[Table of Contents](#)[Index](#)

Programming Wireless Devices with the Java 2 Platform, Micro Edition, Second Edition

By [Roger Riggs](#), [Antero Taivalsaari](#), [Jim Van Peursem](#), [Jyri Huopaniemi](#),
[Mark Patel](#), [Aleksi Uotila](#), [Jim Holliday](#) Editor

[START READING](#)

Publisher: Addison Wesley

Pub Date: June 13, 2003

ISBN: 0-321-19798-4

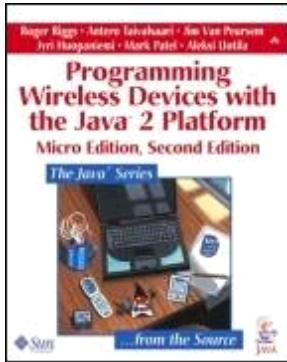
Pages: 464

This book presents the Java(TM) 2 Platform, Micro Edition (J2ME(TM)) standards that support the development of applications for consumer devices such as mobile phones, two-way pagers, and wireless personal organizers. To create these standards, Sun collaborated with such consumer device companies as Motorola, Nokia, NTT DoCoMo, Research In Motion, Samsung, Siemens, Sony Ericsson, and many others. The result is a highly portable, small-footprint application development environment that brings the unique capabilities of Java technology, including platform independence and enhanced security, to the rapidly growing wireless market.

This definitive Java(TM) Series guide provides a programmer's introduction to the Java 2 Platform, Micro Edition. It presents a general description of wireless technology and an overview of the J2ME platform. In addition, the book details the Connected Limited Device Configuration (CLDC) version 1.1 and the Mobile Information Device Profile (MIDP) version 2.0, the standards that define the Java platform features and libraries for wireless, resource-constrained devices.

Written by a team of authors that includes the original J2ME technology experts from Sun, Motorola, and Nokia, this book provides a description of the Java 2 Platform, Micro Edition, as well as practical implementation advice.

The Java(TM) Series is supported, endorsed, and authored by the creators of the Java technology at Sun Microsystems, Inc. It is the official place to go for complete, expert, and definitive information on Java technology. The books in this Series provide the inside information you need to build effective, robust, and portable applications and applets. The Series is an indispensable resource for anyone targeting the Java(TM) 2 platform.

[Table of Contents](#)[Index](#)

Programming Wireless Devices with the Java 2 Platform, Micro Edition, Second Edition

By [Roger Riggs](#), [Antero Taivalsaari](#), [Jim Van Peursem](#), [Jyri Huopaniemi](#),
[Mark Patel](#), [Aleksi Uotila](#), [Jim Holliday](#) Editor

[START READING](#)

Publisher: Addison Wesley

Pub Date: June 13, 2003

ISBN: 0-321-19798-4

Pages: 464

[Copyright](#)[The Java Series](#)[The Jini Technology Series](#)[The Java Series, Enterprise Edition](#)[Figures](#)[Foreword](#)[Preface](#)[Intended Audience](#)[Objectives of This Book](#)[How This Book Is Organized](#)[Related Literature and Helpful Web Pages](#)[Acknowledgments](#)[Chapter 1. Introduction](#)[Section 1.1. The Wireless Internet Revolution](#)[Section 1.2. Why Java Technology for Wireless Devices?](#)[Section 1.3. A Bit of History](#)[Section 1.4. J2ME Standardization Efforts](#)[Chapter 2. Overview of Java 2 Platform, Micro Edition \(J2ME\)](#)[Section 2.1. Java 2 Platform](#)[Section 2.2. Java 2 Platform, Micro Edition \(J2ME\)](#)[Section 2.3. Key Concepts of the J2ME Architecture](#)[Section 2.4. Evolution of the J2ME Platform](#)[Chapter 3. Goals, Requirements, and Scope](#)[Section 3.1. High-Level Goals](#)[Section 3.2. Target Devices](#)[Section 3.3. General Notes on Consumer Devices and Embedded Systems](#)[Section 3.4. Requirements](#)

[Chapter 4. Connected Limited Device Configuration](#)

[Section 4.1. CLDC Expert Groups](#)

[Section 4.2. CLDC Architecture, Application Model, and Security](#)

[Section 4.3. Java Language Specification Compatibility](#)

[Section 4.4. Java Virtual Machine Specification Compatibility](#)

[Section 4.5. New for CLDC 1.1](#)

[Chapter 5. CLDC Libraries](#)

[Section 5.1. Background and Goals](#)

[Section 5.2. Classes Derived from Java 2 Standard Edition](#)

[Section 5.3. CLDC-Specific Classes](#)

[Section 5.4. New for CLDC 1.1](#)

[Chapter 6. Mobile Information Device Profile](#)

[Section 6.1. MIDP Expert Groups](#)

[Section 6.2. Areas Covered by the MIDP Specification](#)

[Chapter 7. MIDP Application Model](#)

[Section 7.1. MIDlets](#)

[Section 7.2. MIDlet Suites](#)

[Section 7.3. New for MIDP 2.0](#)

[Chapter 8. MIDP User Interface Libraries](#)

[Section 8.1. MIDP UI Compared to Desktop AWT](#)

[Section 8.2. Structure of the MIDP User Interface API](#)

[Section 8.3. Display](#)

[Section 8.4. Displayables](#)

[Section 8.5. Commands](#)

[Section 8.6. Advanced Topics](#)

[Section 8.7. New for MIDP 2.0](#)

[Chapter 9. MIDP High-Level User Interface ?Screen](#)

[Section 9.1. List](#)

[Section 9.2. TextBox](#)

[Section 9.3. Alert](#)

[Section 9.4. Form](#)

[Section 9.5. New for MIDP 2.0](#)

[Chapter 10. MIDP High-Level User Interface ?Form](#)

[Section 10.1. Item](#)

[Section 10.2. StringItem](#)

[Section 10.3. ImageItem](#)

[Section 10.4. TextField](#)

[Section 10.5. DateField](#)

[Section 10.6. ChoiceGroup](#)

[Section 10.7. Gauge](#)

[Section 10.8. CustomItem](#)

[Section 10.9. Form Layout](#)

[Section 10.10. New for MIDP 2.0](#)

[Chapter 11. MIDP Low-Level User Interface Libraries](#)

