border that can be used to take input from the user.
Dimension	The Dimension class encapsulates the width and height in integer precision of a component in a single object.
Event	The Event class is a platform-independent implementation of events that can be dispatched from the platform's Graphical User Interface in the Java 1.0 event model.
EventQueue	The EventQueue class provides a platform-independent mechanism for queuing events.
FlowLayout	The FlowLayout class lays out components in a left-to-right flow, like lines of text in a paragraph.
Font	The Font class represents a font.
FontMetrics	The FontMetrics class defines a font metrics object, which encapsulates information about the rendering of a particular font on a particular screen.
Frame	The Frame class represents a top-level window with a title and a border.
Graphics	The Graphics class is the abstract base class for all graphics contexts that allow an application to draw onto components that are realized on various devices, as well as onto off-screen images.
GraphicsConfiguration	The GraphicsConfiguration class describes the capabilities of a graphics destination.
GraphicsDevice	The GraphicsDevice class holds the graphics devices that might be available in a particular graphics environment.
GraphicsEnvironment	The GraphicsEnvironment class describes a set of GraphicsDevice and Font objects available on a particular platform.
GridBagConstraints	The GridBagConstraints class defines constraints for components that are laid out in a GridBagLayout.
GridBagLayout	The GridBagLayout class provides a flexible layout manager that is capable of positioning components according to constraints in the GridBayConstraints class.
GridLayout	The GridLayout class lays out the components of a container in a rectangular grid.
Image	The Image class is the super-class of all classes that represent graphical images.

Insets	The Insets class represents the borders of a container. It specifies the space that a container must leave at each of its edges.
Label	The Label class represents a component for placing text in a container.
List	The List class represents a graphical component with a scrolling list of text items.
MediaTracker	The MediaTracker class is a utility class providing a mechanism to track the status of media objects such as images.
Menu	The Menu class represents an object acting as pull-down menu component that is deployed in a menu bar.
MenuBar	The MenuBar class encapsulates the platform's concept of a menu bar bound to a frame.
MenuComponent	The MenuComponent class is the super-class of all menu-related components.
MenuItem	The MenuItem class represents on item in a menu.
MenuShortcut	The MenuShortcut class represents a keyboard accelerator for a MenuItem.
Panel	The Panel class is the simplest container class.
Point	The Point class represents a location in (x, y) coordinate space, specified in integer precision.
Polygon	The Polygon class represents a geometric description of a closed, two-dimensional region within a coordinate space.
PopupMenu	The Popup class a menu that can be dynamically popped up at a specified position within a component.
Rectangle	The Rectangle class specifies an area in a coordinate space that is enclosed by the Rectangle object's top-left point (x, y) in the coordinate space, its width, and its height.
Scrollbar	The Scrollbar class represents a scroll bar user-interface object.
ScrollPane	The ScrollPane class represents a container class which implements automatic horizontal and/or vertical scrolling for a single child component.
SystemColor	The SystemColor class encapsulates a set symbolic colors representing the color of GUI objects on a particular platform.
TextArea	The TextArea class represents a graphical component capable of displaying a multi-line region text.
TextComponent	The TextComponent class is the super-class of any component that allows the editing of text.
TextField	The TextField class represents a graphical component allowing the user to edit a single line of text.
Toolkit	The Toolkit class is the abstract super-class of all actual implementations of the Abstract Window Toolkit. Subclasses of Toolkit are used to bind the various components to particular native toolkit implementations.
Window	The Window class is a top-level window with no borders and no menubar.

Exceptions

AWTException	The AWTException signalizes that an AWT related

	excaption occurred.
IllegalComponentStateException	The IllegalComponentStateException signalizes that an AWT component is not in an appropriate state for the operation.

Error

The java.awt package consists of the AWTError only that signalizes that a serious error in the Abstract Window Toolkit occurred.

The java.awt.event Package

This package provides interfaces and classes for dealing with different types of events fired by AWT components.

Interfaces

ActionListener	The ActionListener interface provides methods for receiving action events.
AdjustmentListener	The AdjustmentListener interface provides methods for receiving adjustment events.
AWTEventListener	The AWTEventListener interface provides methods for receiving notification of events dispatched to objects that are instances of Component or MenuComponent or their subclasses.
ComponentListener	The ComponentListener interface provides methods for receiving component events.
ContainerListener	The ContainerListener interface provides methods for receiving container events.
FocusListener	The FocusListener interface provides methods for receiving keyboard focus events on a component.
ItemListener	The ItemListener interface provides methods for receiving item events.
KeyListener	The KeyListener interface provides methods for receiving keyboard events (keystrokes).
MouseListener	The MouseListener interface provides methods for receiving mouse events such as (press, release, click, enter, and exit) on a component.
MouseMotionListener	The MouseMotionListener interface provides methods for receiving mouse motion events on a component.
TextListener	The TextListener interface provides methods for receiving text events.
WindowListener	The WindowListener interface provides methods for receiving window events.

Classes

ActionEvent	The ActionEvent class represents a semantic event, indicating that a component-defined action occurred.
AdjustmentEvent	The AdjustmentEvent class encapsulates an event emitted by Adjustable objects.
ComponentAdapter	The ComponentAdapter class provides an abstract adapter class for receiving component events.

ComponentEvent	The ComponentEvent class encapsulates a low-level event indicating that a component moved, changed size, or changed visibility
ContainerAdapter	The ContainerAdapter class provides an abstract adapter class for receiving container events.
ContainerEvent	The ContainerEvent class encapsulates a low-level event which indicates that a container's contents has changed because a component was added or removed.
FocusAdapter	The FocusAdapter class provides an abstract adapter class for receiving focus events.
FocusEvent	The FocusEvent class encapsulates a low-level event indicating that a component has gained or lost the keyboard focus.
InputEvent	The InputEvent class represents the root event class for all component-level input events.
InvocationEvent	The InvocationEvent encapsulates an event executing the run() method on a Runnable object when dispatched by the AWT event dispatcher thread.
ItemEvent	The ItemEvent encapsulates a semantic event which indicates that an item was selected or deselected.
KeyAdapter	The KeyAdapter class provides an abstract adapter class for receiving keyboard events.
KeyEvent	The KeyEvent class encapsulates an event indicating that a keystroke occurred in a component.
MouseAdapter	The MouseAdapter class provides an abstract adapter class for receiving mouse events.
MouseEvent	The MouseEvent class encapsulates an event indicating that a mouse action occurred in a component.
MouseMotionAdapter	The MouseMotionAdapter class provides an abstract adapter class for receiving mouse motion events.
PaintEvent	The PaintEvent class encapsulates a component-level paint event.
TextEvent	The TextEvent class encapsulates a semantic event indicating that an object's text changed.
WindowAdapter	The WindowAdapter class provides an abstract adapter class for receiving window events.
WindowEvent	The WindowEvent class encapsulates a low-level event indicating that a window has changed its status.

The `java.awt.image` Package

This package provides classes for creating and modifying images.

Interfaces

ImageConsumer	The ImageConsumer interface is used for expressing interest in image data through the ImageProducer interfaces.
ImageObserver	The ImageObserver interface is asynchronous update interface for receiving notifications about Image information as the Image is constructed.
ImageProducer	The ImageProducer interface defines methods for objects that can produce the image data for images.

Classes

AreaAveragingScaleFilter	The AreaAveragingScaleFilter class is used to scale images using a simple area averaging algorithm that produces smoother results than the nearest neighbor algorithm.
ColorModel	The ColorModel class is an abstract class encapsulating methods for translating a pixel value to color components and an alpha component.
CropImageFilter	The CropImagefilter class is used for cropping images.
DirectColorModel	The DirectColorModel is a ColorModel class that works with pixel values that represent RGB color and alpha information as separate samples and that pack all samples for a single pixel into a single int, short, or byte quantity.
FilteredImageSource	The FilteredImageSource class is an implementation of the ImageProducer interface which takes an existing image and a filter object and uses them to produce image data for a new filtered version of the original image.
ImageFilter	The ImageFilter class implements a filter for the set of interface methods that are used to deliver data from an ImageProducer to an ImageConsumer.
IndexColorModel	The IndexColorModel class that works with pixel values consisting of a single sample which is an index into a fixed colormap.
MemoryImageSource	The MemoryImageSource is an implementation of the ImageProducer interface which uses an array to produce pixel values for an Image.
PixelGrabber	The PixelGrabber class is an implementation of the ImageConsumer class which can be attached to an Image or ImageProducer object to retrieve a subset of the pixels in that image.
ReplicateScaleFilter	The ReplicateScaleFilter class is used for scaling images using the simplest algorithm.
RGBImageFilter	The RGBImageFilter class provides an easy way to create an ImageFilter which modifies the pixels of an image in the default RGB ColorModel.

The javax.microedition.pim Package

This package contains all of the classes for accessing the Personal Information Manager for PDAP applications.

Interfaces

Contact	The Contact interface defines all methods that need to be implemented by a Contact object of an address book.
ContactList	The ContactList interface defines all methods that need to be implemented by a ContactList object which is used to store the Contacts of an address book
Event	The Event interface defines all methods that needs to be implemented by an Event object of a calendar.
EventList	The EventList interface defines all methods that needs to be implemented by a EventList object which is used to store Events of a given calendar.
PIMElement	The PIMElement interface defines all methods necessary for an element of a PIMList. Interfaces extending the PIMElement interface are Contact,

	Event and ToDo.
PIMList	The PIMList interface defines all methods necessary for a list that is capable of storing PIMElements. Interfaces extending the PIMList interface are ContactList, EventList and ToDoList.
ToDo	The ToDo interface defines all methods that need to be implemented by an entry of a ToDoList.
ToDoList	The ToDoList interface defines all methods that needs to be implemented by a ToDoList object which is used to store ToDo elements.

Classes

EventRepeat	The EventRepeat class represents a description for a repeating pattern for an Event element.
PIM	The PIM class is used to access all PIM databases which are available on the device by providing static access methods.

Exception

PimException	The PimException is thrown to indicate that a general error in the pim classes occurred.

PDAP Additions to the java.io package

PDAP adds the PrintWriter class which is used to print formatted representations of objects to a text-output stream.

PDAP Additions to the java.lang.reflect package

PDAP adds the InvocationTargetException that can be thrown in the java.awt. EventQueue.invokeAndWait() method.

PDAP Additions to the java.net package

PDAP need to add the following class and exception to the java.net package.

URL	The URL class represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web.
MalformedURLException	The MalformedURLException is thrown to indicate that a malformed URL has occurred.

PDAP Additions to the java.util package

PDAP need to add the following class and exception to the java.net package.

Interface

EventListener	The EventListener interface is a tagging interface that all event listener interfaces must extend.

Classes

`EventObject`	The `EventObject` is the root class from which all event state objects shall be derived.
`Locale`	The `Locale` class represents a specific geographical, political, or cultural region.

Exception

`MissingResourceException`	The `MissingResourceException` is thrown to indicate that a resource could not be found.

Appendix B. Comparison Charts

IN THIS APPENDIX

This appendix compares J2SE classes (v 1.3) to their J2ME CLDC and CLDC-NG(1.1)/PDAP counterparts. If a J2SE class is contained in J2ME, but methods are omitted, a detailed comparison is given in the sections following the package descriptions.

java.awt

The `java.awt` package and its sub packages `java.awt.event` and `java.awt.image` are available in PDAP only.

For a discussion of the MIDP counterpart, the javax.microedition.lcdui package, refer to Chapter 3, "MIDP Programming."

Table B.1. Interfaces of the `java.awt` Package

J2SE Interface	***Availability in PDAP***
`ActiveEvent`	All J2SE Methods are available in PDAP.
`Adjustable`	All J2SE Methods are available in PDAP.
`Composite`	Not available in PDAP.
`CompositeContext`	Not available in PDAP.
`ItemSelectable`	All J2SE Methods are available in PDAP.
`LayotManager`	All J2SE Methods are available in PDAP.
`LayoutManager2`	All J2SE Methods are available in PDAP.
`MenuContainer`	All J2SE Methods are available in PDAP.
`Paint`	Not available in PDAP.
`PaintContext`	Not available in PDAP.
`PrintGraphics`	Not available in PDAP.
`Shape`	Partly contained; see Table B.5 for details.
`Stroke`	Not available in PDAP.
`Transparency`	Not available in PDAP.

Table B.2. Classes of the `java.awt` Package

J2SE Class	***Availability in PDAP***
`AlphaComposite`	Not available in PDAP.
`AWTEvent`	Fully available in PDAP.
`AWTEventMulticaster`	Partially contained; see Table B.6 for details.
`AWTPermission`	Not available in PDAP.
`BasicStroke`	Not available in PDAP.
`BorderLayout`	All J2SE Methods are available in PDAP.
`Button`	Partially contained; see Table B.7 for details.
`Canvas`	Partially contained; see Table B.8 for details.
`CardLayout`	All J2SE Methods are available in PDAP.
`Checkbox`	Partially contained; see Table B.9 for details.
`CheckboxGroup`	All J2SE Methods are available in PDAP.
`CheckboxMenuItem`	Partly contained; see Table B.10 for details.
`Choice`	Partially contained; see Table B.11 for details.
`Color`	Partially contained; see Table B.12 for details.
`Component`	Partially contained; see Table B.13 for details.
`ComponentOrientation`	Not available in PDAP.
`Container`	Partially contained; see Table B.14 for details.
`Cursor`	Partly contained; see Table B.15 for details.
`Dialog`	Partially contained; see Table B.16 for details.
`Dimension`	Partially contained; see Table B.17 for details.
`Event`	All J2SE Methods are available in PDAP.
`EventQueue`	All J2SE Methods are available in PDAP.

FileDialog	Partially contained; see Table B.18 for details.
FlowLayout	All J2SE Methods are available in PDAP.
Font	Partially contained; see Table B.19 for details.
FontMetrics	Partially contained; see Table B.20 for details.
Frame	Partially contained; see Table B.21 for details.
GradientPaint	Not available in PDAP.
Graphics	Partially contained; see Table B.22 for details.
Graphics2D	Not available in PDAP, please use Graphics instead.
GraphicsConfigTemplate	Not available in PDAP.
GraphicsConfiguration	Partially contained; see Table B.23 for details.
GraphicsDevice	Partially contained; see Table B.24 for details.
GraphicsEnvironment	Partially contained; see Table B.25 for details.
GridBagConstraints	Partially contained; see Table B.26 for details.
GridBayLayout	Partially contained; see Table B.27 for details.
GridLayout	All J2SE methods are available in PDAP.
Image	All J2SE methods are available in PDAP.
Insets	Partially contained; see Table B.28 for details.
JobAttributes	Not available in PDAP.
Label	Partially contained; see Table B.29 for details.
List	Partially contained; see Table B.30 for details.
MediaTracker	All J2SE methods are available in PDAP.
Menu	Partially contained; see Table B.31 for details.
MenuBar	Partially contained; see Table B.32 for details.
MenuComponent	Partially contained; see Table B.33 for details.
MenuItem	Partially contained; see Table B.34 for details.
MenuShortcut	All J2SE methods are available in PDAP.
PageAttributes	Not available in PDAP.
Panel	Partially contained; see Table B.35 for details.
Point	Partially contained; see Table B.36 for details.
Polygon	Partially contained; see Table B.37 for details.
PopupMenu	Partially contained; see Table B.38 for details.
PrintJob	Not available in PDAP.
Rectangle	Partially contained; see Table B.39 for details.
RenderingHints	Not available in PDAP.
Robot	Not available in PDAP.
Scrollbar	Partially contained; see Table B.40 for details.
ScrollPane	Partially contained; see Table B.41 for details.
SystemColor	Partially contained; see Table B.42 for details.
TextArea	Partially contained; see Table B.43 for details.
TextComponent	Partially contained; see Table B.44 for details.
TextField	Partially contained; see Table B.45 for details.
TexturePaint	Not available in PDAP.
Toolkit	Partially contained; see Table B.46 for details.
Window	Partially contained; see Table B.47 for details.

Table B.3. Exceptions of the `java.awt` Package

J2SE Exception	***Availability in PDAP***
AWTException	Available in PDAP.

`FontFormatException`	Not available in PDAP.
`IllegalComponentStateException`	Available in PDAP.

Table B.4. Errors of the `java.awt` Package

J2SE Error	*Availability in PDAP*
`AWTError`	Available in PDAP.

Shape

Table B.5. Methods ofthe Class Shape

Method	*Alternative/Workaround*
`boolean contains(double x, double y)`	Not available in PDAP.
`boolean contains(double x, double y, double w, double h)`	Not available in PDAP.
`boolean contains(Point2D p)`	Not available in PDAP.
`boolean contains(Rectangle2D r)`	Not available in PDAP.
`Rectangle getBounds()`	Available in PDAP.
`Rectangle2D getBounds2D()`	Not available in PDAP.
`PathIterator getPathIterator (AffineTransform at)`	Not available in PDAP.
`PathIterator getPathIterator (AffineTransform at, double flatness)`	Not available in PDAP.
`boolean intersects(double x, double y, double w, double h)`	Not available in PDAP.
`boolean intersects(Rectangle2D r)`	Not available in PDAP.

`AWTEventMulticaster`

Table B.6. Methods of the Class `AWTEventMulticaster`

Method	*Alternative/Workaround*
`Protected AWTEventMulticaster (EventListener a, EventListener b)`	Available in PDAP.
`void actionPerformed(ActionEvent e)`	Available in PDAP.
`static ActionListener add(ActionListener a, ActionListener b)`	Available in PDAP.
`static AdjustmentListener add(AdjustmentListener a, AdjustmentListener b)`	Available in PDAP.
`static ComponentListener add(ComponentListener a, ComponentListener b)`	Available in PDAP.
`static ContainerListener add(ContainerListener a, ContainerListener b)`	Available in PDAP.
`static FocusListener add(FocusListener a, FocusListener b)`	Available in PDAP.
`static HierarchyBoundsListener add(HierarchyBoundsListener a, HierarchyBoundsListener b)`	Not available in PDAP.
`static HierarchyListener add(HierarchyListener a, HierarchyListener b)`	Not available in PDAP.
`static InputMethodListener add(InputMethodListener a, InputMethodListener b)`	Not available in PDAP.
`static ItemListener add(ItemListener a, ItemListener b)`	Available in PDAP.
`static KeyListener add(KeyListener a, KeyListener`	Available in PDAP.

b)	
static MouseListener add(MouseListener a, MouseListener b)	Available in PDAP.
static MouseMotionListener add(MouseMotionListener a, MouseMotionListener b)	Available in PDAP.
static TextListener add(TextListener a, TextListener b)	Available in PDAP.
static WindowListener add(WindowListener a, WindowListener b)	Available in PDAP.
protected static EventListener addInternal (EventListener a, EventListener b)	Available in PDAP.
void adjustmentValueChanged(AdjustmentEvent e)	Available in PDAP.
void ancestorMoved(HierarchyEvent e)	Not available in PDAP.
void ancestorResized(HierarchyEvent e)	Not available in PDAP.
void caretPositionChanged(InputMethodEvent e)	Not available in PDAP.
void componentAdded(ContainerEvent e)	Available in PDAP.
void componentHidden(ComponentEvent e)	Available in PDAP.
void componentMoved(ComponentEvent e)	Available in PDAP.
void componentRemoved(ContainerEvent e)	Available in PDAP.
void componentResized(ComponentEvent e)	Available in PDAP.
void componentShown(ComponentEvent e)	Available in PDAP.
void focusGained(FocusEvent e)	Available in PDAP.
void focusLost(FocusEvent e)	Available in PDAP.
void hierarchyChanged(HierarchyEvent e)	Not available in PDAP.
void inputMethodTextChanged(InputMethodEvent e)	Not available in PDAP.
void itemStateChanged(ItemEvent e)	Available in PDAP.
void keyPressed(KeyEvent e)	Available in PDAP.
void keyReleased(KeyEvent e)	Available in PDAP.
void keyTyped(KeyEvent e)	Available in PDAP.
void mouseClicked(MouseEvent e)	Available in PDAP.
void mouseDragged(MouseEvent e)	Available in PDAP.
void mouseEntered(MouseEvent e)	Available in PDAP.
void mouseExited(MouseEvent e)	Available in PDAP.
void mouseMoved(MouseEvent e)	Available in PDAP.
void mousePressed(MouseEvent e)	Available in PDAP.
void mouseReleased(MouseEvent e)	Available in PDAP.
static ActionListener remove (ActionListener l, ActionListener oldl)	Available in PDAP.
static AdjustmentListener remove (AdjustmentListener l,AdjustmentListener oldl)	Available in PDAP.
static ComponentListener remove (ComponentListener l, ComponentListener oldl)	Available in PDAP.
static ContainerListener remove (ContainerListener l, ContainerListener oldl)	Available in PDAP.
protected EventListener remove (EventListener oldl)	Available in PDAP.
static FocusListener remove (FocusListener l, FocusListener oldl)	Available in PDAP.
static HierarchyBoundsListener remove (HierarchyBoundsListener l,	Not available in PDAP.