[Section 11.1. The Canvas API](#)

[Section 11.2. Low-Level API for Events in Canvases](#)

[Section 11.3. Graphics](#)

[Section 11.4. Creating and Using Images](#)

[Section 11.5. Drawing Primitives](#)

[Section 11.6. New for MIDP 2.0](#)

[Chapter 12. MIDP Game API](#)

[Section 12.1. The GameCanvas API](#)

[Section 12.2. Layers](#)

[Section 12.4. TiledLayer](#)
[Section 12.5. LayerManager](#)
[Section 12.6. Collision Detection](#)
[Section 12.7. Sample Code: A Simple Game](#)
[Section 12.8. New for MIDP 2.0](#)

[Chapter 13. MIDP Sound API](#)

[Section 13.1. Overview of the MIDP 2.0 Sound API](#)
[Section 13.2. Player Creation and Management](#)
[Section 13.3. Media Controls](#)
[Section 13.4. Enhanced Media Support Using the Mobile Media API](#)
[Section 13.5. New for MIDP 2.0](#)

[Chapter 14. MIDP Persistence Libraries](#)

[Section 14.1. The Record Management System](#)
[Section 14.2. Manipulating Record Stores and Records](#)
[Section 14.3. Sample Code \(RMSMIDlet.java\)](#)
[Section 14.4. New for MIDP 2.0](#)

[Chapter 15. MIDP Networking and Serial Communications](#)

[Section 15.1. Characteristics of Wireless Data Networks](#)
[Section 15.2. Network Interface Considerations](#)
[Section 15.3. The HttpURLConnection Interface](#)
[Section 15.4. Sample Code \(NetClientMIDlet.java\)](#)
[Section 15.5. SocketConnection](#)
[Section 15.6. ServerSocketConnection](#)
[Section 15.7. UDPDatagramConnection](#)
[Section 15.8. CommConnection](#)
[Section 15.9. New for MIDP 2.0](#)

[Chapter 16. Secure Networking](#)

[Section 16.1. Checking the Security Properties of a Connection](#)
[Section 16.2. HttpsConnection](#)
[Section 16.3. SecureConnection](#)
[Section 16.4. MIDP X.509 Certificate Profile](#)
[Section 16.5. New for MIDP 2.0](#)

[Chapter 17. Event-Driven Application Launch](#)

[Section 17.1. Alarm-Based MIDlet Launch](#)
[Section 17.2. Network-Based MIDlet Launch](#)
[Section 17.3. Listening and Launching](#)
[Section 17.4. Handling Connections after Launch](#)
[Section 17.5. Security of the Push Registry](#)
[Section 17.6. Sample Usage Scenarios](#)
[Section 17.7. New for MIDP 2.0](#)

[Chapter 18. Security for MIDlet Suites](#)

[Section 18.1. Assumptions](#)
[Section 18.2. Sandbox for Untrusted MIDlet Suites](#)
[Section 18.3. Trusted MIDlet Suite Security Model](#)
[Section 18.4. APIs That Are Not Security Sensitive](#)
[Section 18.5. Establishing Trust for MIDlet Suites by Using X.509 PKI](#)
[Section 18.6. Recommended Security Policy for GSM/UMTS Devices](#)
[Section 18.7. New for MIDP 2.0](#)

[Chapter 19. MIDlet Deployment](#)

[Section 19.1. MIDlet Suites](#)
[Section 19.2. MIDP System Software](#)
[Section 19.3. Over-the-Air User-Initiated Provisioning](#)
[Section 19.4. New for MIDP 2.0](#)

[Section 20.1. Timer Support](#)

[Section 20.2. System Properties](#)

[Section 20.3. Application Resource Files](#)

[Section 20.4. Exiting a MIDlet](#)

[Chapter 21. Summary](#)

[References](#)

[Appendix A. CLDC Application Programming Interface](#)

[Almanac Legend](#)

[CLDC Almanac](#)

[Appendix B. MIDP Application Programming Interface](#)

[MIDP Almanac](#)

[Index](#)

[\[Team LiB \]](#)

[!\[\]\(0fb13ad0bfa3d86868cdd3883e5665b3_img.jpg\) PREVIOUS](#)

[!\[\]\(799877f5c2f906134441300079881630_img.jpg\) NEXT](#)

[\[Team LiB \]](#)

[!\[\]\(d0a1791f26d167e866e44ebbf83efebe_img.jpg\) PREVIOUS](#) [!\[\]\(cb1960474df5b19cdeae2009c7323e63_img.jpg\) NEXT](#) 

Copyright

Copyright 2003 Sun Microsystems, Inc.

4150 Network Circle, Santa Clara, California 95054, U.S.A.

All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to implementations of the technology described in this publication. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents, foreign patents, or pending applications. Sun, Sun Microsystems, the Sun logo, J2ME, Java Card, and all Sun and Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact U.S. Corporate and Government Sales, (800) 382-3419, corpsales@pearsontechgroup.com.

For sales outside of the U.S., please contact International Sales, (317) 581-3793, intertntional@pearsontechgroup.com.

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Programming wireless devices with the Java 2 Platform : micro edition /Roger Riggs ... [et al.]

[\[Team LiB \]](#)

[!\[\]\(2b9000c261447981d88674ebdb52dc1e_img.jpg\) PREVIOUS](#) [!\[\]\(27556f02ee62a6967e4a51490c76ce3d_img.jpg\) NEXT](#) 

The Java Series

Lisa Friendly, Series Editor

Tim Lindholm, Technical Editor

Ken Arnold, Technical Editor of The JiniTM Technology Series

Jim Inscore, Technical Editor of The JavaTM Series, Enterprise Edition <http://www.javaseries.com>

Eric Armstrong, Stephanie Bodoff, Debbie Carson, Maydene Fisher, Dale Green, Kim Haase

The Java Web Services Tutorial

Ken Arnold, James Gosling, David Holmes

The Java Programming Language, Third Edition

Cindy Bloch, Annette Wagner

MIDP 2.0 Style Guide

Joshua Bloch

Effective Java Programming Language Guide

Mary Campione, Kathy Walrath, Alison Huml

The Java Tutorial, Third Edition: A Short Course on the Basics

Mary Campione, Kathy Walrath, Alison Huml, Tutorial Team

The Java Tutorial Continued: The Rest of the JDK

Patrick Chan

>The Java Developers Almanac 1.4, Volume 1

Patrick Chan

The Java Developers Almanac 1.4, Volume 2

Patrick Chan, Rosanna Lee

The Java Class Libraries, Second Edition, Volume 2: java.applet, java.awt, java.beans

Patrick Chan, Rosanna Lee, Doug Kramer

The Java Class Libraries, Second Edition, Volume 1: java.io, java.lang, java.math, java.net, java.text, java.util

Patrick Chan, Rosanna Lee, Doug Kramer

The Java Class Libraries, Second Edition, Volume 1: Supplement for the Java 2 Platform, Standard Edition, v1.2

Kirk Chen, Li Gong

Programming Open Service Gateways with Java Embedded Server

Zhiqun Chen

Java Card Technology for Smart Cards: Architecture and Programmer's Guide

Maydene Fisher, Jon Ellis, Jonathan Bruce

[\[Team LiB \]](#)

◀ PREVIOUS NEXT ▶

The Jini Technology Series

Eric Freeman, Susanne Hupfer, Ken Arnold
JavaSpaces Principles, Patterns, and Practice

[\[Team LiB \]](#)

◀ PREVIOUS NEXT ▶

The Java Series, Enterprise Edition

Stephanie Bodoff, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, Beth Stearns
The J2EE Tutorial

Rick Cattell, Jim Inscore, Enterprise Partners
J2EE Technology in Practice: Building Business Applications with the Java 2 Platform, Enterprise Edition

Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, Kim Haase
Java Message Service API Tutorial and Reference: Messaging for the J2EE Platform

Inderjeet Singh, Beth Stearns, Mark Johnson, Enterprise Team
Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition

Vlada Matena, Sanjeev Krishnan, Linda DeMichiel, Beth Stearns
Applying Enterprise JavaBeans 2.1, Second Edition: Component-Based Development for the J2EE Platform

Bill Shannon, Mark Hapner, Vlada Matena, James Davidson, Eduardo Pelegri-Llopert, Larry Cable, Enterprise Team
Java 2 Platform, Enterprise Edition: Platform and Component Specifications

Rahul Sharma, Beth Stearns, Tony Ng
J2EE Connector Architecture and Enterprise Application Integration

[\[Team LiB \]](#)

[!\[\]\(98b0d0ccc757b6bc0d84eb54a134e84b_img.jpg\) PREVIOUS](#) [!\[\]\(71c0e1b873ce5bacd02860899972a812_img.jpg\) NEXT](#) 

Figures

[Figure 2.1](#) Java 2 Platform editions and their target markets

[Figure 2.2](#) Software layers in a J2ME device

[Figure 2.3](#) Relationship between J2ME configurations and Java 2 Standard Edition

[Figure 3.1](#) Downloading customized services

[Figure 3.2](#) CLDC and MIDP target devices

[Figure 3.3](#) One-handed, two-handed, and stylus-operated mobile information devices

[Figure 4.1](#) Architecture overview of a CLDC target device

[Figure 4.2](#) Two-phase class file verification in CLDC

[Figure 5.1](#) Connection interface hierarchy

[Figure 7.1](#) MIDlet states and state transitions

[Figure 8.1](#) MIDP user interface class hierarchy

[Figure 10.1](#) General principle of Form layout

[Figure 10.2](#) Item layout policy dictated by the device implementation

[Figure 10.3](#) Applying vertical and horizontal layout directives

[Figure 11.1](#) Pixel coordinate system showing the pixel at (4,3)

[Figure 11.2](#) Example of the getRGB method

[\[Team LiB \]](#)

[!\[\]\(9215a40c916c06eb7383cdd1fe97b4e5_img.jpg\) PREVIOUS](#) [!\[\]\(75d6a94555ec8d621438fcf7e09fb09a_img.jpg\) NEXT](#) 

Foreword

"If a program is useful,
it will have to be changed."

?CM SIGPLAN Notices (vol 2, no. 2), 1965

"To attain knowledge, add things every day;
to obtain wisdom, remove things every day."

?ao-tsu, Tao Te Ching

"One must learn by doing the thing;
For though you think you know it,
you have no certainty until you try."

?ophocles, Trachiniae

With the delivery of the Java 2 Platform, Micro Edition (J2ME), Java technology has come full circle. The Java technology we know today originally sprang from a project whose goal was to investigate the potential impact of digital technologies outside the mainstream of the computer industry. It was clear that this big expanse of interesting territory was the space close to everyday people. Consumer electronics, telephony, and embedded systems were increasingly becoming a part of the fabric of everyday life.

Being a group with a very hands-on engineering background, we decided to build an artifact as a way to lead our understanding into the details. We built a small (for its day!) handheld device, not unlike today's PDAs. As we considered the implications of this new world, we encountered serious issues with the underlying tools we were using to build the software:

•

Heterogeneity was a fact of life. The consumer world has many different CPU and system architectures. In the desktop world, these differences ("WinTel" versus Macintosh) partition the market based on low-level details that most people know little about. In the consumer/embedded world, there are many more architectures than the two of the desktop world. The fragmented chaos that would ensue would cause serious problems. Heterogeneity becomes an even more urgent issue when these devices are connected in a network and then start sharing software.

•

Reliability is a huge issue. Non-technophiles have a justifiably low tolerance for systems that malfunction. There are a number of areas where low-level issues in the programming language design (memory integrity being one) have a large impact.

•

Security must be addressed. There is no bigger threat to a network than a "teenage guy" out to have fun. (I know, I was one!) Security is not something that can be painted on afterwards; it has to be built in from the

Preface

In the past five years, Sun has collaborated with major consumer device manufacturers and other companies to create a highly portable, secure, small-footprint Java application development environment for resource-constrained, wireless consumer devices such as cellular telephones, two-way pagers, and personal organizers. This work started with the development of a new, small-footprint Java virtual machine called the K Virtual Machine (KVM). Two Java Community Process (JCP) standardization efforts, Connected, Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP), were then carried out to standardize the Java libraries and the associated Java language and virtual machine features across a wide variety of consumer devices.