HierarchyBoundsListener oldl)	
static HierarchyListener remove (HierarchyListener l, HierarchyListener oldl)	Not available in PDAP.
static InputMethodListener remove (InputMethodListener l, InputMethodListener oldl)	Not available in PDAP.
static ItemListener remove (ItemListener l, ItemListener oldl)	Available in PDAP.
static KeyListener remove (KeyListener l, KeyListener oldl)	Available in PDAP.
static MouseListener remove (MouseListener l, MouseListener oldl)	Available in PDAP.
static MouseMotionListener remove (MouseMotionListener l, MouseMotionListener oldl)	Available in PDAP.
static TextListener remove (TextListener l, TextListener oldl)	Available in PDAP.
static WindowListener remove (WindowListener l, WindowListener oldl)	Available in PDAP.
protected static EventListener removeInternal (EventListener l, EventListener oldl)	Available in PDAP.
protected static void save (ObjectOutputStream s, String k, EventListener l)	Not available in PDAP.
protected void saveInternal (ObjectOutputStream s, String k)	Not available in PDAP.
void textValueChanged(TextEvent e)	Available in PDAP.
void windowActivated(WindowEvent e)	Available in PDAP.
void windowClosed(WindowEvent e)	Available in PDAP.
void windowClosing(WindowEvent e)	Available in PDAP.
void windowDeactivated(WindowEvent e)	Available in PDAP.
void windowDeiconified(WindowEvent e)	Available in PDAP.
void windowIconified(WindowEvent e)	Available in PDAP.
void windowOpened(WindowEvent e)	Available in PDAP.

Button

Table B.7. Methods of the Class Button

Method	***Alternative/Workaround***
Button()	Available in PDAP.
Button(String label)	Available in PDAP.
Void addActionListener(ActionListener l)	Available in PDAP.
void addNotify()	Not available in PDAP.
AccessibleContext getAccessibleContext()	Not available in PDAP.
String getActionCommand()	Available in PDAP.
String getLabel()	Available in PDAP.
EventListener[] getListeners(Class listenerType)	Not available in PDAP.
Protected String paramString()	Available in PDAP.
protected void processActionEvent(ActionEvent e)	Available in PDAP.
protected void processEvent(AWTEvent e)	Available in PDAP.
void removeActionListener(ActionListener l)	Available in PDAP.
void setActionCommand(String command)	Available in PDAP.
void setLabel(String label)	Available in PDAP.

Canvas

Table B.8. Methodsof the Class Canvas	
Method	***Alternative/Workaround***
`Canvas()`	Available in PDAP.
Canvas(GraphicsConfiguration config)	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`void paint(Graphics g)`	Available in PDAP.

Checkbox

Table B.9. Methods of the Class Checkbox	
Method	***Alternative/Workaround***
`Checkbox()`	Available in PDAP.
`Checkbox(String label)`	Available in PDAP.
`Checkbox(String label, boolean state)`	Available in PDAP.
`Checkbox(String label, boolean state, CheckboxGroup group)`	Available in PDAP.
`Checkbox(String label, CheckboxGroup group, boolean state)`	Available in PDAP.
`void addItemListener(ItemListener l)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`CheckboxGroup getCheckboxGroup()`	Available in PDAP.
`String getLabel()`	Available in PDAP.
`EventListener[] getListeners(Class listenerType)`	Not available in PDAP.
`Object[] getSelectedObjects()`	Available in PDAP.
`Boolean getState()`	Available in PDAP.
`Protected String paramString()`	Available in PDAP.
`protected void processEvent(AWTEvent e)`	Available in PDAP.
`protected void processItemEvent(ItemEvent e)`	Available in PDAP.
`Void removeItemListener(ItemListener l)`	Available in PDAP.
`Void setCheckboxGroup(CheckboxGroup g)`	Available in PDAP.
`void setLabel(String label)`	Available in PDAP.
`void setState(boolean state)`	Available in PDAP.

CheckboxMenuItem

Table B.10. Methods of the Class CheckboxMenuItem	
Method	***Alternative/Workaround***
`CheckboxMenuItem()`	Available in PDAP.
`CheckboxMenuItem(String label)`	Available in PDAP.
`CheckboxMenuItem(String label, boolean state)`	Available in PDAP.
`Void addItemListener(ItemListener l)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`EventListener[] getListeners(Class listenerType)`	Not available in PDAP.

`Object[] getSelectedObjects()`	Available in PDAP.
`boolean getState()`	Available in PDAP.
`String paramString()`	Available in PDAP.
`Protected void processEvent(AWTEvent e)`	Available in PDAP.
`protected void processItemEvent(ItemEvent e)`	Available in PDAP.
`Void removeItemListener(ItemListener l)`	Available in PDAP.
`void setState(boolean b)`	Available in PDAP.

Choice

Table B.11. Methods of the Class `Choice`

Method	***Alternative/Workaround***
`Choice()`	Available in PDAP.
`void add(String item)`	Available in PDAP.
`void addItem(String item)`	Available in PDAP.
`void addItemListener(ItemListener l)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`int countItems()`	Available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`String getItem(int index)`	Available in PDAP.
`int getItemCount()`	Available in PDAP.
`EventListeners[] getListeners(Class listenerType)`	`Not` available in PDAP.
`int getSelectedIndex()`	Available in PDAP.
`String getSelectedItem()`	Available in PDAP.
`Object[] getSelectedObjects()`	Available in PDAP.
`void insert(String item, int index)`	Available in PDAP.
`protected String paramString()`	Available in PDAP.
`protected void processEvent(AWTEvent e)`	Available in PDAP.
`protected void processItemEvent(ItemEvent e)`	Available in PDAP.
`void remove(int position)`	Available in PDAP.
`void remove(String item)`	Available in PDAP.
`void removeAll()`	Available in PDAP.
`void removeItemListener(ItemListener l)`	Available in PDAP.
`void select(int pos)`	Available in PDAP.
`void select(String str)`	Available in PDAP.

Color

Table B.12. Methods of the Class `Color`

Method	***Alternative/Workaround***
`Color(ColorSpace cspace, float[] components, float alpha)`	Not available in PDAP.
`Color(float r, float g, float b)`	Available in PDAP.
`Color(float r, float g, float b, float a)`	Available in PDAP.
`Color(int rgb)`	Available in PDAP.
`Color(int rgba, boolean hasalpha)`	Available in PDAP.
`Color(int r, int g, int b)`	Available in PDAP.
`Color(int r, int g, int b, int a)`	Available in PDAP.

`Color brighter()`	Available in PDAP.
`PaintContext createContext(ColorModel cm, Rectangle r, Rectangle2D r2d, AffineTransform xform, RenderingHints hints)`	Not available in PDAP.
`Color darker()`	Available in PDAP.
`static Color decode(String nm)`	Available in PDAP.
`boolean equals(Object obj)`	Available in PDAP.
`int getAlpha()`	Available in PDAP.
`int getBlue()`	Available in PDAP.
`static Color getColor (String nm)`	Available in PDAP.
`static Color getColor(String nm, Color v)`	Available in PDAP.
`static Color getColor(String nm, int v)`	Available in PDAP.
`float[] getColorComponents (ColorSpace cspace, float[] compArray)`	Not available in PDAP.
`float[] getColorComponents(float[] compArray)`	Available in PDAP.
`ColorSpace getColorSpace()`	Not available in PDAP.
`float[] getComponents (ColorSpace cspace, float[] compArray)`	Not available in PDAP.
`float[] getComponents(float[] compArray)`	Available in PDAP.
`int getGreen()`	Available in PDAP.
`static Color getHSBColor (float h, float s, float b)`	Available in PDAP.
`int getRed()`	Available in PDAP.
`int getRGB()`	Available in PDAP.
`float[] getRGBColorComponents(float[] compArray)`	Available in PDAP.
`float[] getRGBComponents(float[] compArray)`	Available in PDAP.
`int getTransparency()`	Not available in PDAP.
`int hashCode()`	Available in PDAP.
`static int HSBtoRGB (float hue, float saturation, float brightness)`	Available in PDAP.
`static float[] RGBtoHSB (int r, int g, int b, float[] hsbvals)`	Available in PDAP.
`String toString()`	Available in PDAP.

Component

Table B.13. Methods of the Class Component

Method	*Alternative/Workaround*
`protected Component()`	Available in PDAP.
`boolean action(Event evt, Object what)`	Aavailable in PDAP.
`void add(PopupMenu popup)`	Available in PDAP.
`void addComponentListener(ComponentListener l)`	Available in PDAP.
`void addFocusListener(FocusListener l)`	Available in PDAP.
`void addHierarchyBoundsListener(HierarchyBoundsListener l)`	Not available in PDAP.
`void addHierarchyListener(HierarchyListener l)`	Not available in PDAP.
`void addInputMethodListener(InputMethodListener l)`	Not available in PDAP.
`void addKeyListener(KeyListener l)`	Available in PDAP.
`void addMouseListener(MouseListener l)`	Available in PDAP.

void addMouseMotionListener(MouseMotionListener l)	Available in PDAP.
void addNotify()	Available in PDAP.
void dddPropertyChangeListener(PropertyChangeListener listener)	Not available in PDAP.
void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)	Not available in PDAP.
Rectangle bounds()	Available in PDAP.
int checkImage(Image image, ImageObserver observer)	Available in PDAP.
int checkImage(Image image, int width, int height, ImageObserver observer)	Available in PDAP.
protected AWTEvent coalesceEvents (AWTEvent existingEvent, AWTEvent newEvent)	Available in PDAP.
boolean contains(int x, int y)	Available in PDAP.
boolean contains(Point p)	Available in PDAP.
Image createImage(ImageProducer producer)	Available in PDAP.
Image createImage(int width, int height)	Available in PDAP.
void deliverEvent(Event e)	Available in PDAP.
void disable()	Available in PDAP.
protected void disableEvents(long eventsToDisable)	Available in PDAP.
void dispatchEvent(AWTEvent e)	Available in PDAP.
void doLayout()	Available in PDAP.
void enable()	Available in PDAP.
void enable(boolean b)	Available in PDAP.
protected void enableEvents(long eventsToEnable)	Available in PDAP.
void enableInputMethods(boolean enable)	Not available in PDAP.
protected void firePropertyChange (String propertyName, Object oldValue, Object newValue)	Not available in PDAP.
AccessibleContext getAccessibleContext()	Not available in PDAP.
float getAlignmentX()	Available in PDAP.
float getAlignmentY()	Available in PDAP.
Color getBackground()	Available in PDAP.
Rectangle getBounds()	Available in PDAP.
Rectangle getBounds(Rectangle rv)	Not available in PDAP.
ColorModel getColorModel()	Available in PDAP.
Component getComponentAt(int x, int y)	Available in PDAP.
Component getComponentAt(Point p)	Available in PDAP.
ComponentOrientation getComponentOrientation()	Not available in PDAP.
Cursor getCursor()	Available in PDAP.
DropTarget getDropTarget()	Not available in PDAP.
Font getFont()	Available in PDAP.
FontMetrics getFontMetrics(Font font)	Available in PDAP.
Color getForeground()	Available in PDAP.
Graphics getGraphics()	Available in PDAP.
GraphicsConfiguration getGraphicsConfiguration()	Available in PDAP.
int getHeight()	Available in PDAP.
InputContext getInputContext()	Not available in PDAP.
InputMethodRequests getInputMethodRequests()	Not available in PDAP.
EventListener[] getListeners(Class listenerType)	Not available in PDAP.
Locale getLocale()	Available in PDAP.

`Point getLocation()`	Available in PDAP.
`Point getLocation(Point rv)`	Available in PDAP.
`Point getLocationOnScreen()`	Available in PDAP.
`Dimension getMaximumSize()`	Available in PDAP.
`Dimension getMinimumSize()`	Available in PDAP.
`String getName()`	Available in PDAP.
`Container getParent()`	Available in PDAP.
`java.awt.peer.ComponentPeer getPeer()`	Not available in PDAP (deprecated J2SE method).
`Dimension getPreferredSize()`	Available in PDAP.
`Dimension getSize()`	Available in PDAP.
`Dimension getSize(Dimension rv)`	Available in PDAP.
`Toolkit getToolkit()`	Available in PDAP.
`Object getTreeLock()`	Available in PDAP.
`int getWidth()`	Available in PDAP.
`int getX()`	Available in PDAP.
`int getY()`	Available in PDAP.
`boolean gotFocus(Event evt, Object what)`	Available in PDAP.
`boolean handleEvent(Event evt)`	Available in PDAP.
`boolean hasFocus()`	Available in PDAP.
`void hide()`	Available in PDAP.
`boolean imageUpdate(Image img, int infoflags, int x, int y, int w, int h)`	Available in PDAP.
`boolean inside(int x, int y)`	Available in PDAP.
`void invalidate()`	Available in PDAP.
`boolean isDisplayable()`	Available in PDAP.
`boolean isDoubleBuffered()`	Available in PDAP.
`boolean isEnabled()`	Available in PDAP.
`boolean isFocusTraversable()`	Available in PDAP.
`boolean isLightweight()`	Available in PDAP.
`boolean isOpaque()`	Available in PDAP.
`boolean isShowing()`	Available in PDAP.
`boolean isValid()`	Available in PDAP.
`boolean isVisible()`	Available in PDAP.
`boolean keyDown(Event evt, int key)`	Available in PDAP.
`boolean keyUp(Event evt, int key)`	Available in PDAP.
`void layout()`	Available in PDAP.
`void list()`	Available in PDAP.
`void list(PrintStream out)`	Available in PDAP.
`void list(PrintStream out, int indent)`	Available in PDAP.
`void list(PrintWriter out)`	Available in PDAP.
`void list(PrintWriter out, int indent)`	Available in PDAP.
`Component locate(int x, int y)`	Available in PDAP.
`Point location()`	Available in PDAP.
`boolean lostFocus(Event evt, Object what)`	Available in PDAP.
`Dimension minimumSize()`	Available in PDAP.
`boolean mouseDown(Event evt, int x, int y)`	Available in PDAP.

boolean mouseDrag(Event evt, int x, int y)	Available in PDAP.
boolean mouseEnter(Event evt, int x, int y)	Available in PDAP.
boolean mouseExit(Event evt, int x, int y)	Available in PDAP.
boolean mouseMove(Event evt, int x, int y)	Available in PDAP.
boolean mouseUp(Event evt, int x, int y)	Available in PDAP.
void move(int x, int y)	Available in PDAP.
void nextFocus()	Available in PDAP.
void paint(Graphics g)	Available in PDAP.
void paintAll(Graphics g)	Available in PDAP.
protected String paramString()	Available in PDAP.
boolean postEvent(Event e)	Available in PDAP.
Dimension preferredSize()	Available in PDAP.
boolean prepareImage (Image image, ImageObserver observer)	Available in PDAP.
boolean prepareImage(Image image, int width, int height, ImageObserver observer)	Available in PDAP.
void print(Graphics g)	Available in PDAP.
void printAll(Graphics g)	Available in PDAP.
protected void processComponentEvent (ComponentEvent e)	Available in PDAP.
Protected void processEvent(AWTEvent e)	Available in PDAP.
Protected void processFocusEvent(FocusEvent e)	Available in PDAP.
protected void processHierarchyBoundsEvent (HierarchyEvent e)	Not available in PDAP.
protected void processHierarchyEvent (HierarchyEvent e)	Not available in PDAP.
protected void processInputMethodEvent (InputMethodEvent e)	Not available in PDAP.
Protected void processKeyEvent(KeyEvent e)	Available in PDAP.
Protected void processMouseEvent(MouseEvent e)	Available in PDAP.
Protected void processMouseMotionEvent (MouseEvent e)	Available in PDAP.
void remove(MenuComponent popup)	Available in PDAP.
void removeComponentListener(ComponentListener l)	Available in PDAP.
void removeFocusListener(FocusListener l)	Available in PDAP.
void removeHierarchyBoundsListener(HierarchyBoundsListener l)	Not available in PDAP.
void removeHierarchyListener(HierarchyListener l)	Not available in PDAP.
void removeInputMethodListener(InputMethodListener l)	Not available in PDAP.
void removeKeyListener(KeyListener l)	Available in PDAP.
void removeMouseListener(MouseListener l)	Available in PDAP.
void removeMouseMotionListener(MouseMotionListener l)	Available in PDAP.
void removeNotify()	Available in PDAP.
void removePropertyChangeListener(PropertyChangeListener listener)	Not available in PDAP.
void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)	Not available in PDAP.
void repaint()	Available in PDAP.
void repaint(int x, int y, int width, int height)	Available in PDAP.

`void repaint(long tm)`	Available in PDAP.
`void repaint(long tm, int x, int y, int width, int height)`	Available in PDAP.
`void requestFocus()`	Available in PDAP.
`void reshape(int x, int y, int width, int height)`	Available in PDAP.
`void resize(Dimension d)`	Available in PDAP.
`void resize(int width, int height)`	Available in PDAP.
`void setBackground(Color c)`	Available in PDAP.
`void setBounds(int x, int y, int width, int height)`	Available in PDAP.
`void setBounds(Rectangle r)`	Available in PDAP.
`void setComponentOrientation (ComponentOrientation o)`	Not available in PDAP.
`void setCursor(Cursor cursor)`	Available in PDAP.
`void setDropTarget(DropTarget dt)`	Not available in PDAP.
`void setEnabled(boolean b)`	Available in PDAP.
`void setFont(Font f)`	Available in PDAP.
`void setForeground(Color c)`	Available in PDAP.
`void setLocale(Locale l)`	Available in PDAP.
`void setLocation(int x, int y)`	Available in PDAP.
`void setLocation(Point p)`	Available in PDAP.
`void setName(String name)`	Available in PDAP.
`void setSize(Dimension d)`	Available in PDAP.
`void setSize(int width, int height)`	Available in PDAP.
`void setVisible(boolean b)`	Available in PDAP.
`void show()`	Available in PDAP.
`void show(boolean b)`	Available in PDAP.
`Dimension size()`	Available in PDAP.
`String toString()`	Available in PDAP.
`void transferFocus()`	Available in PDAP.
`void update(Graphics g)`	Available in PDAP.
`void validate()`	Available in PDAP.

`Container`

Table B.14. Methods of the Class `Container`

Method	***Alternative/Workaround***
`Container()`	Available in PDAP.
`Component add(Component comp)`	Available in PDAP.
`Component add(Component comp, int index)`	Available in PDAP.
`void add(Component comp, Object constraints)`	Available in PDAP.
`void add(Component comp, Object constraints, int index)`	Available in PDAP.
`Component add(String name, Component comp)`	Available in PDAP.
`void addContainerListener(ContainerListener l)`	Available in PDAP.
`protected void addImpl(Component comp, Object constraints, int index)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`int countComponents()`	Available in PDAP.
`void deliverEvent(Event e)`	Available in PDAP.

`void doLayout()`	Available in PDAP.
`Component findComponentAt(int x, int y)`	Available in PDAP.
`Component findComponentAt(Point p)`	Available in PDAP.
`float getAlignmentX()`	Available in PDAP.
`float getAlignmentY()`	Available in PDAP.
`Component getComponent(int n)`	Available in PDAP.
`Component getComponentAt(int x, int y)`	Available in PDAP.
`Component getComponentAt(Point p)`	Available in PDAP.
`int getComponentCount()`	Available in PDAP.
`Component[] getComponents()`	Available in PDAP.
`Insets getInsets()`	Available in PDAP.
`LayoutManager getLayout()`	Available in PDAP.
`EventListener[] getListeners(Class listenerType)`	Not available in PDAP.
`Dimension getMaximumSize()`	Available in PDAP.
`Dimension getMinimumSize()`	Available in PDAP.
`Dimension getPreferredSize()`	Available in PDAP.
`Insets insets()`	Available in PDAP.
`void invalidate()`	Available in PDAP.
`boolean isAncestorOf(Component c)`	Available in PDAP.
`void layout()`	Available in PDAP.
`void list(PrintStream out, int indent)`	Available in PDAP.
`void list(PrintWriter out, int indent)`	Available in PDAP.
`Component locate(int x, int y)`	Available in PDAP.
`Dimension minimumSize()`	Available in PDAP.
`void paint(Graphics g)`	Available in PDAP.
`void paintComponents(Graphics g)`	Available in PDAP.
`protected String paramString()`	Available in PDAP.
`Dimension preferredSize()`	Available in PDAP.
`void print(Graphics g)`	Available in PDAP.
`void printComponents(Graphics g)`	Available in PDAP.
`protected void processContainerEvent (ContainerEvent e)`	Available in PDAP.
`protected void processEvent(AWTEvent e)`	Available in PDAP.
`void remove(Component comp)`	Available in PDAP.
`void remove(int index)`	Available in PDAP.
`void removeAll()`	Available in PDAP.
`void removeContainerListener(ContainerListener l)`	Available in PDAP.
`void removeNotify()`	Not available in PDAP.
`void setFont(Font f)`	Available in PDAP.
`void setLayout(LayoutManager mgr)`	Available in PDAP.
`void update(Graphics g)`	Available in PDAP.
`void validate()`	Available in PDAP.
`protected void validateTree()`	Available in PDAP.

`Cursor`

Table B.15. Methods of the Class `Cursor`

Method	***Alternative/Workaround***

`Cursor(int type)`	Available in PDAP.
`protected Cursor(String name)`	Not available in PDAP.
`protected void finalize()`	Available in PDAP.
`static Cursor getDefaultCursor()`	Available in PDAP.
`String getName()`	Not available in PDAP.
`static Cursor getPredefinedCursor(int type)`	Not available in PDAP.
`static Cursor getSystemCustomCursor(String name)`	Not available in PDAP.
`int getType()`	Available in PDAP.
`String toString()`	Not available in PDAP.

Dialog

Table B.16. Methods of the Class Dialog

Method	*Alternative/Workaround*
`Dialog(Dialog owner)`	Not available in PDAP.
`Dialog(Dialog owner, String title)`	Not available in PDAP.
`Dialog(Dialog owner, String title, boolean modal)`	Not available in PDAP.
`Dialog(Frame owner)`	Available in PDAP.
`Dialog(Frame owner, boolean modal)`	Available in PDAP.
`Dialog(Frame owner, String title)`	Available in PDAP.
`Dialog(Frame owner, String title, boolean modal)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`void dispose()`	Available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`String getTitle()`	Available in PDAP.
`void hide()`	Available in PDAP.
`boolean isModal()`	Available in PDAP.
`boolean isResizable()`	Available in PDAP.
`protected String paramString()`	Available in PDAP.
`void setModal(boolean b)`	Available in PDAP.
`void setResizable(boolean resizable)`	Available in PDAP.
`void setTitle(String title)`	Available in PDAP.
`void show()`	Available in PDAP.

Dimension

Table B.17. Methods of the Class Dimension

Method	*Alternative/Workaround*
`Dimension()`	Available in PDAP.
`Dimension(Dimension d)`	Available in PDAP.
`Dimension(int width, int height)`	Available in PDAP.
`boolean equals(Object obj)`	Available in PDAP.
`double getHeight()`	Not available in PDAP.
`Dimension getSize()`	Available in PDAP.
`double getWidth()`	Not available in PDAP.
`int hashCode()`	Available in PDAP.
`void setSize(Dimension d)`	Available in PDAP.
`void setSize(double width, double height)`	Not available in PDAP.

`void setSize(int width, int height)`	Available in PDAP.
`String toString()`	Available in PDAP.

`FileDialog`

Table B.18. Methods of the Class `FileDialog`

Method	***Alternative/Workaround***
`FileDialog(Frame parent)`	Available in PDAP.
`FileDialog(Frame parent, String title)`	Available in PDAP.
`FileDialog(Frame parent, String title, int mode)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`String getDirectory()`	Available in PDAP.
`String getFile()`	Available in PDAP.
`FilenameFilter getFilenameFilter()`	Not available in PDAP.
`int getMode()`	Available in PDAP.
`protected String paramString()`	Available in PDAP.
`void setDirectory(String dir)`	Available in PDAP.
`void setFile(String file)`	Available in PDAP.
`void setFilenameFilter(FilenameFilter filter)`	Not available in PDAP.
`void setMode(int mode)`	Available in PDAP.

`Font`

Table B.19. Methods of the Class Font

Method	***Alternative/Workaround***
`Font(Map attributes)`	Not available in PDAP.
`Font(String name, int style, int size)`	Available in PDAP.
`boolean canDisplay(char c)`	Not available in PDAP.
`int canDisplayUpTo (char[] text, int start, int limit)`	Not available in PDAP.
`int canDisplayUpTo (CharacterIterator iter, int start, int limit)`	Not available in PDAP.
`int canDisplayUpTo(String str)`	Not available in PDAP.
`static Font createFont (int fontFormat, InputStream fontStream)`	Not available in PDAP.
`GlyphVector createGlyphVector (FontRenderContext frc, char[] chars)`	Not available in PDAP.
`GlyphVector createGlyphVector (FontRenderContext frc, CharacterIterator ci)`	Not available in PDAP.
`GlyphVector createGlyphVector (FontRenderContext frc, int[] glyphCodes)`	Not available in PDAP.
`GlyphVector createGlyphVector (FontRenderContext frc, String str)`	Not available in PDAP.
`static Font decode(String str)`	Available in PDAP.
`Font deriveFont(AffineTransform trans)`	Not available in PDAP.
`Font deriveFont(float size)`	Not available in PDAP.
`Font deriveFont(int style)`	Not available in PDAP.
`Font deriveFont(int style, AffineTransform trans)`	Not available in PDAP.
`Font deriveFont(int style, float size)`	Not available in PDAP.

`Font deriveFont(Map attributes)`	Not available in PDAP.
`boolean equals(Object obj)`	Available in PDAP.
`protected void finalize()`	Not available in PDAP.
`Map getAttributes()` `AttributedCharacterIterator.Attribute[]`	Not available in PDAP.
`getAvailableAttributes()`	Not available in PDAP.
`byte getBaselineFor(char c)`	Not available in PDAP.
`String getFamily()`	Available in PDAP.
`String getFamily(Locale l)`	Not available in PDAP.
`static Font getFont(Map attributes)`	Not available in PDAP.
`static Font getFont(String nm)`	Available in PDAP.
`static Font getFont(String nm, Font font)`	Available in PDAP.
`String getFontName()`	Not available in PDAP.
`String getFontName(Locale l)`	Not available in PDAP.
`float getItalicAngle()`	Not available in PDAP.
`LineMetrics getLineMetrics(char[] chars, int beginIndex, int limit, FontRenderContext frc)`	Not available in PDAP.
`LineMetrics getLineMetrics(CharacterIterator ci, int beginIndex, int limit, FontRenderContext frc)`	Not available in PDAP.
`LineMetrics getLineMetrics (String str, FontRenderContext frc)`	Not available in PDAP.
`LineMetrics getLineMetrics(String str, int beginIndex, int limit, FontRenderContext frc)`	Not available in PDAP.
`Rectangle2D getMaxCharBounds(FontRenderContext frc)`	Not available in PDAP.
`int getMissingGlyphCode()`	Not available in PDAP.
`String getName()`	Available in PDAP.
`int getNumGlyphs()`	Not available in PDAP.
`java.awt.peer.FontPeer getPeer()`	Not available in PDAP (deprecated J2SE method).
`String getPSName()`	Not available in PDAP.
`int getSize()`	Available in PDAP.
`float getSize2D()`	Not available in PDAP.
`Rectangle2D getStringBounds(char[] chars, int beginIndex, int limit, FontRenderContext frc)`	Not available in PDAP.
`Rectangle2D getStringBounds(CharacterIterator ci, int beginIndex, int limit, FontRenderContext frc)`	Not available in PDAP.
`Rectangle2D getStringBounds String str, FontRenderContext frc)`	Not available in PDAP.
`Rectangle2D getStringBounds(String str, int beginIndex, int limit, FontRenderContext frc)`	Not available in PDAP.
`int getStyle()`	Available in PDAP.
`AffineTransform getTransform()`	Not available in PDAP.
`int hashCode()`	Available in PDAP.
`boolean hasUniformLineMetrics()`	Not available in PDAP.
`boolean isBold()`	Available in PDAP.
`boolean isItalic()`	Available in PDAP.
`boolean isPlain()`	Available in PDAP.

`String toString()`	Not available in PDAP.

FontMetrics

Table B.20. Methods of the Class FontMetrics

Method	*Alternative/Workaround*
`protected FontMetrics(Font font)`	Available in PDAP.
`int bytesWidth(byte[] data, int off, int len)`	Available in PDAP.
`int charsWidth(char[] data, int off, int len)`	Available in PDAP.
`int charWidth(char ch)`	Available in PDAP.
`int charWidth(int ch)`	Available in PDAP.
`int getAscent()`	Available in PDAP.
`int getDescent()`	Available in PDAP.
`Font getFont()`	Available in PDAP.
`int getHeight()`	Available in PDAP.
`int getLeading()`	Available in PDAP.
`LineMetrics getLineMetrics(char[] chars, int beginIndex, int limit, Graphics context)`	Not available in PDAP.
`LineMetrics getLineMetrics(CharacterIterator ci, int beginIndex, int limit, Graphics context)`	Not available in PDAP.
`LineMetrics getLineMetrics (String str, Graphics context)`	Not available in PDAP.
`LineMetrics getLineMetrics(String str, int beginIndex, int limit, Graphics context)`	Not available in PDAP.
`int getMaxAdvance()`	Available in PDAP.
`int getMaxAscent()`	Available in PDAP.
`Rectangle2D getMaxCharBounds(Graphics context)`	Not available in PDAP.
`int getMaxDecent()`	Available in PDAP.
`int getMaxDescent()`	Available in PDAP.
`Rectangle2D getStringBounds(char[] chars, int beginIndex, int limit, Graphics context)`	Not available in PDAP.
`Rectangle2D getStringBounds(CharacterIterator ci, int beginIndex, int limit, Graphics context)`	Not available in PDAP.
`Rectangle2D getStringBounds (String str, Graphics context)`	Not available in PDAP.
`Rectangle2D getStringBounds(String str, int beginIndex, int limit, Graphics context)`	Not available in PDAP.
`int[] getWidths()`	Available in PDAP.
`boolean hasUniformLineMetrics()`	Not available in PDAP.
`int stringWidth(String str)`	Available in PDAP.
`String toString()`	Available in PDAP.

Frame

Table B.21. Methods of the Class Frame

Method	*Alternative/Workaround*
`Frame()`	Available in PDAP.
`Frame(GraphicsConfiguration gc)`	Available in PDAP.
`Frame(String title)`	Available in PDAP.
`Frame(String title, GraphicsConfiguration gc)`	Available in PDAP.

`void addNotify()`	Not available in PDAP.
`protected void finalize()`	Available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`int getCursorType()`	Available in PDAP.
`static Frame[] getFrames()`	Not available in PDAP.
`Image getIconImage()`	Available in PDAP.
`MenuBar getMenuBar()`	Available in PDAP.
`int getState()`	Available in PDAP.
`String getTitle()`	Available in PDAP.
`boolean isResizable()`	Available in PDAP.
`protected String paramString()`	Available in PDAP.
`void remove(MenuComponent m)`	Available in PDAP.
`void removeNotify()`	Not available in PDAP.
`void setCursor(int cursorType)`	Available in PDAP.
`void setIconImage(Image image)`	Available in PDAP.
`void setMenuBar(MenuBar mb)`	Available in PDAP.
`void setResizable(boolean resizable)`	Available in PDAP.
`void setState(int state)`	Available in PDAP.
`void setTitle(String title)`	Available in PDAP.

Graphics

Table B.22. Methods of the Class Graphics

Method	*Alternative/Workaround*
`protected Graphics()`	Available in PDAP.
`abstract void clearRect(int x, int y, int width, int height)`	Available in PDAP.
`abstract void clipRect(int x, int y, int width, int height)`	Available in PDAP.
`abstract void copyArea(int x, int y, int width, int height, int dx, int dy)`	Available in PDAP.
`Abstract Graphics create()`	Available in PDAP.
`Graphics create(int x, int y, int width, int height)`	Available in PDAP.
`abstract void dispose()`	Available in PDAP.
`void draw3DRect(int x, int y, int width, int height, boolean raised)`	Available in PDAP.
`abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	Available in PDAP.
`void drawBytes(byte[] data, int offset, int length, int x, int y)`	Available in PDAP.
`void drawChars(char[] data, int offset, int length, int x, int y)`	Available in PDAP.
`abstract boolean drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer)`	Available in PDAP.
`abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)`	Available in PDAP.
`abstract boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)`	Available in PDAP.

abstract boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)	Available in PDAP.
abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor, ImageObserver observer)	Available in PDAP.
abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer)	Available in PDAP.
abstract void drawLine(int x1, int y1, int x2, int y2)	Available in PDAP.
abstract void drawOval(int x, int y, int width, int height)	Available in PDAP.
abstract void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)	Available in PDAP.
void drawPolygon(Polygon p)	Available in PDAP.
abstract void drawPolyline(int[] xPoints, int[] yPoints, int nPoints)	Available in PDAP.
void drawRect(int x, int y, int width, int height)	Available in PDAP.
abstract void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)	Available in PDAP.
abstract void drawString (AttributedCharacterIterator iterator, int x, int y)	Not available in PDAP.
abstract void drawString(String str, int x, int y)	Available in PDAP.
void fill3DRect(int x, int y, int width, int height, boolean raised)	Available in PDAP.
abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)	Available in PDAP.
abstract void fillOval(int x, int y, int width, int height)	Available in PDAP.
abstract void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)	Available in PDAP.
void fillPolygon(Polygon p)	Available in PDAP.
abstract void fillRect(int x, int y, int width, int height)	Available in PDAP.
abstract void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)	Available in PDAP.
void finalize()	Available in PDAP.
abstract Shape getClip()	Available in PDAP.
abstract Rectangle getClipBounds()	Available in PDAP.
Rectangle getClipBounds(Rectangle r)	Available in PDAP.
Rectangle getClipRect()	Available in PDAP.
abstract Color getColor()	Available in PDAP.
abstract Font getFont()	Available in PDAP.
FontMetrics getFontMetrics()	Available in PDAP.
abstract FontMetrics getFontMetrics(Font f)	Available in PDAP.
boolean hitClip(int x, int y, int width, int height)	Available in PDAP.
abstract void setClip(int x, int y, int width, int height)	Available in PDAP.

abstract void setClip(Shape clip)	Available in PDAP.
abstract void setColor(Color c)	Available in PDAP.
abstract void setFont(Font font)	Available in PDAP.
abstract void setPaintMode()	Available in PDAP.
abstract void setXORMode(Color c1)	Available in PDAP.
String toString()	Available in PDAP.
abstract void translate(int x, int y)	Available in PDAP.

GraphicsConfiguration

Table B.23. Class GraphicsConfiguration

Method	*Alternative/Workaround*
Protected GraphicsConfiguration()	Available in PDAP.
abstract BufferedImage createCompatibleImage (int width, int height)	Not available in PDAP.
abstract BufferedImage createCompatibleImage (int width, int height, int transparency)	Not available in PDAP.
abstract Rectangle getBounds()	Available in PDAP.
abstract ColorModel getColorModel()	Available in PDAP.
abstract ColorModel getColorModel (int transparency)	Not available in PDAP.
abstract AffineTransform getDefaultTransform()	Not available in PDAP.
abstract GraphicsDevice getDevice()	Available in PDAP.
abstract AffineTransform getNormalizingTransform()	Not available in PDAP.

GraphicsDevice

Table B.24. Class GraphicsDevice

Method	*Alternative/Workaround*
protected GraphicsDevice()	Available in PDAP.
GraphicsConfiguration getBestConfiguration (GraphicsConfigTemplate gct)	Not available in PDAP.
abstract GraphicsConfiguration[] getConfigurations()	Available in PDAP.
abstract GraphicsConfiguration getDefaultConfiguration()	Available in PDAP.
abstract String getIDstring()	Available in PDAP.
abstract int getType()	Available in PDAP.

GraphicsEnvironment

Table B.25. Class GraphicsEnvironment

Method	*Alternative/Workaround*
Protected GraphicsEnvironment()	Available in PDAP.
abstract Graphics2D createGraphics (BufferedImage img)	Not available in PDAP.
abstract Font[] getAllFonts()	Not available in PDAP.
abstract String[] getAvailableFontFamilyNames()	Available in PDAP.
abstract String[] getAvailableFontFamilyNames	Available in PDAP.

`(Locale l)`	
`abstract GraphicsDevice getDefaultScreenDevice()`	Available in PDAP.
`static GraphicsEnvironment getLocalGraphicsEnvironment()`	Available in PDAP.
`abstract GraphicsDevice[] getScreenDevices()`	Available in PDAP.

GridBagConstraints

Table B.26. Methods of the Class `GridBagConstraints`

Method	***Alternative/Workaround***
`GridBagConstraints()`	Available in PDAP.
`GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady)`	Not available in PDAP.
`Object clone()`	Not available in PDAP.

GridBagLayout

Table B.27. Methods of the Class `GridBagLayout`

Method	***Alternative/Workaround***
`GridBagLayout()`	Available in PDAP.
`void addLayoutComponent (Component comp, Object constraints)`	Available in PDAP.
`void addLayoutComponent (String name, Component comp)`	Available in PDAP.
`protected void AdjustForGravity (GridBagConstraints constraints, Rectangle r)`	Available in PDAP.
`protected void ArrangeGrid(Container parent)`	Available in PDAP.
`GridBagConstraints getConstraints (Component comp)`	Available in PDAP.
`float getLayoutAlignmentX(Container parent)`	Available in PDAP.
`float getLayoutAlignmentY(Container parent)`	Available in PDAP.
`int[][] getLayoutDimensions()`	Available in PDAP.
`protected ava.awt.GridBagLayoutInfo GetLayoutInfo(Container parent, int sizeflag)`	Not available in PDAP.
`Point getLayoutOrigin()`	Available in PDAP.
`double[][] getLayoutWeights()`	Available in PDAP.
`protected Dimension GetMinSize (Container parent, java.awt.GridBagLayoutInfo info)`	Not available in PDAP.
`void invalidateLayout(Container target)`	Available in PDAP.
`void layoutContainer(Container parent)`	Available in PDAP.
`Point location(int x, int y)`	Available in PDAP.
`protected GridBagConstraints lookupConstraints (Component comp)`	Available in PDAP.
`Dimension maximumLayoutSize(Container target)`	Available in PDAP.
`Dimension`	
`minimumLayoutSize(Container parent)`	Available in PDAP.
`Dimension preferredLayoutSize(Container parent)`	Available in PDAP.
`void removeLayoutComponent(Component comp)`	Available in PDAP.

`void setConstraints(Component comp, GridBagConstraints constraints)`	Available in PDAP.
`String toString()`	Available in PDAP.

`Insets`

Table B.28. Methods of the Class `Insets`

Method	*Alternative/Workaround*
`Insets(int top, int left, int bottom, int right)`	Available in PDAP.
`Object clone()`	Not available in PDAP.
`boolean equals(Object obj)`	Available in PDAP.
`int hashCode()`	Available in PDAP.
`String toString()`	Available in PDAP.

`Label`

Table B.29. Methods of the Class `Label`

Method	*Alternative/Workaround*
`Label()`	Available in PDAP.
`Label(String text)`	Available in PDAP.
`Label(String text, int alignment)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`int getAlignment()`	Available in PDAP.
`String getText()`	Available in PDAP.
`protected String paramString()`	Available in PDAP.
`void setAlignment(int alignment)`	Available in PDAP.
`void setText(String text)`	Available in PDAP.

`List`

Table B.30. Methods of the Class `List`

Method	*Alternative/Workaround*
`List()`	Available in PDAP.
`List(int rows)`	Available in PDAP.
`List(int rows, Boolean multipleMode)`	Available in PDAP.
`void add(String item)`	Available in PDAP.
`void add(String item, int index)`	Available in PDAP.
`void addActionListener(ActionListener l)`	Available in PDAP.
`void addItem(String item)`	Available in PDAP.
`void addItem(String item, int index)`	Available in PDAP.
`void addItemListener(ItemListener l)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`boolean allowsMultipleSelections()`	Available in PDAP.
`void clear()`	Available in PDAP.
`int countItems()`	Available in PDAP.
`void delItem(int position)`	Available in PDAP.
`void delItems(int start, int end)`	Available in PDAP.

void deselect(int index)	Available in PDAP.
AccessibleContext getAccessibleContext()	Not available in PDAP.
String getItem(int index)	Available in PDAP.
int getItemCount()	Available in PDAP.
String[] getItems()	Available in PDAP.
EventListener[] getListeners(Class listenerType)	Not available in PDAP.
Dimension getMinimumSize()	Available in PDAP.
Dimension getMinimumSize(int rows)	Available in PDAP.
Dimension getPreferredSize()	Available in PDAP.
Dimension getPreferredSize(int rows)	Available in PDAP.
int getRows()	Available in PDAP.
int getSelectedIndex()	Available in PDAP.
int[] getSelectedIndexes()	Available in PDAP.
String getSelectedItem()	Available in PDAP.
String[] getSelectedItems()	Available in PDAP.
Object[] getSelectedObjects()	Available in PDAP.
int getVisibleIndex()	Available in PDAP.
boolean isIndexSelected(int index)	Available in PDAP.
boolean isMultipleMode()	Available in PDAP.
boolean isSelected(int index)	Available in PDAP.
void makeVisible(int index)	Available in PDAP.
Dimension minimumSize()	Available in PDAP.
Dimension minimumSize(int rows)	Available in PDAP.
protected String paramString()	Available in PDAP.
Dimension preferredSize()	Available in PDAP.
Dimension preferredSize(int rows)	Available in PDAP.
protected void processActionEvent(ActionEvent e)	Available in PDAP.
protected void processEvent(AWTEvent e)	Available in PDAP.
protected void processItemEvent(ItemEvent e)	Available in PDAP.
void remove(int position)	Available in PDAP.
void remove(String item)	Available in PDAP.
void removeActionListener(ActionListener l)	Available in PDAP.
void removeAll()	Available in PDAP.
void removeItemListener(ItemListener l)	Available in PDAP.
void removeNotify()	Not available in PDAP.
void replaceItem(String newValue, int index)	Available in PDAP.
void select(int index)	Available in PDAP.
void setMultipleMode(boolean b)	Available in PDAP.
void setMultipleSelections(boolean b)	Available in PDAP.