The first round of the CLDC and MIDP standardization efforts took place during the fall of 1999 and spring of 2000. Twenty-four companies participated in the CLDC 1.0 and MIDP 1.0 standardization efforts directly, and more than five hundred companies and individuals participated indirectly by sending feedback while the standardization efforts were in progress. Major consumer device companies such as Motorola, Nokia, NTT DoCoMo, Palm Computing, Research In Motion (RIM), and Siemens played a key role in these efforts.

After their first release, the CLDC 1.0 and MIDP 1.0 standards have become very popular. The deployment of real-world, Java-enabled wireless devices began in 2000, and the deployments accelerated rapidly in 2001 and 2002, approaching exponential growth. It has been estimated that over 50 million devices supporting the CLDC and MIDP standards were shipped in 2002, and the number is likely to be at least twice as large in 2003. As a result of the widespread acceptance of these standards, major business opportunities are now emerging for Java application developers in the wireless device space.

The second round of the CLDC and MIDP standardization efforts was started in the fall of 2001. The goal of the CLDC 1.1 and MIDP 2.0 efforts was to expand on the success of the original standards, refine the existing feature set, and introduce additional APIs, while keeping a close eye on the strict memory limitations that still constrain the design of wireless devices. More than 60 companies were directly involved in the development of the CLDC 1.1 and MIDP 2.0 specifications, reflecting the broad acceptance and adoption of these standards in the wireless industry.

This book intends to make the results of the standardization work in the wireless Java technology area available to the wider software development community. At the high level, this book combines two Java Community Process Specifications, CLDC 1.1 (JSR 139) and MIDP 2.0 (JSR 118), and presents them as a single monograph in a way that the corresponding Java Community Process (JCP) Specifications cannot accomplish by themselves. We have added a general introduction to the Java 2 Platform, Micro Edition (J2ME), provided more background material, and included a number of small applications to illustrate the use of CLDC and MIDP in the real world. We also provide some guidelines and instructions for getting started with Java 2 Platform, Micro Edition.

A reference implementation of the software discussed in this book is available from Sun Microsystems under the Sun Community Source License (SCSL).

Intended Audience

The book is intended for software developers, content providers, and other professionals who want to develop Java software for resource-constrained, connected devices. The book is also targeted to consumer device manufacturers who want to build small Java Powered devices and would like to integrate a compact Java application development platform in their products.

Objectives of This Book

This book is the definitive statement, "from the source," about the key specifications for Java Powered wireless devices. As such, this book intends to

- provide an overview of Java 2 Platform, Micro Edition (J2ME),
- provide a general introduction to the application development platforms defined by the J2ME standardization efforts,
- explain the technical aspects of the J2ME Connected, Limited Device Configuration version 1.1 (CLDC 1.1),
- explain the technical aspects of the J2ME Mobile Information Device Profile version 2.0 (MIDP 2.0),
- provide sample programs to illustrate the use of CLDC and MIDP, and
- help you get started in writing your own J2ME applications.

[\[Team LiB \]](#)

[!\[\]\(96b3f29ac8767c5aed426b36cd82f536_img.jpg\) PREVIOUS](#) [!\[\]\(fa8b5948abcb405990b3ef581cef0ab6_img.jpg\) NEXT](#) 

How This Book Is Organized

The topics in this book are organized as follows:

- [Chapter 1](#), "Introduction," provides a context for Java 2 Micro Edition and the CLDC and MIDP specifications.
- [Chapter 2](#), "Overview of Java 2 Platform, Micro Edition (J2ME)," provides an overview of Java 2 Micro Edition, its key concepts and components.
- [Chapter 3](#), "Goals, Requirements, and Scope," defines the goals, requirements, and scope of the CLDC and MIDP standardization efforts.
- [Chapter 4](#), "Connected Limited Device Configuration," introduces the CLDC standardization effort and summarizes the supported Java programming language and virtual machine features compared to the Java 2 Platform, Standard Edition.
- [Chapter 5](#), "CLDC Libraries," introduces the Java class libraries defined by the CLDC Specification.
- [Chapter 6](#), "Mobile Information Device Profile," introduces the MIDP standardization effort.
- [Chapter 7](#), "MIDP Application Model," introduces the MIDlet application model defined by the MIDP Specification.
- [Chapter 8](#), "MIDP User Interface Libraries," introduces the user interface libraries defined by the MIDP Specification.
- [Chapter 9](#), "MIDP High-Level User Interface ?Screen," introduces the part of the MIDP high-level user interface revolving around the Screen class.
- [Chapter 10](#), "MIDP High-Level User Interface ?Form," introduces the part of the MIDP high-level user interface revolving around the Form class.
- [Chapter 11](#), "MIDP Low-Level User Interface Libraries," introduces the low-level user interface libraries defined by the MIDP Specification.
- [Chapter 12](#), "MIDP Game API," introduces the game API defined by the MIDP Specification.

[\[Team LiB \]](#)

[!\[\]\(e4f0e151fe7091d65ea97d178b1e424f_img.jpg\) PREVIOUS](#) [!\[\]\(7c8e1511aa58d89938d53b9c55d74d3c_img.jpg\) NEXT](#) 

Related Literature and Helpful Web Pages

The Java Language Specification, Second Edition, by James Gosling, Bill Joy, Guy Steele and Gilad Bracha. Addison-Wesley, 2000, ISBN 0-201-31008-2

The Java Virtual Machine Specification, Second Edition, by Tim Lindholm and Frank Yellin. Addison-Wesley, 1999, ISBN 0-201-43294-3

Programming Wireless Devices with the Java 2 Platform, Micro Edition, First Edition, by Roger Riggs, Antero Taivalsaari, and Mark VandenBrink. Addison-Wesley, 2001, ISBN 0-201-74627-1

MIDP 2.0 Style Guide by Cynthia Bloch and Annette Wagner. Addison-Wesley, 2003, ISBN 0-321-19801-8

Connected, Limited Device Configuration Specification, version 1.0 <http://jcp.org/en/jsr/detail?id=30>

Connected, Limited Device Configuration Specification, version 1.1 <http://jcp.org/en/jsr/detail?id=139>

Mobile Information Device Profile Specification, version 1.0 <http://jcp.org/en/jsr/detail?id=37>

Mobile Information Device Profile Specification, version 2.0 <http://jcp.org/en/jsr/detail?id=118>

Java 2 Micro Edition Product Web Page <http://java.sun.com/products/j2me/>

Connected, Limited Device Configuration (CLDC) Product Web Page <http://java.sun.com/products/cldc/>

Mobile Information Device Profile (MIDP) Product Web Page <http://java.sun.com/products/midp/>

J2ME Wireless Toolkit Product Web Page <http://java.sun.com/products/j2mewtoolkit/>

Web Pages for This Book

Addison-Wesley Page

www.awprofessional.com/titles/0321197984

Sun Microsystems Page

[\[Team LiB \]](#)

[!\[\]\(41ede330716e633900ec1a60a7f07465_img.jpg\) PREVIOUS](#) [!\[\]\(99ae983cae298406fa49f5a7ce293404_img.jpg\) NEXT](#) 

Acknowledgments

Like most books, this book represents the work of many people. In this case, however, an unusually large number of people around the world have worked to make the Java 2 Platform, Micro Edition?nd also this book? reality. In five short years, a humble, two-person research project at Sun Labs grew rapidly into a highly collaborative product development and standardization effort involving hundreds of companies and thousands of people all over the world. The summary that follows is an attempt to give a glimpse into the different groups of people who participated in this journey. To these and many others too numerous to mention, we give our thanks and appreciation for what they did to make these ideas and this book possible. Attempting to name these people in no way diminishes the contributions of those who we also meant to name but in the pressure of time and the failure of memory somehow overlooked.

Many people read draft versions of this book and sent us comments that improved the book substantially. The authors would like to thank all the reviewers for their willingness to send comments and constructive criticism on the various versions of the book and the sample applications. The authors would also like to thank Lisa Friendly for allowing us to publish this book in Sun's Java book series and for lending us capable technical writing resources to finish this book. Jim Holliday, our technical writer and editor at Sun Microsystems, edited various versions of this book tirelessly. Without his expertise in the mysteries of desktop publishing, grammar, and that pesky topic known as punctuation, this work would have been much worse for the wear. Three other people at Sun Microsystems deserve special mention: Cindy Bloch, Senior Technical Writer, who contributed greatly in editing several chapters; Tasneem Sayeed, Staff Engineer, who provided very detailed comments on the various versions of the book; and Tim Dunn, Visual Designer, who devoted many long hours to perfecting the graphics in this book.

Numerous companies have been involved in the standardization efforts related to the Java 2 Platform, Micro Edition. We would like to thank all the CLDC and MIDP expert group members for their active participation and valuable contributions. In addition to the official members of the CLDC and MIDP expert groups, hundreds of other companies and individuals sent us feedback while the CLDC and MIDP standardization efforts were in progress. The authors found it amazing how much of their time people were willing to contribute to ensure the progress of the Java technology in the wireless space.

Someone once said that hardware without software is a space heater. Similarly, without products, the CLDC and MIDP specifications would be limited in their value. The authors would like to thank wireless device manufacturers, wireless network operators, and software developers for widely embracing the CLDC and MIDP standards, thereby allowing software developers all over the world to finally have a common platform for mobile software development.

Various product groups in Sun's Consumer and Mobile Systems Group (CMSG) organization participated in the design and implementation of the CLDC and MIDP reference implementations. The authors would like to thank Sun's CLDC team, MIDP team, and Wireless Toolkit team members who worked on the reference implementations of the standards and products discussed in this book. The TCK (Technology Compatibility Kit) and Quality Assurance teams at CMSG also played a critical role in ensuring the quality and compatibility of the products.

The Nokia authors would like to thank the following Nokia people who have been closely involved in the MIDP 2.0 standardization work: Kari Syst? Kimmo L?t?? Markku Tamaki, Anna Zhuang, and Antti Rantalahti. Special thanks go to all the Nokia Java teams who have made it possible for Nokia to launch over 30 products with J2ME support and to ship tens of millions of J2ME devices.

The Motorola authors would like to thank all the people in Motorola who helped make CLDC and MIDP a success. Special thanks go to the team members in Motorola's WSAS and iDEN groups that had the vision, developed the first CLDC/MIDP device to reach the market, and still continue to push the envelope in terms of performance and capabilities.

Chapter 1. Introduction

[Section 1.1. The Wireless Internet Revolution](#)

[Section 1.2. Why Java Technology for Wireless Devices?](#)

[Section 1.3. A Bit of History](#)

[Section 1.4. J2ME Standardization Efforts](#)

1.1 The Wireless Internet Revolution

The wireless communications industry has seen rapid growth in the past several years. This has made wireless communication one of the fastest growing technology areas in the world. The total number of cellular phone subscribers worldwide exceeded one billion in 2002, and it has been estimated that there will be over 1.5 billion wireless subscribers in the world by 2005. This far exceeds the number of personal computer users in the world, which was estimated to be about 500 million in 2002. The total annual sales of cell phones in the world is expected to grow from about 423 million phones sold in 2002 to nearly 600 million phones sold in 2006.

At the same time, the rapid emergence of the Internet has changed the landscape of modern computing. People have become more and more dependent on the information that is available on the Internet, and they will increasingly want to access the Internet not only from their personal computers and office workstations but also from mobile, wireless devices. Consequently, the rapid and efficient deployment of new wireless data and mobile Internet services has become a high priority for communication equipment manufacturers and telecommunication operators.

The transition to wireless, mobile Internet devices will fundamentally alter the landscape and architecture of communication networks, devices, and services. Unlike in the past, when wireless devices typically came from the factory with a hard-coded feature set, the devices will become more and more customizable. The possibility to download new applications and features over wireless networks will open up completely new possibilities for device manufacturers, network operators, service and content providers, and device users themselves.

The wireless Internet revolution will be facilitated by another important technological advance: the introduction of broadband third- and fourth-generation wireless networks. While current wireless networks have limited data rates, allowing the users to transfer only up to a few tens of kilobits of data per second, broadband wireless networks will provide data rates ranging from hundreds of kilobits up to several megabits per second. This will provide enough bandwidth to transfer photographs, live video, and high-quality audio, as well as to download significantly larger applications and services than today.

All these changes will happen relatively fast. Even though it will still take a few years before broadband wireless networks are in widespread use, it is safe to estimate that the majority of new wireless devices will have high-speed connectivity to the Internet by the end of the decade.