Menu

Table B.31. Methods of the Class Menu

Method	*Alternative/Workaround*
Menu()	Available in PDAP.
Menu(String label)	Available in PDAP.
Menu(String label, Boolean tearOff)	Available in PDAP.
MenuItem add(MenuItem mi)	Available in PDAP.

void add(String label)	Available in PDAP.
void addNotify()	Not available in PDAP.
void addSeparator()	Available in PDAP.
int countItems()	Available in PDAP.
AccessibleContext getAccessibleContext()	Not available in PDAP.
MenuItem getItem(int index)	Available in PDAP.
int getItemCount()	Available in PDAP.
void insert(MenuItem menuitem, int index)	Available in PDAP.
void insert(String label, int index)	Available in PDAP.
void insertSeparator(int index)	Available in PDAP.
boolean isTearOff()	Available in PDAP.
String paramString()	Available in PDAP.
void remove(int index)	Available in PDAP.
void remove(MenuComponent item)	Available in PDAP.
void removeAll()	Available in PDAP.
void removeNotify()	Not available in PDAP.

MenuBar

Table B.32. Methods of the Class MenuBar

Method	*Alternative/Workaround*
MenuBar()	Available in PDAP.
Menu add(Menu m)	Available in PDAP.
void addNotify()	Not available in PDAP.
int countMenus()	Available in PDAP.
void deleteShortcut(MenuShortcut s)	Available in PDAP.
AccessibleContext getAccessibleContext()	Not available in PDAP.
Menu getHelpMenu()	Available in PDAP.
Menu getMenu(int i)	Available in PDAP.
int getMenuCount()	Available in PDAP.
MenuItem getShortcutMenuItem(MenuShortcut s)	Available in PDAP.
void remove(int index)	Available in PDAP.
void remove(MenuComponent m)	Available in PDAP.
void removeNotify()	Not available in PDAP.
void setHelpMenu(Menu m)	Available in PDAP.
Enumeration shortcuts()	Available in PDAP.

MenuComponent

Table B.33. Methods of the Class MenuComponent

Method	*Alternative/Workaround*
MenuComponent()	Available in PDAP.
void dispatchEvent(AWTEvent e)	Available in PDAP.
AccessibleContext getAccessibleContext()	Not available in PDAP.
Font getFont()	Available in PDAP.
String getName()	Available in PDAP.
MenuContainer getParent()	Available in PDAP.

java.awt.peer.MenuComponentPeer	
getPeer()	Not available in PDAP (deprecated J2SE method).
protected Object Object getTreeLock()	Available in PDAP.
protected String paramString()	Available in PDAP.
boolean postEvent(Event evt)	Available in PDAP.
protected void processEvent(AWTEvent e)	Available in PDAP.
void removeNotify()	Not available in PDAP.
void setFont(Font f)	Available in PDAP.
void setName(String name)	Available in PDAP.
String toString()	Available in PDAP.

MenuItem

Table B.34. Methods of the Class MenuItem

Method	*Alternative/Workaround*
MenuItem()	Available in PDAP.
MenuItem(String label)	Available in PDAP.
MenuItem(String label, MenuShortcut s)	Available in PDAP.
void addActionListener(ActionListener l)	Available in PDAP.
void addNotify()	Not available in PDAP.
void deleteShortcut()	Available in PDAP.
void disable()	Available in PDAP.
protected void disableEvents (long eventsToDisable)	Available in PDAP.
void enable()	Available in PDAP.
void enable(boolean b)	Available in PDAP.
protected void enableEvents(long eventsToEnable)	Available in PDAP.
AccessibleContext getAccessibleContext()	Not available in PDAP.
String getActionCommand()	Available in PDAP.
String getLabel()	Available in PDAP.
EventListener[] getListeners(Class listenerType)	Not available in PDAP.
MenuShortcut getShortcut()	Available in PDAP.
boolean isEnabled()	Available in PDAP.
String paramString()	Available in PDAP.
protected void processActionEvent(ActionEvent e)	Available in PDAP.
protected void processEvent(AWTEvent e)	Available in PDAP.
void removeActionListener(ActionListener l)	Available in PDAP.
void setActionCommand(String command)	Available in PDAP.
void setEnabled(boolean b)	Available in PDAP.
void setLabel(String label)	Available in PDAP.
void setShortcut(MenuShortcut s)	Available in PDAP.

Panel

Table B.35. Methods of the Class Panel

Method	*Alternative/Workaround*

Panel()	Available in PDAP.
Panel(LayoutManager layout)	Available in PDAP.
void addNotify()	Not available in PDAP.
AccessibleContext getAccessibleContext()	Not available in PDAP.

Point

Table B.36. Methods of the Class Point

Method	*Alternative/Workaround*
Point()	Available in PDAP.
Point(int x, int y)	Available in PDAP.
Point(Point p)	Available in PDAP.
Boolean equals(Object obj)	Available in PDAP.
Point getLocation()	Available in PDAP.
double getX()	Not available in PDAP.
double getY()	Not available in PDAP.
void move(int x, int y)	Available in PDAP.
void setLocation(double x, double y)	Not available in PDAP.
void setLocation(int x, int y)	Not available in PDAP.
void setLocation(Point p)	Available in PDAP.
String toString()	Available in PDAP.
void translate(int x, int y	Available in PDAP.

Polygon

Table B.37. Methods of the Class Polygon

Method	*Alternative/Workaround*
Polygon()	Available in PDAP.
Polygon (int[] xpoints, int[] ypoints, int npoints)	Available in PDAP.
void addPoint(int x, int y)	Available in PDAP.
boolean contains(double x, double y)	Not available in PDAP.
boolean contains (double x, double y, double w, double h)	Not available in PDAP.
boolean contains(int x, int y)	Available in PDAP.
boolean contains(Point p)	Available in PDAP.
boolean contains(Point2D p)	Not available in PDAP.
boolean contains(Rectangle2D r)	Not available in PDAP.
Rectangle getBoundingBox()	Available in PDAP.
Rectangle getBounds()	Available in PDAP.
Rectangle2D getBounds2D()	Not available in PDAP.
PathIterator getPathIterator(AffineTransform at)	Not available in PDAP.
PathIterator getPathIterator (AffineTransform at, double flatness)	Not available in PDAP.
boolean inside(int x, int y)	Available in PDAP.
boolean intersects (double x, double y, double w, double h)	Not available in PDAP.
boolean intersects(Rectangle2D r)	Not available in PDAP.
void translate(int deltaX, int deltaY)	Available in PDAP.

PopupMenu

Table B.38. Methods of the Class PopupMenu	
Method	***Alternative/Workaround***
`PopupMenu()`	Available in PDAP.
`PopupMenu(String label)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`void show(Component origin, int x, int y)`	Available in PDAP.

Rectangle

Table B.39. Methods of the Class Rectangle	
Method	***Alternative/Workaround***
`Rectangle()`	Available in PDAP.
`Rectangle(Dimension d)`	Available in PDAP.
`Rectangle(int width, int height)`	Available in PDAP.
`Rectangle(int x, int y, int width, int height)`	Available in PDAP.
`Rectangle(Point p)`	Available in PDAP.
`Rectangle(Point p, Dimension d)`	Available in PDAP.
`Rectangle(Rectangle r)`	Available in PDAP.
`void add(int newx, int newy)`	Available in PDAP.
`void add(Point pt)`	Available in PDAP.
`void add(Rectangle r)`	Available in PDAP.
`boolean contains(int x, int y)`	Available in PDAP.
`boolean contains(int X, int Y, int W, int H)`	Available in PDAP.
`boolean contains(Point p)`	Available in PDAP.
`boolean contains(Rectangle r)`	Available in PDAP.
`Rectangle2D createIntersection(Rectangle2D r)`	Not available in PDAP.
`Rectangle2D createUnion(Rectangle2D r)`	Not available in PDAP.
`boolean equals(Object obj)`	Available in PDAP.
`Rectangle getBounds()`	Available in PDAP.
`Rectangle2D getBounds2D()`	Not available in PDAP.
`double getHeight()`	Not available in PDAP.
`Point getLocation()`	Available in PDAP.
`Dimension getSize()`	Available in PDAP.
`double getWidth()`	Not available in PDAP.
`double getX()`	Not available in PDAP.
`double getY()`	Not available in PDAP.
`void grow(int h, int v)`	Available in PDAP.
`boolean inside(int x, int y)`	Available in PDAP.
`Rectangle intersection(Rectangle r)`	Available in PDAP.
`boolean intersects(Rectangle r)`	Available in PDAP.
`boolean isEmpty()`	Available in PDAP.
`void move(int x, int y)`	Available in PDAP.
`int outcode(double x, double y)`	Not available in PDAP.
`void reshape(int x, int y, int width, int height)`	Available in PDAP.
`void resize(int width, int height)`	Available in PDAP.

`void setBounds(int x, int y, int width, int height)`	Available in PDAP.
`void setBounds(Rectangle r)`	Available in PDAP.
`void setLocation(int x, int y)`	Available in PDAP.
`void setLocation(Point p)`	Available in PDAP.
`void setRect(double x, double y, double width, double height)`	Not available in PDAP.
`void setSize(Dimension d)`	Available in PDAP.
`void setSize(int width, int height)`	Available in PDAP.
`String toString()`	Available in PDAP.
`void translate(int x, int y)`	Available in PDAP.
`Rectangle union(Rectangle r)`	Available in PDAP.

`Scrollbar`

Table B.40. Methods of the Class `Scrollbar`

Method	***Alternative/Workaround***
`Scrollbar()`	Available in PDAP.
`Scrollbar(int orientation)`	Available in PDAP.
`Scrollbar(int orientation, int value, int visible, int minimum, int maximum)`	Available in PDAP.
`void addAdjustmentListener(AdjustmentListener l)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`int getBlockIncrement()`	Available in PDAP.
`int getLineIncrement()`	Available in PDAP.
`EventListener[] getListeners(Class listenerType)`	Not available in PDAP.
`int getMaximum()`	Available in PDAP.
`int getMinimum()`	Available in PDAP.
`int getOrientation()`	Available in PDAP.
`int getPageIncrement()`	Available in PDAP.
`int getUnitIncrement()`	Available in PDAP.
`int getValue()`	Available in PDAP.
`int getVisible()`	Available in PDAP.
`int getVisibleAmount()`	Not available in PDAP.
`protected String paramString()`	Available in PDAP.
`protected void processAdjustmentEvent (AdjustmentEvent e)`	Available in PDAP.
`protected void processEvent(AWTEvent e)`	Available in PDAP.
`void removeAdjustmentListener(AdjustmentListener l)`	Available in PDAP.
`void setBlockIncrement(int v)`	Available in PDAP.
`void setLineIncrement(int v)`	Available in PDAP.
`void setMaximum(int newMaximum)`	Available in PDAP.
`void setMinimum(int newMinimum)`	Available in PDAP.
`void setOrientation(int orientation)`	Available in PDAP.
`void setPageIncrement(int v)`	Available in PDAP.
`void setUnitIncrement(int v)`	Available in PDAP.
`void setValue(int newValue)`	Available in PDAP.

`void setValues(int value, int visible, int minimum, int maximum)`	Available in PDAP.
`void setVisibleAmount(int newAmount)`	Available in PDAP.

ScrollPane

Table B.41. Methods of the Class ScrollPane

Method	*Alternative/Workaround*
`ScrollPane()`	Available in PDAP.
`ScrollPane(int scrollbarDisplayPolicy)`	Available in PDAP.
`protected void addImpl (Component comp, Object constraints, int index)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`void doLayout()`	Available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`Adjustable getHAdjustable()`	Available in PDAP.
`int getHScrollbarHeight()`	Available in PDAP.
`int getScrollbarDisplayPolicy()`	Available in PDAP.
`Point getScrollPosition()`	Available in PDAP.
`Adjustable getVAdjustable()`	Available in PDAP.
`Dimension getViewportSize()`	Available in PDAP.
`int getVScrollbarWidth()`	Available in PDAP.
`void layout()`	Available in PDAP.
`String paramString()`	Available in PDAP.
`void printComponents(Graphics g)`	Available in PDAP.
`void setLayout(LayoutManager mgr)`	Available in PDAP.
`void setScrollPosition(int x, int y)`	Available in PDAP.
`void setScrollPosition(Point p)`	Available in PDAP.

SystemColor

Table B.42. Methods of the Class SystemColor

Method	*Alternative/Workaround*
`PaintContext createContext(ColorModel cm, Rectangle r, Rectangle2D r2d, AffineTransform xform, RenderingHints hints)`	Not available in PDAP.
`int getRGB()`	Available in PDAP.
`String toString()`	Available in PDAP.

TextArea

Table B.43. Methods of the Class TextArea

Method	*Alternative/Workaround*
`TextArea()`	Available in PDAP.
`TextArea(int rows, int columns)`	Available in PDAP.
`TextArea(String text)`	Available in PDAP.
`TextArea(String text, int rows, int columns)`	Available in PDAP.
`TextArea(String text, int rows, int columns, int scrollbars)`	Available in PDAP.

`void addNotify()`	Not available in PDAP.
`void append(String str)`	Available in PDAP.
`void appendText(String str)`	Available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`int getColumns()`	Available in PDAP.
`Dimension getMinimumSize()`	Available in PDAP.
`Dimension getMinimumSize(int rows, int columns)`	Available in PDAP.
`Dimension getPreferredSize()`	Available in PDAP.
`Dimension getPreferredSize(int rows, int columns)`	Available in PDAP.
`int getRows()`	Available in PDAP.
`int getScrollbarVisibility()`	Available in PDAP.
`void insert(String str, int pos)`	Available in PDAP.
`void insertText(String str, int pos)`	Available in PDAP.
`Dimension minimumSize()`	Available in PDAP.
`Dimension minimumSize(int rows, int columns)`	Available in PDAP.
`protected String paramString()`	Available in PDAP.
`Dimension preferredSize()`	Available in PDAP.
`Dimension preferredSize(int rows, int columns)`	Available in PDAP.
`void replaceRange(String str, int start, int end)`	Available in PDAP.
`void replaceText(String str, int start, int end)`	Available in PDAP.
`void setColumns(int columns)`	Available in PDAP.
`void setRows(int rows)`	Available in PDAP.

`TextComponent`

Table B.44. Methods of the Class `TextComponent`

Method	***Alternative/Workaround***
`void addNotify()`	Not available in PDAP.
`void addTextListener(TextListener l)`	Available in PDAP.
`void enableInputMethods(boolean enable)`	Not available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`Color getBackground()`	Not available in PDAP.
`int getCaretPosition()`	Available in PDAP.
`EventListener[] getListeners(Class listenerType)`	Not available in PDAP.
`String getSelectedText()`	Available in PDAP.
`int getSelectionEnd()`	Available in PDAP.
`int getSelectionStart()`	Available in PDAP.
`String getText()`	Available in PDAP.
`boolean isEditable()`	Available in PDAP.
`protected String paramString()`	Available in PDAP.
`protected void processEvent(AWTEvent e)`	Available in PDAP.
`protected void processTextEvent(TextEvent e)`	Available in PDAP.
`void removeNotify()`	Not available in PDAP.
`void removeTextListener(TextListener l)`	Available in PDAP.
`void select(int selectionStart, int selectionEnd)`	Available in PDAP.
`void selectAll()`	Available in PDAP.
`void setBackground(Color c)`	Not available in PDAP.
`void setCaretPosition(int position)`	Available in PDAP.

`void setEditable(boolean b)`	Available in PDAP.
`void setSelectionEnd(int selectionEnd)`	Available in PDAP.
`void setSelectionStart(int selectionStart)`	Available in PDAP.
`void setText(String t)`	Available in PDAP.

`TextField`

Table B.45. Methods of the Class `TextField`

Method	***Alternative/Workaround***
`TextField()`	Available in PDAP.
`TextField(int columns)`	Available in PDAP.
`TextField(String text)`	Available in PDAP.
`TextField(String text, int columns)`	Available in PDAP.
`void addActionListener(ActionListener l)`	Available in PDAP.
`void addNotify()`	Available in PDAP.
`boolean echoCharIsSet()`	Available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`int getColumns()`	Available in PDAP.
`char getEchoChar()`	Available in PDAP.
`EventListener[] getListeners(Class listenerType)`	Not available in PDAP.
`Dimension getMinimumSize()`	Available in PDAP.
`Dimension getMinimumSize(int columns)`	Available in PDAP.
`Dimension getPreferredSize()`	Available in PDAP.
`Dimension getPreferredSize(int columns)`	Available in PDAP.
`Dimension minimumSize()`	Available in PDAP.
`Dimension minimumSize(int columns)`	Available in PDAP.
`protected String paramString()`	Available in PDAP.
`Dimension preferredSize()`	Available in PDAP.
`Dimension preferredSize(int columns)`	Available in PDAP.
`protected void processActionEvent(ActionEvent e)`	Available in PDAP.
`protected void processEvent(AWTEvent e)`	Available in PDAP.
`void removeActionListener(ActionListener l)`	Available in PDAP.
`void setColumns(int columns)`	Available in PDAP.
`void setEchoChar(char c)`	Available in PDAP.
`void setEchoCharacter(char c)`	Available in PDAP.
`void setText(String t)`	Not available in PDAP.

`Toolkit`

Table B.46. Methods of the Class `Toolkit`

Method	***Alternative/Workaround***
`Toolkit()`	Available in PDAP.
`void addAWTEventListener(AWTEventListener listener, long eventMask)`	Available in PDAP.
`void addPropertyChangeListener(String name, PropertyChangeListener pcl)`	Not available in PDAP.
`abstract void beep()`	Available in PDAP.
`abstract int checkImage(Image image, int width,`	Available in PDAP.

int height, ImageObserver observer)	
protected abstract java.awt.peer.ButtonPeer createButton(Button target)	Not available in PDAP.
protected abstract java.awt.peer.CanvasPeer createCanvas(Canvas target)	Not available in PDAP.
protected abstract java.awt.peer.CheckboxPeer createCheckbox(Checkbox target)	Not available in PDAP.
protected abstract java.awt.peer. CheckboxMenuItemPeer createCheckboxMenuItem (CheckboxMenuItem target)	Not available in PDAP.
protected abstract java.awt.peer.ChoicePeer createChoice(Choice target)	Not available in PDAP.
protected java.awt.peer.LightweightPeer createComponent(Component target)	Not available in PDAP.
Cursor createCustomCursor (Image cursor, Point hotSpot, String name)	Not available in PDAP.
DragGestureRecognizer createDragGestureRecognizer (Class abstractRecognizerClass, DragSource ds, Component c, int srcActions, DragGestureListener dgl)	Not available in PDAP.
abstract java.awt.dnd.peer.DragSourceContextPeer createDragSourceContextPeer(DragGestureEvent dge)	Not available in PDAP.
protected abstract java.awt.peer.FileDialogPeer createFileDialog(FileDialog target)	Not available in PDAP.
protected abstract java.awt.peer.FramePeer createFrame(Frame target)	Not available in PDAP.
Image createImage(byte[] imagedata)	Available in PDAP.
abstract Image createImage(byte[] imagedata, int imageoffset, int imagelength)	Available in PDAP.
abstract Image createImage(ImageProducer producer)	Available in PDAP.
abstract Image createImage(String filename)	Available in PDAP.
abstract Image createImage(URL url)	Available in PDAP.
protected abstract java.awt.peer.LabelPeer createLabel(Label target)	Not available in PDAP.
protected abstract java.awt.peer.ListPeer createList(List target)	Not available in PDAP.
protected abstract java.awt.peer.MenuPeer createMenu(Menu target)	Not available in PDAP.
protected abstract java.awt.peer.MenuBarPeer createMenuBar(MenuBar target)	Not available in PDAP.
protected abstract java.awt.peer.MenuItemPeer createMenuItem(MenuItem target)	Not available in PDAP.
protected abstract java.awt.peer.PanelPeer createPanel(Panel target)	Not available in PDAP.
protected abstract java.awt.peer.PopupMenuPeer createPopupMenu(PopupMenu target)	Not available in PDAP.
protected abstract java.awt.peer.ScrollbarPeer createScrollbar(Scrollbar target)	Not available in PDAP.
protected abstract java.awt.peer.ScrollPanePeer createScrollPane(ScrollPane target)	Not available in PDAP.
protected abstract java.awt.peer.TextAreaPeer createTextArea(TextArea target)	Not available in PDAP.

protected abstract java.awt.peer.TextFieldPeer createTextField(TextField target)	Not available in PDAP.
protected abstract java.awt.peer.WindowPeer createWindow(Window target)	Not available in PDAP.
Dimension getBestCursorSize(int preferredWidth, int preferredHeight)	Not available in PDAP.
abstract ColorModel getColorModel()	Available in PDAP.
static Toolkit getDefaultToolkit()	Available in PDAP.
Object getDesktopProperty(String propertyName)	Not available in PDAP.
abstract String[] getFontList()	Available in PDAP.
abstract FontMetrics getFontMetrics(Font font)	Available in PDAP.
protected abstract java.awt.peer.FontPeer getFontPeer(String name, int style)	Not available in PDAP.
abstract Image getImage(String filename)	Available in PDAP.
abstract Image getImage(URL url)	Not available in PDAP.
boolean getLockingKeyState(int keyCode)	Not available in PDAP.
int getMaximumCursorColors()	Not available in PDAP.
int getMenuShortcutKeyMask()	Available in PDAP.
protected static Container getNativeContainer (Component c)	Not available in PDAP.
PrintJob getPrintJob(Frame frame, String jobtitle, JobAttributes jobAttributes, PageAttributes pageAttributes)	Not available in PDAP.
abstract PrintJob getPrintJob(Frame frame, String jobtitle, Properties props)	Not available in PDAP.
static String getProperty (String key, String defaultValue)	Available in PDAP.
abstract int getScreenResolution()	Available in PDAP.
abstract Dimension getScreenSize()	Available in PDAP.
abstract Clipboard getSystemClipboard()	Not available in PDAP.
EventQueue getSystemEventQueue()	Available in PDAP.
protected abstract EventQueue getSystemEventQueueImpl()	Available in PDAP.
protected void initializeDesktopProperties()	Not available in PDAP.
protected Object lazilyLoadDesktopProperty (String name)	Not available in PDAP.
protected void loadSystemColors(int[] systemColors)	Available in PDAP.
abstract Map mapInputMethodHighlight (InputMethodHighlight highlight)	Not available in PDAP.
abstract boolean prepareImage(Image image, int width, int height, ImageObserver observer)	Available in PDAP.
void removeAWTEventListener(AWTEventListener listener)	Available in PDAP.
void removePropertyChangeListener(String name, PropertyChangeListener pcl)	Not available in PDAP.
protected void setDesktopProperty(String name, Object newValue)	Not available in PDAP.
void setLockingKeyState(int keyCode, boolean on)	Not available in PDAP.
abstract void sync()	Available in PDAP.