[\[Team LiB \]](#)

[!\[\]\(4cb5c2912b9923108852be895c69764a_img.jpg\) PREVIOUS](#) [!\[\]\(3c3b12093cfdc7b2f25c84e8c10da894_img.jpg\) NEXT](#) 

1.2 Why Java Technology for Wireless Devices?

The wireless Internet revolution will transform wireless devices from voice-oriented communication devices with relatively static, hard-coded functionality into extensible, Internet-enabled devices with advanced data and software capabilities. These devices will need to support dynamic downloading of new software and be capable of running software written not only by the device manufacturers themselves but also by third-party software developers. This will make the devices much more dependent on software and will place a much higher emphasis on software interoperability, security, and reliability.

The Java programming language is ideally suited to become the standard application development language for wireless devices. After all, the Java platform provides a number of important benefits:

- Dynamic delivery of content. New applications, services, and content can be downloaded dynamically over different kinds of networks.
- Security. Class file verification, well-defined application programming interfaces, and security features ensure that third-party applications behave reliably and cannot harm the devices or the networks.
- Cross-platform compatibility. Standardized language features and libraries mean that applications and content can be transferred flexibly between different devices, within constraints of the supported J2ME configuration and profiles (see [Section 2.3](#), "Key Concepts of the J2ME Architecture," for details).
- Enhanced user experience and interactive content. The standards defined for wireless Java technology support sophisticated user interaction and provide compelling graphics capabilities for small devices.
- Offline access. Applications can also be used without active network connection. This reduces transport costs and alleviates the impact of possible network failures.
- The power of a modern object-oriented programming language. The Java programming language has far better abstraction mechanisms and higher-level programming constructs than other languages and tools that are currently used for wireless software development, allowing applications to be developed more efficiently.
- Large developer community. It is estimated that there are more than three million Java software developers worldwide. The Java programming language is rapidly becoming the most popular programming language taught in schools and universities. The developer talent needed for Java software development already exists and is readily available.

Ultimately, Java technology will deliver far more compelling, entertaining, and engaging capabilities to wireless devices. What is particularly important is that this can be accomplished incrementally, by complementing existing technologies and standards, rather than by competing with them. One of the key points we emphasize throughout this book is that we are not defining a new operating system or a complete system software stack for wireless devices.

1.3 A Bit of History

The Java programming language was initially targeted towards consumer devices, especially the interactive TV market. However, over time the Java platform evolved more and more towards the needs of desktop and enterprise computing. Enterprise applications generally require rich library functionality, and over time the Java libraries grew larger and more comprehensive to cater better to the needs of the enterprise market and large server-side applications. However, this evolution made the libraries too large and unsuitable for the majority of small, resource-constrained devices.

In January 1998, the Spotless project was started at Sun Microsystems Laboratories (Sun Labs) to investigate the use of the Java programming language in extremely resource-constrained devices. The research goal of the project was to build a Java runtime environment that would fit in less than one-tenth of the typical size. At the implementation level, the goal was to build a Java virtual machine with the following characteristics:

- small size
- portability
- ease of use and readability of the source code

Small size is important, since the majority of wireless, mobile devices (for example, cell phones) are still very resource-limited and often have only a few tens or hundreds of kilobytes of memory available for applications. Portability, ease of use, and the readability of the source code are equally important. Most embedded device manufacturers need to support dozens or even hundreds of different hardware configurations that run on several different hardware platforms, and it would be not only tedious but also very expensive to spend a lot of time porting and customizing the Java platform implementation to all those hardware configurations and platforms. Also, embedded device manufacturers cannot generally be expected to be experts on the Java programming language or virtual machine. Therefore, the easier the implementation is to understand and use, the faster the device manufacturers will deploy it across their devices.

Even though the Spotless effort was initially a research project, the project group established active contacts with external customers early on. External customers, especially Motorola, played a significant role in convincing Sun to turn the Spotless system from a research project into a commercial product. The product version of the Spotless virtual machine is known today as the K Virtual Machine (KVM). The Spotless system is documented in the Sun Labs technical report *The Spotless System: Implementing a Java System for the Palm Connected Organizer* (Sun Labs Technical Report SMLI TR-99-73).

[\[Team LiB \]](#)

[!\[\]\(8e5f660ab0fc8a458c6b01dae5bd68a8_img.jpg\) PREVIOUS](#) [!\[\]\(c90f77b8ce5e490bbf01b3ab4a85ff14_img.jpg\) NEXT](#) 

1.4 J2ME Standardization Efforts

Once Motorola, Nokia, NTT DoCoMo, Palm Computing, RIM, Siemens, and other device manufacturers became interested in the KVM development effort, standardization was necessary in order to guarantee interoperability between the different kinds of Java Powered devices from different manufacturers. Two Java Community Process (JCP) standardization efforts were launched in the fall of 1999.

The first of these standardization efforts, Connected, Limited Device Configuration (CLDC), was launched on October 1, 1999. The goal of this effort was to define the "lowest common denominator" Java platform for a wide variety of small, connected, resource-constrained devices. This specification defines the minimum required complement of Java technology components and libraries for small devices. Java programming language and virtual machine features, core libraries, input/output, networking, and security are the primary topics addressed by the CLDC Specification. The CLDC standard does not target any specific device category. Rather, it defines a general-purpose building block on top of which more device category specific profiles are defined. Eighteen companies participated in the CLDC 1.0 (JSR 30) expert group work.

The second standardization effort, Mobile Information Device Profile (MIDP), was started in November 1999. That effort was based on the platform defined by the CLDC effort, adding features and APIs that focus specifically on two-way wireless communication devices such as cell phones and two-way pagers. Application model, user interface, networking, and storage APIs are the primary focus areas of the MIDP Specification. Twenty-two companies participated in the MIDP 1.0 (JSR 37) expert group work.

After becoming widely accepted in the marketplace, second-generation CLDC and MIDP standardization efforts were launched in the fall of 2001.

The CLDC 1.1 (JSR 139) effort was started in September 2001. CLDC 1.1 adds a number of features that improve the compatibility of the J2ME platform with the full Java 2 Platform, Standard Edition (J2SE). The most significant new feature added in CLDC 1.1 is floating point support, but there are a number of other important features as well. The CLDC standardization effort is described in more detail in [Chapter 4](#), "Connected Limited Device Configuration," and [Chapter 5](#), "CLDC Libraries."

The MIDP 2.0 (JSR 118) effort was started in August 2001. MIDP 2.0 adds a number of important new libraries such as a Sound API and a Game API, as well as provides important refinements and extensions to the existing MIDP APIs. The MIDP standardization effort is discussed in [Chapter 6](#), "Mobile Information Device Profile" and subsequent chapters.

The general framework for CLDC and MIDP, as well as other Java technology standardization efforts in the small device space, is known as Java 2 Platform, Micro Edition (J2ME) or simply Java 2 Micro Edition. An introduction to Java 2 Micro Edition is provided in the next chapter.

In addition to the CLDC and MIDP standardization efforts, there are a large number of additional J2ME standardization efforts that build upon these core standards. A summary of the J2ME optional packages created by the additional standardization efforts will be provided in [Section 2.4](#), "Evolution of the J2ME Platform."

Chapter 2. Overview of Java 2 Platform, Micro Edition (J2ME)

[Section 2.1. Java 2 Platform](#)

[Section 2.2. Java 2 Platform, Micro Edition \(J2ME\)](#)

[Section 2.3. Key Concepts of the J2ME Architecture](#)

[Section 2.4. Evolution of the J2ME Platform](#)

[\[Team LiB \]](#)

[!\[\]\(aee8be35822e3e1cd71695d6362eeb1d_img.jpg\) PREVIOUS](#) [!\[\]\(845c674af7ec1ea17faf4036fe88840b_img.jpg\) NEXT](#) 

2.1 Java 2 Platform

Recognizing that one size does not fit all, Sun Microsystems has grouped Java technologies into three editions, each aimed at a specific area of today's vast computing industry:

- Java 2 Platform, Enterprise Edition (J2EE) for enterprises needing to serve their customers, suppliers, and employees with scalable server solutions.
- Java 2 Platform, Standard Edition (J2SE) for the familiar and well-established desktop computer market.
- Java 2 Platform, Micro Edition (J2ME) for the combined needs of:
 - consumer and embedded device manufacturers who build a diversity of information devices,
 - service providers who wish to deliver content to their customers over those devices, and
 - content creators who want to make compelling content for small, resource-constrained devices.

Each Java platform edition defines a set of technologies that can be used with a particular product:

- Java Virtual Machines that fit inside a wide range of computing devices,
- libraries and APIs specialized for each kind of computing device, and
- tools for deployment and device configuration.

[Figure 2.1](#) illustrates the Java 2 Platform editions and their target markets, starting from the high-end platforms on the left and moving towards low-end platforms on the right. Basically, five target markets or broad device categories are identified. Servers and enterprise computers are supported by Java 2 Enterprise Edition, and desktop and personal computers by Java 2 Standard Edition. Java 2 Micro Edition is divided broadly into two categories that focus on "high-end" and "low-end" consumer devices. Java 2 Micro Edition is discussed in more detail later in this chapter. Finally, the Java Card standard focuses on the smart card market.

Figure 2.1. Java 2 Platform editions and their target markets

[\[Team LiB \]](#)

[!\[\]\(0dc690309bcc577bffe1c73810af57c4_img.jpg\) PREVIOUS](#) [!\[\]\(80dcce03b8fbe6ae6dc71dfcb75cfc28_img.jpg\) NEXT](#) 

2.2 Java 2 Platform, Micro Edition (J2ME)

Java 2 Platform, Micro Edition (henceforth referred to as Java 2 Micro Edition or J2ME) specifically addresses the large, rapidly growing consumer space, which covers a range of devices from tiny commodities, such as pagers, all the way up to the TV set-top box, an appliance almost as powerful as a desktop computer. Like the larger Java editions, Java 2 Micro Edition aims to maintain the qualities that Java technology has become known for, including built-in consistency across products, portability of code, safe network delivery, and upward scalability.

The high-level idea behind J2ME is to provide comprehensive application development platforms for creating dynamically extensible, networked devices and applications for the consumer and embedded market. J2ME enables device manufacturers, service providers, and content creators to capitalize on new market opportunities by developing and deploying compelling new applications and services to their customers worldwide. Furthermore, J2ME allows device manufacturers to open up their devices for widespread third-party application development and dynamically downloaded content without losing the security or the control of the underlying manufacturer-specific platform.

At a high level, J2ME is targeted at two broad categories of products:

- "High-end" consumer devices. In [Figure 2.1](#), this category is represented by the grouping labeled CDC (Connected Device Configuration). Typical examples of devices in this category include TV set-top boxes, Internet TVs, Internet-enabled screenphones, high-end wireless communicators, and automobile entertainment/navigation systems. These devices have a large range of user interface capabilities, total memory budgets starting from about two to four megabytes, and persistent, high-bandwidth network connections, often using TCP/IP.
- "Low-end" consumer devices. In [Figure 2.1](#), this category is represented by the grouping labeled CLDC (Connected, Limited Device Configuration). Cell phones, pagers, and personal organizers are examples of devices in this category. These devices have simple user interfaces (compared to desktop computer systems), minimum memory budgets starting from about 128?56 kilobytes, and low bandwidth, intermittent network connections. In this category of products, network communication is often not based on the TCP/IP protocol suite. Most of these devices are battery-operated.

The line between these two categories is fuzzy and becoming more so every day. As a result of the ongoing technological convergence in the computer, telecommunication, consumer electronics, and entertainment industries, there will be less technical distinction between general-purpose computers, personal communication devices, consumer electronics devices, and entertainment devices. Also, future devices are more likely to use wireless connectivity instead of traditional fixed or wired networks. In practice, the line between the two categories is defined more by the memory budget, bandwidth considerations, battery power consumption, and physical screen size of the device rather than by its specific functionality or type of connectivity.

Because of strict manufacturing cost constraints, the majority of high-volume wireless devices today, such as cell phones, belong to the low-end consumer device category. Therefore, this book focuses only on the CLDC and MIDP standards that were specifically designed for that category of products.