Window

Table B.47. Methods of the Class `Window`	
Method	***Alternative/Workaround***
`Window()`	Not available in PDAP.
`Window(Frame owner)`	Available in PDAP.
`Window(Window owner)`	Available in PDAP.
`Window(Window owner, GraphicsConfiguration gc)`	Available in PDAP.
`void addNotify()`	Not available in PDAP.
`void addWindowListener(WindowListener l)`	Available in PDAP.
`void applyResourceBundle(ResourceBundle rb)`	Not available in PDAP.
`void applyResourceBundle(String rbName)`	Not available in PDAP.
`void dispose()`	Available in PDAP.
`protected void finalize()`	Not available in PDAP.
`AccessibleContext getAccessibleContext()`	Not available in PDAP.
`Component getFocusOwner()`	Available in PDAP.
`GraphicsConfiguration getGraphicsConfiguration()`	Not available in PDAP.
`InputContext getInputContext()`	Not available in PDAP.
`EventListener[] getListeners(Class listenerType)`	Not available in PDAP.
`Locale getLocale()`	Available in PDAP.
`Window[] getOwnedWindows()`	Not available in PDAP.
`Window getOwner()`	Not available in PDAP.
`Toolkit getToolkit()`	Available in PDAP.
`String getWarningString()`	Available in PDAP.
`void hide()`	Not available in PDAP.
`boolean isShowing()`	Available in PDAP.
`void pack()`	Available in PDAP.
`boolean postEvent(Event e)`	Available in PDAP.
`protected void processEvent(AWTEvent e)`	Available in PDAP.
`protected void processWindowEvent(WindowEvent e)`	Available in PDAP.
`void removeWindowListener(WindowListener l)`	Available in PDAP.
`void setCursor(Cursor cursor)`	Available in PDAP.
`void show()`	Available in PDAP.
`void toBack()`	Available in PDAP.
`void toFront()`	Available in PDAP.

java.awt.event

Table B.48. Interfaces of the java.awt.event Package	
J2SE Interface	***Availability in PDAP***
`ActionListener`	Available in PDAP.
`AdjustmentListener`	Available in PDAP.
`AWTEventListener`	Available in PDAP.
`ComponentListener`	Available in PDAP.
`ContainerListener`	Available in PDAP.
`FocusListener`	Available in PDAP.
`HierarchyBoundsListener`	Not available in PDAP.
`HierarchyListener`	Not available in PDAP.
`InputMethodListener`	Not available in PDAP.
`ItemListener`	Available in PDAP.
`KeyListener`	Available in PDAP.
`MouseListener`	Available in PDAP.
`MouseMotionListener`	Available in PDAP.
`TextListener`	Available in PDAP.
`WindowListener`	Available in PDAP.

Table B.49. Classes of the `java.awt.event` Package	
J2SE Interface	***Availability in PDAP***
`ActionEvent`	All J2SE methods are available in PDAP.
`AdjustmentEvent`	All J2SE methods are available in PDAP.
`ComponentAdapter`	All J2SE methods are available in PDAP.
`ComponentEvent`	All J2SE methods are available in PDAP.
`ContainerAdapter`	All J2SE methods are available in PDAP.
`ContainerEvent`	All J2SE methods are available in PDAP.
`FocusAdapter`	All J2SE methods are available in PDAP.
`FocusEvent`	All J2SE methods are available in PDAP.
`HierarchyBoundsAdapter`	Not available in PDAP.
`HierarchyEvent`	Not available in PDAP.
`InputEvent`	All J2SE methods are available in PDAP.
`InputMethodEvent`	Not available in PDAP.
`InvocationEvent`	All J2SE methods are available in PDAP.
`ItemEvent`	All J2SE methods are available in PDAP.
`KeyAdapter`	All J2SE methods are available in PDAP.
`KeyEvent`	All J2SE methods are available in PDAP.
`MouseAdapter`	All J2SE methods are available in PDAP.
`MouseEvent`	All J2SE methods are available in PDAP.
`MouseMotionAdapter`	All J2SE methods are available in PDAP.
`PaintEvent`	All J2SE methods are available in PDAP.
`TextEvent`	All J2SE methods are available in PDAP.
`WindowAdapter`	All J2SE methods are available in PDAP.
`WindowEvent`	All J2SE methods are available in PDAP.

java.awt.image

Table B.50. Interfaces of the `java.awt.image` Package	
J2SE Interface	***Availability in PDAP***
`BufferedImageOp`	Not available in PDAP.
`ImageConsumer`	All J2SE methods are available.
`ImageObserver`	All J2SE methods are available.
`ImageProducer`	All J2SE methods are available.
`RasterOp`	Not available in PDAP.
`RenderedImage`	Not available in PDAP.
`TileObserver`	Not available in PDAP.
`WritableRenderedImage`	Not available in PDAP.

Table B.51. Classes of the `java.awt.image` Package	
J2SE Class	***Availability in PDAP***
`AffineTransformOp`	Not available in PDAP.
`AreaAveragingScaleFilter`	All J2SE methods are available.
`BandCombineOp`	Not available in PDAP.
`BandedSampleModel`	Not available in PDAP.
`BufferedImage`	Not available in PDAP.
`BufferedImageFilter`	Not available in PDAP.
`ByteLookupTable`	Not available in PDAP.
`ColorConvertOp`	Not available in PDAP.
`ColorModel`	Partially contained; see Table B.53 for details.
`ComponentColorModel`	Not available in PDAP.
`ComponentSampleModel`	Not available in PDAP.
`ConvolveOp`	Not available in PDAP.
`CropImageFilter`	All J2SE methods are available.
`DataBuffer`	Not available in PDAP.
`DataBufferByte`	Not available in PDAP.
`DataBufferInt`	Not available in PDAP.
`DataBufferShort`	Not available in PDAP.
`DataBufferUShort`	Not available in PDAP.
`DirectColorModel`	Partially contained; see Table B.54 for details.
`FilteredImageSource`	All J2SE methods are available.
`ImageFilter`	Partially contained; see Table B.55 for details.
`IndexColorModel`	Partially contained; see Table B.56 for details.
`Kernel`	Not available in PDAP.
`LookupOp`	Not available in PDAP.
`LookupTable`	Not available in PDAP.
`MemoryImageSource`	All J2SE methods are available.
`MultiPixelPackedSampleModel`	Not available in PDAP.
`PackedColorModel`	Not available in PDAP.
`PixelGrabber`	All J2SE methods are available.
`PixelInterleavedSampleModel`	Not available in PDAP.
`Raster`	Not available in PDAP.
`ReplicateScaleFilter`	All J2SE methods are available.

RescaleOp	Not available in PDAP.
RGBImageFilter	All J2SE methods are available.
SampleModel	Not available in PDAP.
ShortLookupTable	Not available in PDAP.
SinglePixelPackedSampleModel	Not available in PDAP.
WritableRaster	Not available in PDAP.

Table B.52. Exceptions of the `java.awt.image` Package

J2SE Exception	***Availability in PDAP***
ImagingOpException	Not available in PDAP.
RasterFormatException	Not available in PDAP.

`ColorModel`

Table B.53. Methods of the Class `ColorModel`

Method	***Alternative/Workaround***
ColorModel(int bits)	Available in PDAP.
protected ColorModel(int pixel_bits, int[] bits, ColorSpace cspace, boolean hasAlpha, boolean isAlphaPremultiplied, int transparency, int transferType)	Not available in PDAP.
ColorModel coerceData(WritableRaster raster, boolean isAlphaPremultiplied)	Not available in PDAP.
SampleModel createCompatibleSampleModel (int w, int h)	Not available in PDAP.
WritableRaster createCompatibleWritableRaster (int w, int h)	Not available in PDAP.
boolean equals(Object obj)	Available in PDAP.
void finalize()	Available in PDAP.
abstract int getAlpha(int pixel)	Available in PDAP.
int getAlpha(Object inData)	Not available in PDAP.
WritableRaster getAlphaRaster(WritableRaster raster)	Not available in PDAP.
abstract int getBlue(int pixel)	Available in PDAP.
int getBlue(Object inData)	Not available in PDAP.
ColorSpace getColorSpace()	Not available in PDAP.
int[] getComponents(int pixel, int[] components, int offset)	Not available in PDAP.
int[] getComponents(Object pixel, int[] components, int offset)	Not available in PDAP.
int[] getComponentSize()	Not available in PDAP.
int getComponentSize(int componentIdx)	Not available in PDAP.
int getDataElement(int[] components, int offset)	Not available in PDAP.
Object getDataElements(int[] components, int offset, Object obj)	Not available in PDAP.
Object getDataElements(int rgb, Object pixel)	Not available in PDAP.
abstract int getGreen(int pixel)	Available in PDAP.
int getGreen(Object inData)	Not available in PDAP.
float[] getNormalizedComponents(int[] components, int offset, float[] normComponents, int normOffset)	Not available in PDAP.

int getNumColorComponents()	Not available in PDAP.
int getNumComponents()	Not available in PDAP.
int getPixelSize()	Available in PDAP.
abstract int getRed(int pixel)	Available in PDAP.
int getRed(Object inData)	Not available in PDAP.
int getRGB(int pixel)	Not available in PDAP.
int getRGB(Object inData)	Not available in PDAP.
static ColorModel getRGBdefault()	Available in PDAP.
int getTransferType()	Not available in PDAP.
int getTransparency()	Not available in PDAP.
int[] getUnnormalizedComponents(float[] normComponents, int normOffset, int[] components, int offset)	Not available in PDAP.
boolean hasAlpha()	Available in PDAP.
int hashCode()	Available in PDAP.
boolean isAlphaPremultiplied()	Available in PDAP.
boolean isCompatibleRaster(Raster raster)	Not available in PDAP.
boolean isCompatibleSampleModel(SampleModel sm)	Not available in PDAP.
String toString()	Available in PDAP.

DirectColorModel

Table B.54. Methods of the Class `DirectColorModel`

Method	***Alternative/Workaround***
DirectColorModel(ColorSpace space, int bits, int rmask, int gmask, int bmask, int amask, boolean isAlphaPremultiplied, int transferType)	Not available in PDAP.
DirectColorModel(int bits, int rmask, int gmask, int bmask)	Available in PDAP.
DirectColorModel(int bits, int rmask, int gmask, int bmask, int amask)	Available in PDAP.
ColorModel coerceData(WritableRaster raster, boolean isAlphaPremultiplied)	Not available in PDAP.
WritableRaster createCompatibleWritableRaster (int w, int h)	Not available in PDAP.
int getAlpha(int pixel)	Available in PDAP.
int getAlpha(Object inData)	Not available in PDAP.
int getAlphaMask()	Available in PDAP.
int getBlue(int pixel)	Available in PDAP.
int getBlue(Object inData)	Not available in PDAP.
int getBlueMask()	Available in PDAP.
int[] getComponents(int pixel, int[] components, int offset)	Not available in PDAP.
int[] getComponents(Object pixel, int[] components, int offset)	Not available in PDAP.
int getDataElement(int[] components, int offset)	Not available in PDAP.
Object getDataElements(int[] components, int offset, Object obj)	Not available in PDAP.
Object getDataElements(int rgb, Object pixel)	Not available in PDAP.
int getGreen(int pixel)	Available in PDAP.

`int getGreen(Object inData)`	Not available in PDAP.
`int getGreenMask()`	Available in PDAP.
`int getRed(int pixel)`	Available in PDAP.
`int getRed(Object inData)`	Not available in PDAP.
`int getRedMask()`	Available in PDAP.
`int getRGB(int pixel)`	Available in PDAP.
`int getRGB(Object inData)`	Not available in PDAP.
`boolean isCompatibleRaster(Raster raster)`	Not available in PDAP.
`String toString()`	Available in PDAP.

ImageFilter

Table B.55. Methods of the Class `ImageFilter`

Method	*Alternative/Workaround*
`ImageFilter()`	Available in PDAP.
`Object clone()`	Not available in PDAP.
`ImageFilter getFilterInstance(ImageConsumer ic)`	Available in PDAP.
`void imageComplete(int status)`	Available in PDAP.
`Void resendTopDownLeftRight(ImageProducer ip)`	Available in PDAP.
`void setColorModel(ColorModel model)`	Available in PDAP.
`void setDimensions(int width, int height)`	Available in PDAP.
`void setHints(int hints)`	Available in PDAP.
`void setPixels(int x, int y, int w, int h, ColorModel model, byte[] pixels, int off, int scansize)`	Available in PDAP.
`void setPixels(int x, int y, int w, int h, ColorModel model, int[] pixels, int off, int scansize)`	Available in PDAP.
`void setProperties(Hashtable props)`	Available in PDAP.

IndexColorModel

Table B.56. Methods of the Class `IndexColorModel`

Method	*Alternative/Workaround*
`IndexColorModel(int bits, int size, byte[] r, byte[] g, byte[] b)`	Available in PDAP.
`IndexColorModel(int bits, int size, byte[] r, byte[] g, byte[] b, byte[] a)`	Available in PDAP.
`IndexColorModel(int bits, int size, byte[] r, byte[] g, byte[] b, int trans)`	Available in PDAP.
`IndexColorModel(int bits, int size, byte[] cmap, int start, boolean hasalpha)`	Available in PDAP.
`IndexColorModel(int bits, int size, byte[] cmap, int start, boolean hasalpha, int trans)`	Available in PDAP.
`IndexColorModel(int bits, int size, int[] cmap, int start, boolean hasalpha, int trans, int transferType)`	Not available in PDAP.
`IndexColorModel(int bits, int size, int[] cmap, int start, int transferType, BigInteger validBits)`	Not available in PDAP.
`BufferedImage convertToIntDiscrete (Raster`	Not available in PDAP.

raster, boolean forceARGB)	
SampleModel createCompatibleSampleModel (int w, int h)	Not available in PDAP.
WritableRaster createCompatibleWritableRaster (int w, int h)	Not available in PDAP.
void finalize()	Available in PDAP.
int getAlpha(int pixel)	Available in PDAP.
void getAlphas(byte[] a)	Available in PDAP.
int getBlue(int pixel)	Available in PDAP.
void getBlues(byte[] b)	Available in PDAP.
int[] getComponents (int pixel, int[] components, int offset)	Not available in PDAP.
int[] getComponents (Object pixel, int[] components, int offset)	Not available in PDAP.
int[] getComponentSize()	Not available in PDAP.
int getDataElement(int[] components, int offset)	Not available in PDAP.
Object getDataElements (int[] components, int offset, Object pixel)	Not available in PDAP.
Object getDataElements(int rgb, Object pixel)	Not available in PDAP.
int getGreen(int pixel)	Available in PDAP.
void getGreens(byte[] g)	Available in PDAP.
int getMapSize()	Available in PDAP.
int getRed(int pixel)	Available in PDAP.
void getReds(byte[] r)	Available in PDAP.
int getRGB(int pixel)	Available in PDAP.
void getRGBs(int[] rgb)	Available in PDAP.
int getTransparency()	Not available in PDAP.
int getTransparentPixel()	Available in PDAP.
BigInteger getValidPixels()	Not available in PDAP.
boolean isCompatibleRaster(Raster raster)	Not available in PDAP.
boolean isCompatibleSampleModel(SampleModel sm)	Not available in PDAP.
boolean isValid()	Not available in PDAP.
boolean isValid(int pixel)	Not available in PDAP.
String toString()	Available in PDAP.

java.io

Table B.57. Interfaces of the `java.io` Package

J2SE Interface	***Availability in CLDC***
`DataInput`	Partially contained; see Table B.60 for details.
`DataOutput`	Partially contained; see Table B.61 for details.
`Externalizable`	Not available in CLDC.
`FileFilter`	Not available in CLDC.
`FilenameFilter`	Not available in CLDC.
`ObjectInput`	Not available in CLDC.
`ObjectInputValidation`	Not available in CLDC.
`ObjectOutput`	Not available in CLDC.
`ObjectStreamConstants`	Not available in CLDC.
`Serializable`	Not available in CLDC.

Table B.58. Classes of the `java.io` Package

J2SE Class	***Availability in CLDC***
`BufferedInputStream`, `BufferedOutputStream`	Not available in CLDC.

`BufferedReader, BufferedWriter`	Not available in CLDC.
	
	Workaround for `readLine()` using PC and UNIX encoding:
	
	`static String readLine (Reader reader) throws IOException {`
	`  StringBuffer buf = new StringBuffer();`
	`  while (true) {`
	`    int c = reader.read();`
	`    if (c == -1) {`
	`      if (buf.length() == 0) return null;`
	`      break;`
	`    }`
	`    if (c == '\n') break;`
	`    if (c != '\r') buf.append ((char) c);`
	`  }`
	`  return buf.toString();`
	`}`

`ByteArrayInputStream`	All J2SE methods are available in CLDC.
`ByteArrayOutputStream`	Partially contained; see Table B.62 for details.
`CharArrayReader`, `CharArrayWriter`	Not available in CLDC.
`DataInputStream`, `DataOutputStream`	Partially contained; see Table B.63 and B.64 for details.
`File`, `FileDescriptor`, `FileInputStream`, `FilePermission`, `FileReader`, `FileWriter`	Files not available in CLDC. Use the classes of the `javax.microedition.rms` as an alternative. For accessing files on memory cards, some devices may provide a file:// protocol implementation in the generic connection framework (see Chapter 6, "Networking: The Generic Connection Framework").

FilterInputStream, FilterOutputStream	Not available in CLDC.
FilterReader, FilterWriter	Not available in CLDC.
InputStream	All J2SE methods are available in CLDC.
InputStreamReader	Partially contained; see Table B.65 for details.
LineNumberInputStream	Not available in CLDC.
LineNumberReader	Not available in CLDC.
ObjectInputStream	Not available in CLDC.
ObjectInputStream.GetField	Not available in CLDC.
ObjectOutputStream	Not available in CLDC.
ObjectOutputStream.PutField	Not available in CLDC.
ObjectStreamClass	Not available in CLDC.
ObjectStreamField	Not available in CLDC.
OutputStream	All J2SE methods are available in CLDC.
OutputStreamWriter	Partially contained; see Table B.66 details.
PipedInputStream, PipedOutputStream	Not available in CLDC.
PipedReader	Not available in CLDC.
PipedWriter	Not available in CLDC.
PrintStream	Partially contained; see Table B.67 for details.
PrintWriter	Not available in CLDC.
PushbackInputStream	Not available in CLDC.
PushbackReader	Not available in CLDC. kXML contains a LookAheadReader that is comparable to some extent.
RandomAccessFile	Files are not available in CLDC. Use the classes of the javax.microedition.rms as an alternative. For accessing files on memory cards, some devices may provide a file:// protocol implementation in the generic connection framework (see Chapter 6).
Reader	Fully available in CLDC.
SequenceInputStream	Not available in CLDC.
SerializablePermission	Not available in CLDC.
StreamTokenizer	Not available in CLDC.
StringBufferInputStream	Not available in CLDC. See StringReader for a workaround.

StringReader	Not available in CLDC.
	
	Use
	
	`new InputStreamReader (new ByteArrayInputStream (s.getBytes()));`
	
	instead of
	
	`new StringReader (s);`

StringWriter	Not available in CLDC. Use

	`ByteArrayOutputStream bos = new ByteArrayOutputStream();`
	`OutputStreamWriter sw = new OutputStreamWriter (bos);`
	`// ... write to sw`
	`String s = new String (bos.getByteArray ());`
	
	instead of
	
	`StringWriter sw = new StringWriter();`
	`// ... write to sw`
	`String s = sw.toString();`

`Writer`	Fully available in CLDC.

Table B.59. Exceptions of the `java.io` Package

J2SE Exception	***Availability in CLDC***
`CharConversionException`	Not available in CLDC.
`EOFException`	Available in CLDC.
`FileNotFoundException`	Not available in CLDC.
`InterruptedIOException`	Available in CLDC.
`InvalidClassException`	Not available in CLDC.
`InvalidObjectException`	Not available in CLDC.
`IOException`	Available in CLDC.
`NotActiveException`	Not available in CLDC.
`NotSerializableException`	Not available in CLDC.
`ObjectStreamException`	Not available in CLDC.
`OptionalDataException`	Not available in CLDC.
`StreamCorruptedException`	Not available in CLDC.
`SyncFailedException`	Not available in CLDC.
`UnsupportedEncodingException`	Available in CLDC.
`UTFDataFormatException`	Available in CLDC.
`WriteAbortedException`	Not available in CLDC.

DataInput

Table B.60. Methods of the Class `DataInput`

Method	***Alternative/Workaround***
`DataInputStream(InputStream in)`	Available in CLDC.
`boolean readBoolean()`	Available in CLDC.
`byte readByte()`	Available in CLDC.
`char readChar()`	Available in CLDC.
`double readDouble()`	Available in CLDC-NG.
`float readFloat()`	Available in CLDC-NG.
`void readFully(byte[] b)`	Available in CLDC.
`void readFully(byte[] b, int off, int len)`	Available in CLDC.
`int readInt()`	Available in CLDC.
`String readLine()`	Not available in CLDC.
`long readLong()`	Available in CLDC.
`short readShort()`	Available in CLDC.

`int readUnsignedByte()`	Available in CLDC.
`int readUnsignedShort()`	Available in CLDC.
`String readUTF()`	Available in CLDC.
`int skipBytes(int n)`	Available in CLDC.

`DataOutput`

Table B.61. Methods of the Class `DataOutput`

Method	*Alternative/Workaround*
`void write(byte[] b)`	Available in CLDC.
`void write(byte[] b, int off, int len)`	Available in CLDC.
`void write(int b)`	Available in CLDC.
`void writeBoolean(boolean v)`	Available in CLDC.
`void writeByte(int v)`	Available in CLDC.
`void writeBytes(String s)`	Not available in CLDC.
`void writeChar(int v)`	Available in CLDC.
`void writeChars(String s)`	Available in CLDC.
`void writeDouble(double v)`	Available in CLDC-NG.
`void writeFloat(float v)`	Available in CLDC-NG.
`void writeInt(int v)`	Available in CLDC.
`void writeLong(long v)`	Available in CLDC.
`void writeShort(int v)`	Available in CLDC.
`void writeUTF(String str)`	classesAvailable in CLDC.

Table B.62. Methods of the Class `ByteArrayOutputStream`

Method	*Alternative/Workaround*
`ByteArrayOutputStream()`	Available in CLDC.
`ByteArrayOutputStream(int size)`	Available in CLDC.
`void close()`	Available in CLDC.
`void reset()`	Available in CLDC.
`void flush()`	Available in CLDC.
`int size()`	Available in CLDC.
`byte [] toByteArray()`	Available in CLDC.
`String toString()`	Available in CLDC.
`String toString(int hibyte)`	Not available in CLDC (deprecated J2SE method).
`String toString(String enc)`	Not available in CLDC.
`void write(byte[] b)`	Available in CLDC.
`void write(byte[] b, int off, int len)`	Available in CLDC.
`void write(int b)`	Available in CLDC.

`void writeTo(OutputStream out)`	Not available in CLDC.
	
	Workaround:
	
	`new DataOutputStream (out).write`
	
	`(byteOutputStream.toByteArray());`

`DataInputStream`

Table B.63. Methods of the Class `DataInputStream`	
Method	***Alternative/Workaround***
`DataInputStream(InputStream in)`	Available in CLDC.
`int available()`	Available in CLDC.
`void close()`	Available in CLDC.
`void mark(int readlimit)`	Available in CLDC.
`Boolean markSupported()`	Available in CLDC.
`int read()`	Available in CLDC.
`int read(byte[] b)`	Available in CLDC.
`int read(byte[] b, int off, int len)`	Available in CLDC.
`boolean readBoolean()`	Available in CLDC.
`byte readByte()`	Available in CLDC.
`char readChar()`	Available in CLDC.
`double readDouble()`	Available in CLDC-NG.
`float readFloat()`	Available in CLDC-NG.
`void readFully(byte[] b)`	Available in CLDC.
`void readFully(byte[] b, int off, int len)`	Available in CLDC.
`int readInt()`	Available in CLDC.
`String readLine()`	Not available in CLDC (deprecated J2SE method).
`long readLong()`	Available in CLDC.
`short readShort()`	Available in CLDC.
`int readUnsignedByte()`	Available in CLDC.
`int readUnsignedShort()`	Available in CLDC.
`String readUTF()`	Available in CLDC.
`static String readUTF(DataInputin)`	Not available in CLDC. Workaround: `in.readUTF()`
`void reset()`	Available in CLDC.
`long skip(long n)`	Available in CLDC.
`int skipBytes(int n)`	Available in CLDC.

`DataOutputStream`

Table B.64. Methods of the Class `DataOutputStream`	
Method	***Alternative/Workaround***
`DataOutputStream(OutputStream out)`	Available in CLDC.
`void close()`	Available in CLDC.
`void flush()`	Available in CLDC.
`int size()`	Not available in CLDC.
`void write(byte[] b)`	Available in CLDC.
`void write(byte[] b, int off, int len)`	Available in CLDC.
`void write(int b)`	Available in CLDC.
`void writeBoolean(boolean v)`	Available in CLDC.

`void writeByte(int v)`	Available in CLDC.
`void writeBytes(String s)`	Not available in CLDC. Please note: This method writes 8-bit chars only and differs from `write (s.getBytes());`
`void writeChar(int v)`	Available in CLDC.
`void writeChars(String s)`	Available in CLDC.
`void writeDouble(double v)`	Available in CLDC-NG.
`void writeFloat(float v)`	Available in CLDC-NG.
`void writeInt(int v)`	Available in CLDC.
`void writeLong(long v)`	Available in CLDC.
`void writeShort(int v)`	Available in CLDC.
`void writeUTF(String str)`	Available in CLDC.

`InputStreamReader`

Table B.65. Methods of the Class `InputStreamReader`

Method	***Alternative/Workaround***
`InputStreamReader(InputStream in)`	Available in CLDC.
`InputStreamReader(InputStream in, String enc)`	Available in CLDC.
`void close()`	Available in CLDC.
`String getEncoding()`	Not available in CLDC.
`void mark(int readAheadLimit)`	Available in CLDC.
`boolean markSupported()`	Available in CLDC.
`int read(char[] cbuf)`	Available in CLDC.
`int read()`	Available in CLDC.
`int read(char[] cbuf, int off, int len)`	Available in CLDC.
`boolean ready()`	Available in CLDC.
`void reset()`	Available in CLDC.
`long skip(long n)`	Available in CLDC.

`OutputStreamWriter`

Table B.66. Methods of the Class `OutputStreamWriter`

Method	***Alternative/Workaround***
`OutputStreamWriter(OutputStream out)`	Available in CLDC.
`OutputStreamWriter(OutputStream out, String enc)`	Available in CLDC.
`void close()`	Available in CLDC.
`void flush()`	Available in CLDC.
`String getEncoding()`	Not available in CLDC.
`void write(char[] cbuf)`	Available in CLDC.
`void write(String str)`	Available in CLDC.
`void write(char[] cbuf, int off, int len)`	Available in CLDC.
`void write(int c)`	Available in CLDC.
`void write(String str, int off, int len)`	Available in CLDC.

`PrintStream`

Table B.67. Methods of the Class `PrintStream`

Method	*Alternative/Workaround*
`PrintStream(OutputStream out)`	Available in CLDC.
`PrintStream(OutputStream out, boolean autoFlush)`	Not available in CLDC.
`boolean checkError()`	Available in CLDC.
`void close()`	Available in CLDC.
`void flush()`	Available in CLDC.
`void print(boolean b)`	Available in CLDC.
`void print(char c)`	Available in CLDC.
`void print(char[] s)`	Available in CLDC.
`void print(double d)`	Not available in CLDC.
`void print(float f)`	Not available in CLDC.
`void print(int i)`	Available in CLDC.
`void print(long l)`	Available in CLDC.
`void print(Object obj)`	Available in CLDC.
`void print(String s)`	Available in CLDC.
`void println()`	Available in CLDC.
`void println(boolean x)`	Available in CLDC.
`void println(char x)`	Available in CLDC.
`void println(char[] x)`	Available in CLDC.
`void println(double x)`	Not available in CLDC.
`void println(float x)`	Not available in CLDC.
`void println(int x)`	Available in CLDC.
`void println(long x)`	Available in CLDC.
`void println(Object x)`	Available in CLDC.
`void println(String x)`	Available in CLDC.
`protected void setError()`	Available in CLDC.
`void write(byte[] buf, int off, int len)`	Available in CLDC.
`void write(int b)`	Available in CLDC.

java.lang

Table B.68. Interfaces of the `java.lang` Package

J2SE Interface	Availability in CLDC
`Clonable`	Not available in CLDC.
`Comparable`	Not available in CLDC.
`Runnable`	Fully available in CLDC.

Table B.69. Classes of the `java.lang` Package

J2SE Class	Availability in CLDC
`Boolean`	Partially contained; see Table B.72 for details.
`Byte`	Partially contained; see Table B.73 for details.
`Character`	Partially contained; see Table B.74 for details.
`Character.Subset`	Not available in CLDC.
`Character.UnicodeBlock`	Not available in CLDC.
`Class`	Partially contained; see Table B.75 for details.
`ClassLoader`	Not available in CLDC.
`Compiler`	Not available in CLDC.
`Double`	Partially contained in CLDC-NG; see Table B.76 for details.
`Float`	Partially contained in CLDC-NG; see Table B.77 for details.
`InheritableThreadLocal`	Not available in CLDC.
`Integer`	Partially contained; see Table B.78 for details.
`Long`	Partially contained; see Table B.79 for details.
`Math`	Partially contained; see Table B.80 for details.
`Number`	Not available in CLDC.
`Object`	Partially contained. The CLDC version lacks the `clone()` and `finalize()` methods that are provided in J2SE.
`Package`	Not available in CLDC.
`Process`	Not available in CLDC.
`Runtime`	Partially contained; see the section "Runtime" for details.
`RuntimePermission`	Not available in CLDC.
`SecurityManager`	Not available in CLDC.
`Short`	Partially contained; see Table B.81 for details.
`StrictMath`	Not available in CLDC.
`String`	Partially contained; see Table B.82 for details.
`StringBuffer`	Partially contained; see Table B.83 for details.
`System`	Partially contained; see Table B.84 for details.
`Thread`	Partially contained; see Table B.85 for details.
`ThreadGroup`	Not available in CLDC.
`ThreadLocal`	Not available in CLDC.
`Throwable`	Partially contained; see Table B.86 for details.
`Void`	Not available in CLDC.

Table B.70. Exceptions of the `java.lang` Package

J2SE Exception	Availability in CLDC
`ArithmeticException`	Fully available in CLDC, except the missing methods of the `Throwable` class.
`ArrayIndexOutOfBoundsException`	Fully available in CLDC, except the missing

	methods of the Throwable class.
ArrayStoreException	Fully available in CLDC, except the missing methods of the Throwable class.
ClassCastException	Fully available in CLDC, except the missing methods of the Throwable class.
ClassNotFoundException	Fully available in CLDC, except the missing methods of the Throwable class.
CloneNotSupportedException	Not available in CLDC.
Exception	Fully available in CLDC, except the missing methods of the Throwable class.
IllegalAccessException	Fully available in CLDC, except the missing methods of the Throwable class.
IllegalArgumentException	Fully available in CLDC, except the missing methods of the Throwable class.
IllegalMonitorStateException	Fully available in CLDC, except the missing methods of the Throwable class.
IllegalStateException	Fully available in CLDC, except the missing methods of the Throwable class.
IllegalThreadStateException	Fully available in CLDC, except the missing methods of the Throwable class.
IndexOutOfBoundsException	Fully available in CLDC, except the missing methods of the Throwable class.
InstantiationException	Fully available in CLDC, except the missing methods of the Throwable class.
InterruptedException	Fully available in CLDC, except the missing methods of the Throwable class.
NegativeArraySizeException	Fully available in CLDC, except the missing methods of the Throwable class.
NoSuchFieldException	Fully available in CLDC, except the missing methods of the Throwable class.
NoSuchMethodException	Fully available in CLDC, except the missing methods of the Throwable class.
NullPointerException	Fully available in CLDC, except the missing methods of the Throwable class.
NumberFormatException	Fully available in CLDC, except the missing methods of the Throwable class.
RuntimeException	Fully available in CLDC, except the missing methods of the Throwable class.
SecurityException	Fully available in CLDC, except the missing methods of the Throwable class.
StringIndexOutOfBoundsException	Fully available in CLDC, except the missing methods of the Throwable class.
UnsupportedOperationException	Not available in CLDC.

Table B.71. Errors of the java.lang Package

J2SE Error	*Availability in CLDC*
AbstractMethodError	Not available in CLDC.
ClassCircularityError	Not available in CLDC.
ClassFormatError	Not available in CLDC.
Error	Fully available in CLDC, except the missing methods of the Throwable class.

`ExceptionInInitializerError`	Not available in CLDC.
`IllegalAccessError`	Not available in CLDC.
`IncompatibleClassChangeError`	Not available in CLDC.
`InstantiationError`	Not available in CLDC.
`InternalError`	Not available in CLDC.
`LinkageError`	Not available in CLDC.
`NoClassDefFoundError`	Available in CLDC-NG.
`NoSuchFieldError`	Not available in CLDC.
`NoSuchMethodError`	Not available in CLDC.
`OutOfMemoryError`	Fully available in CLDC, except the missing methods of the `Throwable` class.
`StackOverflowError`	Not available in CLDC.
`ThreadDeath`	Not available in CLDC.
`UnknownError`	Not available in CLDC.
`UnsatisfiedLinkError`	Not available in CLDC.
`UnsupportedClassVersionError`	Not available in CLDC.
`VerifyError`	Not available in CLDC.
`VirtualMachineError`	Fully available in CLDC, except the missing methods of the `Throwable` class.

`Boolean`

Table B.72. Methods of the Class `Boolean`

Method	***Availability in CLDC***
`Boolean(boolean value)`	Available in CLDC.
`Boolean(String s)`	Not available in CLDC. Workaround: `Boolean (s != null && s.toLowerCase().equals ("true"));`
`Boolean booleanValue()`	Available in CLDC.
`Boolean equals(Object obj)`	Available in CLDC.
`static boolean getBoolean(String name)`	Not available in CLDC. Workaround: `(name != null && name.toLowerCase().equals ("true"));`
`int hashCode()`	Available in CLDC.
`String toString()`	Available in CLDC.
`static Boolean valueOf(String s)`	Not available in CLDC. Workaround: `new Boolean (s != null && s.toLowerCase().equals ("true"));`

`Byte`

Table B.73. Methods of the Class `Byte`	
Method	***Availability in CLDC***
`Byte(byte value)`	Available in CLDC.
`Byte(String s)`	Not available in CLDC.
`byte byteValue()`	Available in CLDC.
`int compareTo(Byte anotherByte)`	Not available in CLDC. Workaround: compare the return values of `byteValue()`
`int compareTo(Object o)`	Not available in CLDC.
`static Byte decode(String nm)`	Not available in CLDC. Workaround: Determine radix using `String.startsWith()`, then apply `parseByte()` with the corresponding radix.
`double doubleValue()`	Not available in CLDC.
`boolean equals(Object obj)`	Available in CLDC.
`float floatValue()`	Not available in CLDC.
`int hashCode()`	Available in CLDC.
`int intValue()`	Not available in CLDC. Workaround: `(int) byteValue()`
`long longValue()`	Not available in CLDC. Workaround: `(long) byteValue()`
`static byte parseByte(String s)`	Available in CLDC.
`static byte parseByte(String s, int radix)`	Available in CLDC.
`Short shortValue()`	Not available in CLDC. Workaround: `(short) byteValue()`
`String toString()`	Available in CLDC.

`static String toString(byte b)`	Not available in CLDC.
	
	Workaround:
	
	`""+b`
	
	or
	
	`new Byte (b).toString()`

`static Byte valueOf(String s)`	Not available in CLDC. Workaround:

	`new Byte (Byte.parseByte (s));`
`static Byte valueOf (String s, intradix)`	Not available in CLDC. Workaround: `new Byte (Byte.parseByte (s, radix);`

`Character`

Table B.74. Methods of the Class `Character`

Method	***Availability in CLDC***
`Character(char value)`	Available in CLDC.
`Char charValue()`	Available in CLDC.
`int compareTo(Character anotherCharacter)`	Not available in CLDC. Workaround: Compare the return values of `charValue()`
`int compareTo(Object o)`	Not available in CLDC.
`static int digit(char ch, int radix)`	Available in CLDC.
`boolean equals(Object obj)`	Available in CLDC.

`static char forDigit(int digit, int radix)`	Not available in CLDC.
	
	Workaround:
	
	`(char) (digit > 9`
	`? ((int) 'a') + digit – 10`
	`: ((int) '0') + digit)`

`static int getNumericValue (char ch)`	Not available in CLDC. Incomplete workaround: `((int) ch) – ((int) '0')`
`static int getType(char ch)`	Not available in CLDC. Workaround: For limited cases, `isDigit()` may help.
`int hashCode()`	Available in CLDC.
`static boolean isDefined(char ch)`	Not available in CLDC.
`static boolean isDigit (char ch)`	Available in CLDC.
`static boolean isIdentifierIgnorable(char ch)`	Not available in CLDC.
`static boolean isISOControl (char ch)`	Not available in CLDC.
`static boolean isJavaIdentifierPart(char ch)`	Not available in CLDC.
`static boolean isJavaIdentifierStart(char ch)`	Not available in CLDC.
`static boolean isJavaLetter (char ch)`	Not available in CLDC(deprecated J2SE method).
`static boolean isJavaLetterOrDigit (char ch)`	Not available in CLDC (deprecated J2SE method).
`static boolean isLetter(char ch)`	Not available in CLDC.
`static boolean isLetterOrDigit (char`	Not available in CLDC.

ch)	
static boolean isLowerCase (char ch)	Available in CLDC.
static boolean isSpace(char ch)	Not available in CLDC (deprecated J2SE method).
static boolean isSpaceChar(char ch)	Not available in CLDC.
static boolean isTitleCase(char ch)	Not available in CLDC.
static boolean isUnicodeIdentifierPart(char ch)	Not available in CLDC.
static boolean isUnicodeIdentifierStart(char ch)	Not available in CLDC.
static boolean isUpperCase(char ch)	Available in CLDC.
static boolean isWhitespace (char ch)	Not available in CLDC. Incomplete workaround: `ch <= ' '`
static char toLowerCase(char ch)	Available in CLDC.
String toString()	Available in CLDC.
static char toTitleCase(char ch)	Not available in CLDC.
static char toUpperCase(char ch)	Available in CLDC.

Class

Table B.75. Methods of the Class Class

Method	***Availability in CLDC***
static Class forName(String className)	Available in CLDC.
static Class forName(String name, boolean initialize, ClassLoaderloader)	Not available in CLDC.
Class[] getClasses()	Not available in CLDC.
ClassLoader getClassLoader()	Not available in CLDC.
Class getComponentType()	Not available in CLDC.
Constructor getConstructor(Class[] parameterTypes)	Not available in CLDC.
Constructor[] getConstructors()	Not available in CLDC.
Class[] getDeclaredClasses()	Not available in CLDC.
Constructor getDeclaredConstructor (Class[] parameterTypes)	Not available in CLDC.
Constructor[] getDeclaredConstructors()	Not available in CLDC.
Field getDeclaredField(String name)	Not available in CLDC.
Field[] getDeclaredFields()	Not available in CLDC.
Method getDeclaredMethod(String name, Class[] parameterTypes)	Not available in CLDC.

`Method[] getDeclaredMethods()`	Not available in CLDC.
`Class getDeclaringClass()`	Not available in CLDC.
`Field getField(String name)`	Not available in CLDC.
`Field[] getFields()`	Not available in CLDC.
`Class[] getInterfaces()`	Not available in CLDC.
`Method getMethod(String name, Class[] parameterTypes)`	Not available in CLDC.
`Method[] getMethods()`	Not available in CLDC.
`int getModifiers()`	Not available in CLDC.
`String getName()`	Available in CLDC.
`Package getPackage()`	Not available in CLDC.
`ProtectionDomain getProtectionDomain()`	Not available in CLDC.
`URL getResource(String name)`	Not available in CLDC.
`InputStream getResourceAsStream(String name)`	Available in CLDC.
`Object[] getSigners()`	Not available in CLDC.
`Class getSuperclass()`	Not available in CLDC.
`boolean isArray()`	Available in CLDC.
`boolean isAssignableFrom(Class cls)`	Available in CLDC.
`boolean isInstance(Object obj)`	Available in CLDC.
`boolean isInterface()`	Available in CLDC.
`boolean isPrimitive()`	Not available in CLDC.
`Object newInstance()`	Available in CLDC.
`String toString()`	Available in CLDC.

`Double`

Table B.76. Methods of the Class `Double`

Method	***Availability in CLDC-NG***
`Double(double value)`	Available in CLDC-NG.
`Double(String s)`	Not available in CLDC-NG.
`byte byteValue()`	Available in CLDC-NG.

`int compareTo(Double anotherDouble)`	Not available in CLDC-NG.
`int compareTo(Object o)`	Not available in CLDC-NG.
`static long doubleToLongBits(double value)`	Available in CLDC-NG.
`static long doubleToRawLongBits(double value)`	Not available in CLDC-NG.
`double doubleValue()`	Available in CLDC-NG.
`boolean equals(Object obj)`	Available in CLDC-NG.
`float floatValue()`	Available in CLDC-NG.
`int hashCode()`	Available in CLDC-NG.
`int intValue()`	Available in CLDC-NG.
`boolean isInfinite()`	Available in CLDC-NG.
`static boolean isInfinite(double v)`	Available in CLDC-NG.
`boolean isNaN()`	Available in CLDC-NG.
`static boolean isNaN(double v)`	Available in CLDC-NG.
`static double longBitsToDouble(long bits)`	Available in CLDC-NG.
`long longValue()`	Available in CLDC-NG.
`static double parseDouble(String s)`	Available in CLDC-NG.
`short shortValue()`	Available in CLDC-NG.
`String toString()`	Available in CLDC-NG.
`static String toString(double d)`	Available in CLDC-NG.
static Double valueOf(String s)	Available in CLDC-NG.

Float

Table B.77. Methods Class Float

Method	*Availability in CLDC-NG*
`Float(double value)`	Available in CLDC-NG.
`Float(float value)`	Available in CLDC-NG.
`Float(String s)`	Not available in CLDC-NG.
`byte byteValue()`	Available in CLDC-NG.
`int compareTo(Float anotherFloat)`	Not available in CLDC-NG.
`int compareTo(Object o)`	Not available in CLDC-NG.
`double doubleValue()`	Available in CLDC-NG.
`boolean equals(Object obj)`	Available in CLDC-NG.
`static int floatToIntBits(float value)`	Available in CLDC-NG.
`static int floatToRawIntBits(float value)`	Not available in CLDC-NG.
`float floatValue()`	Available in CLDC-NG.
`int hashCode()`	Available in CLDC-NG.
`static float intBitsToFloat(int bits)`	Available in CLDC-NG.
`int intValue()`	Available in CLDC-NG.
`boolean isInfinite()`	Available in CLDC-NG.
`static boolean isInfinite(float v)`	Available in CLDC-NG.
`boolean isNaN()`	Available in CLDC-NG.
`static boolean isNaN(float v)`	Available in CLDC-NG.
`long longValue()`	Available in CLDC-NG.
`static float parseFloat(String s)`	Available in CLDC-NG.
`short shortValue()`	Available in CLDC-NG.
`String toString()`	Available in CLDC-NG.
`static String toString(float f)`	Available in CLDC-NG.

`static Float valueOf(String s)`	Available in CLDC-NG.

`Integer`

Table B.78. Methods of the Class `Integer`	
Method	***Availability in CLDC***
`Integer(int value)`	Available in CLDC.
`Integer(String s)`	Not available in CLDC. Workaround: `new Integer (Integer.parseInt (s));`
`byte byteValue()`	Available in CLDC.
`int compareTo(Integer anotherInteger)`	Not available in CLDC. Workaround: Compare the values of `intValue()`
`int compareTo(Object o)`	Not available in CLDC.
`static Integer decode(String nm)`	Not available in CLDC. Workaround: Determine sign and radix using `String.startsWith()`, then apply `Integer.parseInt` using the corresponding radix.
`double doubleValue()`	Available in CLDC-NG.
`boolean equals(Object obj)`	Available in CLDC.
`float floatValue()`	Available in CLDC-NG.
`static Integer getInteger (String nm)`	Not available in CLDC. Workaround: `Integer.parseInt (System.getProperty(nm));`
`static Integer getInteger (String nm, int val)`	Not available in CLDC.
`static Integer getInteger (String nm, Integer val)`	Not available in CLDC.
`Int hashCode()`	Available in CLDC.
`int intValue()`	Available in CLDC.
`long longValue()`	Available in CLDC.
`static int parseInt(String s)`	Available in CLDC.
`static int parseInt (String s, int radix)`	Available in CLDC.
`short shortValue()`	Available in CLDC.
`static String toBinaryString(int i)`	Available in CLDC.
`static String toHexString(int i)`	Available in CLDC.
`static String toOctalString(int i)`	Available in CLDC.
`String toString()`	Available in CLDC.
`static String toString(int i)`	Available in CLDC.

`static String toString (int i, int radix)`	Available in CLDC.
`static Integer valueOf(String s)`	Available in CLDC.
`static Integer valueOf (String s, int radix)`	Available in CLDC.

Long

Table B.79. Methods of the Class Long	
Method	***Availability in CLDC***
`Long(long value)`	Available in CLDC.
`Long(String s)`	Not available in CLDC. Workaround: `Long (Long.parseLong (value);`
`byte byteValue()`	Not available in CLDC. Workaround: `(byte) longValue();`
`int compareTo(Long anotherLong)`	Not available in CLDC. Workaround: Compare the values of `longValue()`.
`int compareTo(Object o)`	Not available in CLDC.
`static Long decode(String nm)`	Not available in CLDC. Workaround: Determine sign and radix using `String.startsWith()`, then apply `Long.parseLong` using the corresponding radix.
`double doubleValue()`	Available in CLDC-NG.
`boolean equals(Object obj)`	Available in CLDC.
`float floatValue()`	Available in CLDC-NG.
`static Long getLong(String nm)`	Not available in CLDC. Workaround: `Long.parseLong (System.getProperty (nm));`
`static Long getLong (String nm, long val)`	Not available in CLDC.
`static Long getLong (String nm, Long val)`	Not available in CLDC.
`int hashCode()`	Available in CLDC.
`int intValue()`	Not available in CLDC. Workaround: `(int) longValue()`
`long longValue()`	Available in CLDC.
`Static long parseLong(String s)`	Available in CLDC.

`Static long parseLong(String s, int radix)`	Available in CLDC.
`short shortValue()`	Not available in CLDC. Workaround: `(short) longValue()`
`Static String toBinaryString (long i)`	Not available in CLDC. Workaround: `toString (i, 2);`
`Static String toHexString(long i)`	Not available in CLDC. Workaround: `toString (i, 16);`
`Static String toOctalString(long i)`	Not available in CLDC. Workaround: `toString (i, 8);`
`String toString()`	Available in CLDC.
`Static String toString(long i)`	Available in CLDC.
`Static String toString (long i, int radix)`	Available in CLDC.
`Static Long valueOf(String s)`	Not available in CLDC. Workaround: `new Long(Long.parseLong (s));`
`Static Long valueOf (String s, int radix)`	Not available in CLDC. Workaround: `new Long(Long.parseLong (s, radix));`

Math

Table B.80. Methods of the Class `Math`

Method	*Alternative/Workaround*
`static double abs(double a)`	Available in CLDC-NG.
`static float abs(float a)`	Available in CLDC-NG.
`static int abs(int a)`	Available in CLDC.
`static long abs(long a)`	Available in CLDC.
`static double acos(double a)`	Not available in CLDC.
`static double asin(double a)`	Not available in CLDC.
`static double atan(double a)`	Not available in CLDC.
`static double atan2(double a, double b)`	Not available in CLDC.
`static double ceil(double a)`	Available in CLDC-NG.

`static double cos(double a)`	Not available in CLDC.
`static double exp(double a)`	Not available in CLDC.
`static double floor(double a)`	Available in CLDC-NG.
`static double IEEEremainder (double f1, double f2)`	Not available in CLDC.
`static double log(double a)`	Not available in CLDC.
`static double max(double a, double b)`	Available in CLDC-NG.
`static float max(float a, float b)`	Available in CLDC-NG.
`static int max(int a, int b)`	Available in CLDC.
`static long max(long a, long b)`	Available in CLDC.
`static double min(double a, double b)`	Available in CLDC-NG.
`static float min(float a, float b)`	Available in CLDC-NG.
`static int min(int a, int b)`	Available in CLDC.
`static long min(long a, long b)`	Available in CLDC.
`static double pow(double a, double b)`	Not available in CLDC.
`static double random()`	Not available in CLDC.
`static double rint(double a)`	Not available in CLDC.
`static long round(double a)`	Not available in CLDC.
`static int round(float a)`	Not available in CLDC.
`static double sin(double a)`	Not available in CLDC.
`static double sqrt(double a)`	Not available in CLDC.
`static double tan(double a)`	Not available in CLDC.
`static double toDegrees(double angrad)`	Available in CLDC-NG.
static double toRadians(double angdeg)	Available in CLDC-NG.

`Runtime`

Because executing additional processes or finalization is not supported in CLDC, the only methods of the J2SE `Runtime` class are listed as follows:

- `void exit(int status)`
- `long freeMemory()`
- `void gc()`
- `static Runtime getRuntime()`
- `long totalMemory()`

`Short`

Table B.81. Methods of the Class `Short`

Method	***Availability in CLDC***
`Short(short value)`	Available in CLDC.
`Short(String s)`	Not available in CLDC. Workaround: `Short (parseShort (s))`
`byte byteValue()`	Not available in CLDC. Workaround:

	`(byte) shortValue()`
`int compareTo(Object o)`	Not available in CLDC.
`int compareTo(Short anotherShort)`	Not available in CLDC. Workaround: Compare the return values of `shortValue()`
`Static Short decode(String nm)`	Not available in CLDC. Workaround: Determine radix using `nm.startsWith()`, then apply `parseShort()` with corresponding radix.
`Double doubleValue()`	Not available in CLDC.
`boolean equals(Object obj)`	Available in CLDC.
`float floatValue()`	Not available in CLDC.
`int hashCode()`	Available in CLDC.
`int intValue()`	Not available in CLDC. Workaround: `(int) shortValue()`
`long longValue()`	Not available in CLDC. Workaround: `(int) shortValue()`
`static short parseShort(String s)`	Available in CLDC.
`static short parseShort (Strings, int radix)`	Available in CLDC.
`short shortValue()`	Available in CLDC.
`String toString()`	Available in CLDC.

`static String toString(shorts)`	Not available in CLDC.
	
	Workaround:
	
	`(""+ s)`
	
	or
	
	`new Short(s).toString()`

`static Short valueOf(String s)`	Not available in CLDC. Workaround: `new Short(parseShort (s));`
`static Short valueOf (Strings, int radix)`	Not available in CLDC. Workaround: `new Short(parseShort (s, radix));`

String

Table B.82. Methods of the Class `String`

Method	***Availability in CLDC***
`String()`	Available in CLDC.
`String(byte[] bytes)`	Available in CLDC.
`String(byte[] ascii, int hibyte)`	Not available in CLDC (deprecated J2SE constructor).
`String(byte[] bytes, int offset, int length)`	Available in CLDC.
`String(byte[] ascii, int hibyte, int offset, int count)`	Not available in CLDC (deprecated J2SE method).
`String(byte[] bytes, int offset, int length, String enc)`	Available in CLDC.
`String(byte[] bytes, String enc)`	Available in CLDC.
`String(char[] value)`	Available in CLDC.
`String(char[] value, int offset, int count)`	Available in CLDC.
`String(String value)`	Available in CLDC.
`String(StringBuffer buffer)`	Available in CLDC.
`char charAt(int index)`	Available in CLDC.
`int compareTo(Object o)`	Not available in CLDC.
`int compareTo(String anotherString)`	Not available in CLDC. Workaround: Iterate over characters and compare them.
`int compareToIgnoreCase(String str)`	Not available in CLDC.
`String concat(String str)`	Available in CLDC.
`static String copyValueOf (char[] data)`	Not available in CLDC.
`static String copyValueOf(char[] data, int offset, int count)`	Not available in CLDC.
`boolean endsWith(String suffix)`	Available in CLDC.
`boolean equals(Object anObject)`	Available in CLDC.
`boolean equalsIgnoreCase (String anotherString)`	Not available in CLDC. Workaround: `toLower().equalsIgnoreCase (str.toLower);`
`byte[] getBytes()`	Available in CLDC.
`void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)`	Not available in CLDC (deprecated J2SE method).
`byte[] getBytes(String enc)`	Available in CLDC.
`void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`	Available in CLDC.
`int hashCode()`	Available in CLDC.
`int indexOf(int ch)`	Available in CLDC.
`int indexOf(int ch, int fromIndex)`	Available in CLDC.
`int indexOf(String str)`	Available in CLDC.
`int indexOf(String str, int fromIndex)`	Available in CLDC.
`String intern()`	Not available in CLDC.

	A workaround for some purposes of intern may be to store `String`s in a hashtable.
`int lastIndexOf(int ch)`	Available in CLDC.
`int lastIndexOf (int ch, int fromIndex)`	Available in CLDC.
`int lastIndexOf(String str)`	Not available in CLDC.
`int lastIndexOf(String str, int fromIndex)`	Not available in CLDC.
`int length()`	Available in CLDC.
`boolean regionMatches(Boolean ignoreCase, int toffset, String other, int ooffset, int len)`	Available in CLDC.
`boolean regionMatches(int toffset, String other, int ooffset, int len)`	Not available in CLDC.
`String replace (char oldChar, char newChar)`	Available in CLDC.
`boolean startsWith(String prefix)`	Available in CLDC.
`boolean startsWith (String prefix, int toffset)`	Available in CLDC.
`String substring(int beginIndex)`	Available in CLDC.
`String substring (int beginIndex, int endIndex)`	Available in CLDC.
`char[]toCharArray()`	Available in CLDC.
`String toLowerCase()`	Available in CLDC.
`String toLowerCase(Locale locale)`	Not available in CLDC.
`String toString()`	Available in CLDC.
`String toUpperCase()`	Available in CLDC.
`String toUpperCase(Locale locale)`	Not available in CLDC.
`String trim()`	Available in CLDC.
`static String valueOf(boolean b)`	Available in CLDC.
`static String valueOf(char c)`	Available in CLDC.
`static String valueOf(char[] data)`	Available in CLDC.
`static String valueOf(char[] data, int offset, int count)`	Available in CLDC.
`static String valueOf(double d)`	Available in CLDC-NG.
`static String valueOf(float f)`	Available in CLDC-NG.
`static String valueOf(int i)`	Available in CLDC.
`static String valueOf(long l)`	Available in CLDC.
`static String valueOf(Object obj)`	Available in CLDC.

`StringBuffer`

Table B.83. Methods of the Class `StringBuffer`

Method	***Availability in CLDC***
`StringBuffer()`	Available in CLDC.
`StringBuffer(int length)`	Available in CLDC.
`StringBuffer(String str)`	Available in CLDC.
`StringBuffer append(boolean b)`	Available in CLDC.
`StringBuffer append(char c)`	Available in CLDC.
`StringBuffer append(char[] str)`	Available in CLDC.

`StringBuffer append(char[] str, int offset, int len)`	Available in CLDC.
`StringBuffer append(double d)`	Available in CLDC-NG.
`StringBuffer append(float f)`	Available in CLDC-NG.
`StringBuffer append(int i)`	Available in CLDC.
`StringBuffer append(long l)`	Available in CLDC.
`StringBuffer append(Object obj)`	Available in CLDC.
`StringBuffer append(String str)`	Available in CLDC.
`int capacity()`	Available in CLDC.
`char charAt(int index)`	Available in CLDC.
`StringBuffer delete (int start, int end)`	Available in CLDC.
`StringBuffer deleteCharAt (int index)`	Available in CLDC.
`void ensureCapacity (int minimumCapacity)`	Available in CLDC.
`void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`	Available in CLDC.
`StringBuffer insert (int offset, boolean b)`	Available in CLDC.
`StringBuffer insert (int offset, char c)`	Available in CLDC.
`StringBuffer insert (int offset, char[] str)`	Available in CLDC.
`StringBuffer insert(int index, char[] str, int offset, int len)`	Not available in CLDC.
`StringBuffer insert (int offset, double d)`	Available in CLDC-NG.
`StringBuffer insert (int offset, float f)`	Available in CLDC-NG.
`StringBuffer insert (int offset, int i)`	Available in CLDC.
`StringBuffer insert (int offset, long l)`	Available in CLDC.
`StringBuffer insert (int offset, Object obj)`	Available in CLDC.
`StringBuffer insert (int offset, String str)`	Available in CLDC.
`int length()`	Available in CLDC.

`StringBuffer replace (int start, int end, String str)`	Not available in CLDC.
	
	Workaround:
	
	`delete (start, end);`
	`insert (start, str);`

`StringBuffer reverse()`	Available in CLDC.
`void setCharAt(int index, char ch)`	Available in CLDC.
`void setLength(int newLength)`	Available in CLDC.
`String substring(int start)`	Not available in CLDC.
`String substring (int start, int end)`	Not available in CLDC.
`String toString()`	Available in CLDC.

System

Table B.84. Methods of the Class `System`

Method	*Availability in CLDC*
`static void arraycopy(Object src, int src_position, Object dst, int dst_position, int length)`	Available in CLDC.

`static long currentTimeMillis()`	Available in CLDC.
`static void exit(int status)`	Available in CLDC.
`static void gc()`	Available in CLDC.
`static String getenv(String name)`	Not available in CLDC (deprecated J2SE method).
`static Properties getProperties()`	Not available in CLDC.
`static String getProperty (String key)`	Available in CLDC.

`static String getProperty (String key, String def)`	Not available in CLDC.
	
	Workaround:
	
	`Static String getProperty ( String key, String def) {`
	`String val = System.getProperty (key);`
	`return val == null ? def : val;`
	`}`

`static SecurityManager getSecurityManager()`	Not available in CLDC.
`static int identityHashCode (Object x)`	Available in CLDC.
`static void load(String filename)`	Not available in CLDC.
`static void loadLibrary (String libname)`	Not available in CLDC.
`static String mapLibraryName (String libname)`	Not available in CLDC.
`static void runFinalization()`	Not available in CLDC.
`static void runFinalizersOnExit (boolean value)`	Not available in CLDC (deprecated J2SE method).
`static void setErr(PrintStream err)`	Not available in CLDC.
`static void setIn(InputStream in)`	Not available in CLDC.
`static void setOut(PrintStream out)`	Not available in CLDC.
`static void setProperties (Properties props)`	Not available in CLDC.
`static String setProperty (String key, String value)`	Not available in CLDC.
`static void setSecurityManager (SecurityManager s)`	Not available in CLDC.

Thread

Table B.85. Methods of the Class Thread

Method	***Availability in CLDC***
`Thread()`	Available in CLDC.
`Thread(Runnable target)`	Available in CLDC.
`Thread(Runnable target, String name)`	Available in CLDC-NG.
`Thread(String name)`	Available in CLDC-NG.
`Thread(ThreadGroup group, Runnable target)`	Not available in CLDC, because `ThreadGroup` is not supported.
`Thread(ThreadGroup group, Runnable target, String name)`	Not available in CLDC, because `ThreadGroup` is not supported.

`Thread(ThreadGroup group,String name)`	Not available in CLDC, because `ThreadGroup` is not supported.
`static int activeCount()`	Available in CLDC.
`void checkAccess()`	Not available in CLDC.
`int countStackFrames()`	Not available in CLDC (deprecated J2SE method).
`static Thread currentThread()`	Available in CLDC.
`void destroy()`	Not available in CLDC.
`static void dumpStack()`	Not available in CLDC.
`static int enumerate (Thread[] tarray)`	Not available in CLDC.
`ClassLoader getContextClassLoader()`	Not available in CLDC.
`String getName()`	Available in CLDC-NG.
`int getPriority()`	Available in CLDC.
`ThreadGroup getThreadGroup()`	Not available in CLDC.
`void interrupt()`	Not available in CLDC.
`static boolean interrupted()`	Not available in CLDC.
`boolean isAlive()`	Available in CLDC.
`boolean isDaemon()`	Available in CLDC-NG.
`boolean isInterrupted()`	Not available in CLDC.
`void join()`	Available in CLDC.
`void join(long millis)`	Not available in CLDC.
`void join(long millis, int nanos)`	Not available in CLDC.
`void resume()`	Not available in CLDC (deprecated J2SE method).
`void run()`	Available in CLDC.
`void setContextClassLoader (ClassLoader cl)`	Not available in CLDC.
`void setDaemon(boolean on)`	Available in CLDC-NG.
`void setName(String name)`	Not available in CLDC.
`void setPriority(int newPriority)`	Available in CLDC.
`static void sleep(long millis)`	Available in CLDC.
`static void sleep (long millis, int nanos)`	Not available in CLDC.
`void start()`	Available in CLDC.
`void stop()`	Not available in CLDC (deprecated J2SE method).
`void stop(Throwable obj)`	Not available in CLDC (deprecated J2SE method).
`void suspend()`	Not available in CLDC.
`String toString()`	Available in CLDC.
`static void yield()`	Available in CLDC.

`Throwable`

Table B.86. Methods of the Class `Throwable`

Method	***Availability in CLDC***
`Throwable()`	Available in CLDC.
`Throwable(String message)`	Available in CLDC.
`Throwable fillInStackTrace()`	Not available in CLDC.

`String getLocalizedMessage()`	Not available in CLDC.
`String getMessage()`	Available in CLDC.
`void printStackTrace()`	Available in CLDC.
`void printStackTrace(PrintStream s)`	Not available in CLDC.
`void printStackTrace(PrintWriter s)`	Not available in CLDC.
`String toString()`	Available in CLDC.

java.lang.ref

Table B.87. Classes of the java.lang.ref Package	
J2SE Classes	***Availability in PDAP***
PhantomReference	Not available in CLDC.
Reference	Partially contained in CLDC-NG; see Table B.88 for details.
ReferenceQueue	Not available in CLDC.
SoftReference	Not available in CLDC.
WeakReference	Partially contained; see Table B.89 for details.

`Reference`

Table B.88. Methods of the Class `Reference`	
Method	***Availability in CLDC***
void clear()	Available in CLDC-NG.
boolean enqueue()	Not available in CLDC.
Object get()	Available in CLDC-NG.
boolean isEnqueued()	Not available in CLDC.

`WeakReference`

Table B.89. Methods of the Class `WeakReference`	
Method	***Availability in CLDC***
WeakReference(Object referent)	Available in CLDC-NG.
WeakReference(Object referent, ReferenceQueue q)	Not available in CLDC.

java.lang.reflect

The package java.lang.reflect includes the InvocationTargetException in PDAP only. No interfaces or classes are supported.

java.net

The package `java.net` supports the `URL` class and the `MalformedURLException` only. The following table shows the methods supported by the `URL` class.

URL

Table B.90. Methods of the Class URL

Method	***Availability in PDAP***
`URL(String spec)`	Available in PDAP.
`URL(String protocol, String host, int port, String file)`	Available in PDAP.
`URL(String protocol, String host, int port, String file, URLStreamHandler handler)`	Not available in PDAP.
`URL(String protocol, String host, String file)`	Available in PDAP.
`URL(URL context, String spec)`	Available in PDAP.
`URL(URL context, String spec, URLStreamHandler handler)`	Not available in PDAP.
`boolean equals(Object obj)`	Available in PDAP.
`String getAuthority()`	Available in PDAP.
`Object getContent()`	Not available in PDAP.
`Object getContent(Class[] classes)`	Not available in PDAP.
`String getFile()`	Available in PDAP.
`String getHost()`	Available in PDAP.
`String getPath()`	Available in PDAP.
`int getPort()`	Available in PDAP.
`String getProtocol()`	Available in PDAP.
`String getQuery()`	Available in PDAP.
`String getRef()`	Available in PDAP.
`String getUserInfo()`	Available in PDAP.
`int hashCode()`	Available in PDAP.
`URLConnection openConnection()`	Not available in PDAP.
`InputStream openStream()`	Not available in PDAP.

`boolean sameFile(URL other)`	Available in PDAP.
`protected void set(String protocol, String host, int port, String file, String ref)`	Available in PDAP.
`protected void set(String protocol, String host, int port, String authority, String userInfo, String path, String query, String ref)`	Available in PDAP.
`static void setURLStreamHandlerFactory (URLStreamHandlerFactory fac)`	Not available in PDAP.
`String toExternalForm()`	Available in PDAP.
`String toString()`	Available in PDAP.

java.util

Table B.91. Interfaces of the `java.util` Package

J2SE Interface	*Availability in CLDC/PDAP*
`Collection`	Not available in CLDC. Workaround: `Vector`
`Comparator`	Not available in CLDC.
`Enumeration`	Fully available in CLDC.
`EventListener`	Fully available in PDAP.
`Iterator`	Not available in CLDC. Workaround: `Enumeration`
`List`	Not available in CLDC. Workaround: `Vector`
`ListIterator`	Not available in CLDC.
`Map`	Not available in CLDC. Workaround: `Hashtable`
`Map.Entry`	Not available in CLDC.
`Observer`	Not available in CLDC.
`Set`	Not available in CLDC.
`SortedMap`	Not available in CLDC.
`SortedSet`	Not available in CLDC.

Table B.92. Classes of the `java.util` Package

J2SE Class	*Availability in CLDC/PDAP*
`AbstractCollection`	Not available in CLDC.
`AbstractList`	Not available in CLDC.
`AbstractMap`	Not available in CLDC.
`AbstractSequentialList`	Not available in CLDC.
`AbstractSet`	Not available in CLDC.
`ArrayList`	Not available in CLDC. Workaround: `Vector`
`Arrays`	Not available in CLDC.
`BitSet`	Not available in CLDC.
`Calendar`	Partially contained; see Table B.94 for details.
`Collections`	Not available in CLDC.
`Date`	Partially contained; see Table B.95 for details.
`Dictionary`	Not available in CLDC. Workaround: `Hashtable`
`EventObject`	Fully available in PDAP.
`GregorianCalendar`	Not available in CLDC.
`HashMap`	Not available in CLDC. Workaround: `Hashtable`

HashSet	Not available in CLDC.
Hashtable	Partially contained; see Table B.96 for details.
LinkedList	Not available in CLDC. Workaround: Vector
ListResourceBundle	Not available in CLDC.
Locale	Partially contained in PDAP; see Table B.97 for details.
Observable	Not available in CLDC.
Properties	Not available in CLDC. Workaround: Hashtable
PropertyPermission	Not available in CLDC.
PropertyResourceBundle	Not available in CLDC.
Random	Partially contained; see Table B.98 for details.
ResourceBundle	Not available in CLDC.
SimpleTimeZone	Not available in CLDC.
Stack	Fully available in CLDC.
StringTokenizer	Not available in CLDC.
Timer	Partially contained; see Table B.99 for details. This class is an MIDP-specific addition to CLDC.
TimerTask	Fully available in CLDC. This class is an MIDP-specific addition to CLDC.
TimeZone	Partially contained; see Table B.100 for details.
TreeMap	Not available in CLDC. Workaround: Hashtable
TreeSet	Not available in CLDC.
Vector	Partially contained; see Table B.101 for details.
WeakHashMap	Not available in CLDC.

Table B.93. Exceptions of the java.util Package

J2SE Exception	***Availability in CLDC***
ConcurrentModificationException	Not available in CLDC.
EmptyStackException	Available in CLDC.
MissingResourceException	Not available in CLDC.
NoSuchElementException	Available in CLDC.
TooManyListenersException	Not available in CLDC.

Calendar

Table B.94. Methods of the Class Calendar

Method	***Availability in CLDC***
protected Calendar()	Available in CLDC.
protected Calendar (TimeZone zone, Locale aLocale)	Not available in CLDC.
abstract void add(int field, int amount)	Not available in CLDC.
boolean after(Object when)	Available in CLDC.
boolean before(Object when)	Available in CLDC.
void clear()	Not available in CLDC.

`void clear(int field)`	Not available in CLDC.
`Object clone()`	Not available in CLDC.
`protected void complete()`	Not available in CLDC.
`protected abstract void computeFields()`	Not available in CLDC.
`protected abstract void computeTime()`	Not available in CLDC.
`boolean equals(Object obj)`	Available in CLDC.
`int get(int field)`	Available in CLDC.

`int getActualMaximum(int field)`	Not available in CLDC.
	
	Workaround for `DAY_OF_MONTH`:
	
	`private static int`
	`daysInMonth[] = {`
	`31, 28, 31, 30, 31, 30,`
	`31, 31, 30, 31, 30, 31} ;`
	`public static int`
	`daysInMonth (Calendar`
	`calendar) {`
	`int year = calendar.get`
	`(Calendar.YEAR);`
	`int month = calendar.get`
	`(Calenadar.MONTH);`
	`int days = daysInMonth`
	`[month-Calendar.JANUARY];`
	`if (month ==`
	`Calendar.FEBRUARY`
	`&& (year % 4 == 0`
	`&& (!(year % 100 == 0)`
	`|| (year % 400 ==`
	`0)))`
	`days++;`
	`return days;`

`int getActualMinimum(int field)`	Not available in CLDC.
`static Locale[] getAvailableLocales()`	Not available in CLDC.
`int getFirstDayOfWeek()`	Not available in CLDC.
`abstract int getGreatestMinimum (int field)`	Not available in CLDC.
`static Calendar getInstance()`	Available in CLDC.
`static Calendar getInstance (Locale aLocale)`	Not available in CLDC.
`static Calendar getInstance (TimeZone zone)`	Available in CLDC.
`static Calendar getInstance (TimeZone zone, Locale aLocale)`	Not available in CLDC.
`abstract int getLeastMaximum (int field)`	Not available in CLDC.
`abstract int getMaximum(int field)`	Not available in CLDC.
`int getMinimalDaysInFirstWeek()`	Not available in CLDC.
`abstract int getMinimum(int field)`	Not available in CLDC.
`Date getTime()`	Available in CLDC.
`protected long getTimeInMillis()`	Available in CLDC.
`TimeZone getTimeZone()`	Available in CLDC.
`int hashCode()`	Not available in CLDC.

`protected int internalGet (int field)`	Not available in CLDC.
`boolean isLenient()`	Not available in CLDC.
`boolean isSet(int field)`	Not available in CLDC.
`abstract void roll (int field, boolean up)`	Not available in CLDC.
`void roll(int field, int amount)`	Not available in CLDC.
`void set(int field, int value)`	Available in CLDC.
`void set(int year, int month, int date)`	Not available in CLDC.
`void set(int year, int month, int date, int hour, int minute)`	Not available in CLDC.
`void set(int year, int month, int date, int hour, int minute, int second)`	Not available in CLDC.
`void setFirstDayOfWeek(int value)`	Not available in CLDC.
`void setLenient(boolean lenient)`	Not available in CLDC.
`void setMinimalDaysInFirstWeek (int value)`	Not available in CLDC.
`void setTime(Date date)`	Available in CLDC.
`protected void setTimeInMillis (long millis)`	Available in CLDC.
`void setTimeZone(TimeZone value)`	Available in CLDC.
`String toString()`	Not available in CLDC.

`Date`

Table B.95. Methods of the Class `Date`

Method	***Availability in CLDC***
`Date()`	Available in CLDC.
`Date(int year, int month, int date)`	Not available in CLDC (deprecated J2SE method).
`Date(int year, int month, int date, int hrs, int min)`	Not available in CLDC (deprecated J2SE method).
`Date(int year, int month, int date, int hrs, int min, int sec)`	Not available in CLDC (deprecated J2SE method).
`Date(long date)`	Available in CLDC.
`Date(String s)`	Not available in CLDC (deprecated J2SE method).
`Boolean after(Date when)`	Not available in CLDC.
`Boolean before(Date when)`	Not available in CLDC.
`Object clone()`	Not available in CLDC.
`int compareTo(Date anotherDate)`	Not available in CLDC.
`int compareTo(Object o)`	Not available in CLDC.
`Boolean equals(Object obj)`	Available in CLDC.
`int getDate()`	Not available in CLDC (deprecated J2SE method).
`int getDay()`	Not available in CLDC (deprecated J2SE method).
`int getHours()`	Not available in CLDC (deprecated J2SE method).
`int getMinutes()`	Not available in CLDC (deprecated J2SE method).

int getMonth()	Not available in CLDC (deprecated J2SE method).
int getSeconds()	Not available in CLDC (deprecated J2SE method).
long getTime()	Available in CLDC.
int getTimezoneOffset()	Not available in CLDC (deprecated J2SE method).
int getYear()	Not available in CLDC (deprecated J2SE method).
int hashCode()	Available in CLDC.
static long parse(String s)	Not available in CLDC (deprecated J2SE method).
void setDate(int date)	Not available in CLDC (deprecated J2SE method).
void setHours(int hours)	Not available in CLDC (deprecated J2SE method).
void setMinutes(int minutes)	Not available in CLDC (deprecated J2SE method).
void setMonth(int month)	Not available in CLDC (deprecated J2SE method).
void setSeconds(int seconds)	Not available in CLDC (deprecated J2SE method).
void setTime(long time)	Available in CLDC.
void setYear(int year)	Not available in CLDC (deprecated J2SE method).
String toGMTString()	Not available in CLDC (deprecated J2SE method).
String toLocaleString()	Not available in CLDC (deprecated J2SE method).
String toString()	Not available in CLDC.
static long UTC(int year, int month, int date, int hrs, int min, int sec)	Not available in CLDC (deprecated J2SE method).

Hashtable

Table B.96. Methods of the Class Hashtable

Method	***Availability in CLDC***
Hashtable()	Available in CLDC.
Hashtable(int initialCapacity)	Available in CLDC.
Hashtable(int initialCapacity, float loadFactor)	Not available in CLDC.
Hashtable(Map t)	Not available in CLDC.
void clear()	Available in CLDC.
Object clone()	Not available in CLDC. Workaround: Copy the Hashtable using keys() and elements() enumeration's
boolean contains(Object value)	Available in CLDC.
boolean containsKey(Object key)	Available in CLDC.
boolean containsValue(Object value)	Not available in CLDC. Workaround: Search for the value using the elements() enumeration.
Enumeration elements()	Available in CLDC.

`Set entrySet()`	Not available in CLDC
`boolean equals(Object o)`	Not available in CLDC
`Object get(Object key)`	Available in CLDC.
`int hashCode()`	Not available in CLDC
`boolean isEmpty()`	Available in CLDC.
`Enumeration keys()`	Available in CLDC.
`Set keySet()`	Not available in CLDC
`Object put(Object key, Object value)`	Available in CLDC.
`void putAll(Map t)`	Not available in CLDC
`protected void rehash()`	Available in CLDC.
`Object remove(Object key)`	Available in CLDC.
`int size()`	Available in CLDC.
`String toString()`	Available in CLDC.
`Collection values()`	Not available in CLDC

`Locale`

Table B.97. Methods of the class `Locale`

Method	***Availability in PDAP***
`Locale(String language, String country)`	Available in PDAP.
`Locale(String language, String country, String variant)`	Available in PDAP.
`Object clone()`	Not available in PDAP.
`boolean equals(Object obj)`	Available in PDAP.
`static Locale[] getAvailableLocales()`	Available in PDAP.
`String getCountry()`	Available in PDAP.
`static Locale getDefault()`	Available in PDAP.
`String getDisplayCountry()`	Not available in PDAP.
`String getDisplayCountry(Locale inLocale)`	Not available in PDAP.
`String getDisplayLanguage()`	Not available in PDAP.
`String getDisplayLanguage(Locale inLocale)`	Not available in PDAP.
`String getDisplayName()`	Not available in PDAP.
`String getDisplayName(Locale inLocale)`	Not available in PDAP.
`String getDisplayVariant()`	Not available in PDAP.
`String getDisplayVariant(Locale inLocale)`	Not available in PDAP.
`String getISO3Country()`	Not available in PDAP.
`String getISO3Language()`	Not available in PDAP.
`static String[] getISOCountries()`	Not available in PDAP.

`static String[] getISOLanguages()`	Not available in PDAP.
`String getLanguage()`	Available in PDAP.
`String getVariant()`	Available in PDAP.
`int hashCode()`	Available in PDAP.
`static void setDefault(Locale newLocale)`	Not available in PDAP.
`String toString()`	Available in PDAP.

Random

Table B.98. Methods of the Class Random

Method	***Availability in CLDC***
`Random()`	Available in CLDC.
`Random(long seed)`	Available in CLDC.
`protected int next(int bits)`	Available in CLDC.
`boolean nextBoolean()`	Not available in CLDC. Workaround: `(nextInt() & 1) == 0`
`void nextBytes(byte[] bytes)`	Not available in CLDC.
`double nextDouble()`	Available in CLDC-NG.
`float nextFloat()`	Available in CLDC-NG.
`double nextGaussian()`	Not available in CLDC.
`int nextInt()`	Available in CLDC.
`int nextInt(int n)`	Not available in CLDC.
`long nextLong()`	Available in CLDC.
`void setSeed(long seed)`	Available in CLDC.

Timer

Table B.99. Methods of the Class Timer

Method	***Availability in CLDC***
`Timer()`	Available in CLDC.
`Timer(boolean isDaemon)`	Not available in CLDC because daemon threads are not available.
`void cancel()`	Available in CLDC.
`void schedule(TimerTask task, Date time)`	Available in CLDC.
`void schedule(TimerTask task, Date firstTime, long period)`	Available in CLDC.
`void schedule(TimerTask task, long delay)`	Available in CLDC.
`void schedule(TimerTask task, long delay, long period)`	Available in CLDC.
`void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)`	Available in CLDC.
`void scheduleAtFixedRate(TimerTask task, long delay, long period)`	Available in CLDC.

TimeZone

Table B.100. Methods of the Class TimeZone	
Method	***Availability in CLDC***
`TimeZone()`	Available in CLDC.
`Object clone()`	Not available in CLDC.
`static String[] getAvailableIDs()`	Available in CLDC.
`static String[] getAvailableIDs (int rawOffset)`	Not available in CLDC.
`static TimeZone getDefault()`	Available in CLDC.
`String getDisplayName()`	Not available in CLDC.
`String getDisplayName (boolean daylight, int style)`	Not available in CLDC.
`String getDisplayName(boolean daylight, int style, Locale locale)`	Not available in CLDC.
`String getDisplayName (Locale locale)`	Not available in CLDC.
`String getID()`	Available in CLDC.
`abstract int getOffset(int era, int year, int month, int day, int dayOfWeek, int milliseconds)`	Available in CLDC.
`abstract int getRawOffset()`	Available in CLDC.
`Static TimeZone getTimeZone (String ID)`	Available in CLDC.
`boolean hasSameRules (TimeZone other)`	Not available in CLDC.
`abstract boolean inDaylightTime (Date date)`	Not available in CLDC.
`static void setDefault (TimeZone zone)`	Not available in CLDC.
`void setID(String ID)`	Not available in CLDC.
`abstract void setRawOffset (int offsetMillis)`	Not available in CLDC. Available in CLDC.
`abstract boolean useDaylightTime()`	

Vector

Table B.101. Methods of the Class Vector	
Method	***Availability in CLDC***
`Vector()`	Available in CLDC.
`Vector(Collection c)`	Not available in CLDC.

`Vector(int initialCapacity)`	Available in CLDC.
`Vector(int initialCapacity, int capacityIncrement)`	Available in CLDC.
`void add(int index, Object element)`	Not available in CLDC. Workaround: `insertElementAt (Object element, int index);`
`boolean add(Object o)`	Not available in CLDC. Workaround: `addElement (Object o);`
`boolean addAll(Collection c)`	Not available in CLDC.
`boolean addAll(int index, Collection c)`	Not available in CLDC.
`void addElement(Object obj)`	Available in CLDC.
`int capacity()`	Available in CLDC.
`void clear()`	Not available in CLDC. Workaround: `removeAllElements()`
`Object clone()`	Not available in CLDC.
`boolean contains(Object elem)`	Available in CLDC.
`boolean containsAll(Collection c)`	Not available in CLDC.
`void copyInto(Object[] anArray)`	Available in CLDC.
`Object elementAt(int index)`	Available in CLDC.
`Enumeration elements()`	Available in CLDC.
`void ensureCapacity (int minCapacity)`	Available in CLDC.
`boolean equals(Object o)`	Not available in CLDC.
`Object firstElement()`	Available in CLDC.
`Object get(int index)`	Not available in CLDC. Workaround: `elementAt (index)`
`int hashCode()`	Not available in CLDC.
`int indexOf(Object elem)`	Available in CLDC.
`int indexOf(Object elem, int index)`	Available in CLDC.
`void insertElementAt (Object obj, int index)`	Available in CLDC.
`boolean isEmpty()`	Available in CLDC.
`Object lastElement()`	Available in CLDC.
`int lastIndexOf(Object elem)`	Available in CLDC.
`int lastIndexOf (Object elem, int index)`	Available in CLDC.
`Object remove(int index)`	Not available in CLDC. Workaround: `removeElementAt(int index)`
`boolean remove(Object o)`	Not available in CLDC. Workaround: `removeElement (Object o)`

`boolean removeAll(Collection c)`	Not available in CLDC.
`void removeAllElements()`	Available in CLDC.
`boolean removeElement(Object obj)`	Available in CLDC.
`void removeElementAt(int index)`	Available in CLDC.
`protected void removeRange (int fromIndex, int toIndex)`	Not available in CLDC.
`boolean retainAll(Collection c)`	Not available in CLDC.
`Object set (int index, Object element)`	Not available in CLDC. Workaround: `setElementAt(Object obj, int index)`
`void setElementAt (Object obj, int index)`	Available in CLDC.
`void setSize(int newSiz4e)`	Available in CLDC.
`int size()`	Available in CLDC.
`List subList (int fromIndex, int toIndex)`	Not available in CLDC.
`Object[] toArray()`	Not available in CLDC.
`Object[] toArray(Object[] a)`	Not available in CLDC.
`String toString()`	Available in CLDC.
`void trimToSize()`	Available in CLDC.

java.util.jar

The package `java.util.jar`, which provides an API for reading and writing JAR-archives and additional manifest files, is not supported in CLDC.

java.util.zip

The package `java.util.zip`, which provides an API for reading and writing standard ZIP and GZIP archives, is not supported in CLDC.

Packages not Available in CLDC

Table B.102. Unavailable J2SE Packages

J2SE Package	*J2ME Alternative*
`java.applet`	Not available in CLDC. Use the `MIDlet` class instead.
`javax.swing`	For MIDP applications, use LCDUI. For PDAP applications, use AWT.