[\[Team LiB \]](#)

[!\[\]\(fda2fd5666240df638ac3c5c430323a7_img.jpg\) PREVIOUS](#) [!\[\]\(735ccca6d16f6de990e358706ddc62a5_img.jpg\) NEXT](#) 

2.3 Key Concepts of the J2ME Architecture

While connected consumer devices such as cell phones, pagers, personal organizers, and TV set-top boxes have many things in common, they are also extremely diverse in form, function, and features. Information appliances tend to be special-purpose, limited-function devices. To address this diversity, an essential requirement for the J2ME architecture is not only small size but also modularity and customizability.

In general, serving the information appliance market calls for a large measure of flexibility in how computing technology and applications are deployed. This flexibility is required because of

- the large range of existing device types and hardware configurations,
- the different usage models employed by the devices (key operated, stylus operated, voice operated),
- constantly improving device technology,
- the diverse range of existing applications and features, and
- the need for applications and capabilities to change and grow, often in unforeseen ways, in order to accommodate the future needs of the consumer.

The J2ME architecture is intended to be modular and scalable so that it can support the kinds of flexible deployment demanded by the consumer and embedded markets. To enable this, the J2ME environment provides a range of Java Virtual Machine technologies, each optimized for the different processor types and memory footprints commonly found in the consumer and embedded marketplace.

For low-end, resource-limited consumer products, the J2ME environment supports minimal configurations of the Java Virtual Machine and Java libraries that embody just the essential capabilities of each kind of device. As device manufacturers develop new features in their devices or service providers develop new and exciting applications, these minimal configurations can be expanded with additional libraries that address the needs of a particular market segment. To support this kind of customizability and extensibility, three essential concepts are defined by the J2ME architecture:

- Configuration. A J2ME configuration defines a minimum platform for a "horizontal" category or grouping of devices, each with similar requirements on total memory budget and processing power. A configuration defines the Java language and virtual machine features and minimum class libraries that a device manufacturer or a content provider can expect to be available on all devices of the same category.
- Profile. A J2ME profile is layered on top of (and thus extends) a configuration. A profile addresses the

[\[Team LiB \]](#)

[!\[\]\(ce899d4e578b5a040d437315360e0be8_img.jpg\) PREVIOUS](#) [!\[\]\(4979f6e1a58c71b36c638558b0d6babd_img.jpg\) NEXT](#) 

2.4 Evolution of the J2ME Platform

After the successful completion of the CLDC and MIDP standardization efforts, a number of additional J2ME standardization efforts have been launched to complement the functionality provided by the core standards. Most of these additional efforts define optional packages that can be deployed on top of MIDP. These efforts are summarized in [Section 2.4.2](#), "Optional Packages for the Wireless Market" below. In addition, there are a number of more fundamental, core standardization efforts that are summarized in [Section 2.4.1](#), "Core J2ME Standardization Efforts."

2.4.1 Core J2ME Standardization Efforts

There are a number of standardization activities that have an important role in defining the overall J2ME architecture. These efforts can be viewed as "umbrella" activities that specify the ground rules for a number of standardization activities or provide additional instructions, recommendations, and clarifications for binding together the existing J2ME standards:

- JSR 68: J2ME Platform Specification
- JSR 185: Java Technology for Wireless Industry

JSR 68: J2ME Platform Specification

This specification defines the "ground rules" for the J2ME platform architecture and J2ME standardization activities. It formalizes the fundamental concepts behind J2ME, such as the notions of a configuration and profile, and defines how new J2ME APIs can be formed by subsetting existing APIs from the Java 2 Platform, Standard Edition (J2SE).

JSR 185: Java Technology for the Wireless Industry (JTWI)

This specification defines how various technologies associated with MIDP work together to form a complete handset solution for the wireless services industry. The specification provides an exposition of the overall architecture of the wireless client software stack, including a description of the following aspects:

- Which optional packages fit with which profiles?
- How does an end-to-end solution for interoperable Java applications work?
- How does the migration of applications occur, and to which profiles, as the devices become more capable?

A key goal of the JT WI Specification is to minimize the fragmentation of the Java APIs in the mobile handset market by creating a community that coordinates the API evolution and deployment as the industry continues to expand the

Chapter 3. Goals, Requirements, and Scope

The CLDC and MIDP standards intend to bring the benefits of Java technology to resource-constrained wireless devices with limited Internet connectivity. This chapter reviews the goals, requirements, and scope of these standards.

[\[Team LiB \]](#)

[!\[\]\(9aecbf7394bad11edef7e00552fc2da9_img.jpg\) PREVIOUS](#) [!\[\]\(c6cbc04a751350992fc5e4f23a28d34f_img.jpg\) NEXT](#) 

3.1 High-Level Goals

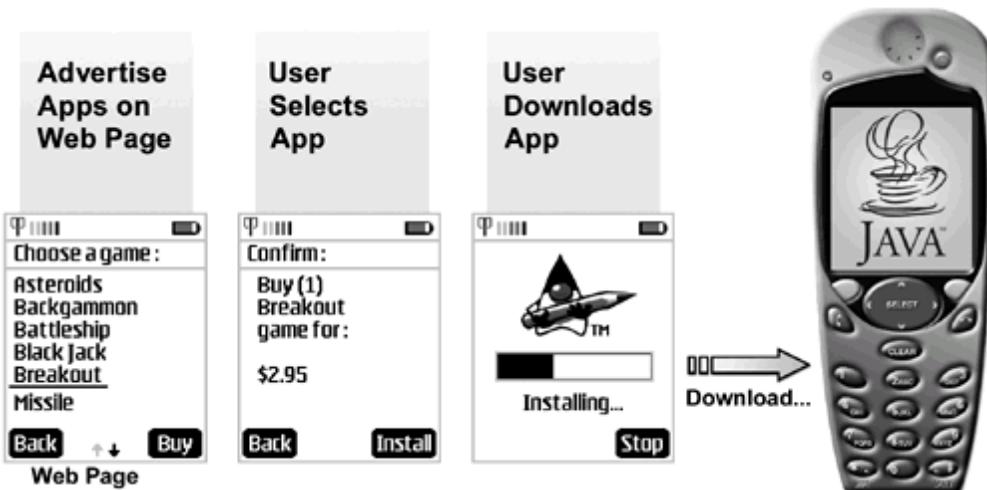
The general goal of the J2ME standardization efforts discussed in this book is to define a highly portable, secure, small-footprint application development environment for resource-constrained, connected devices. We emphasize the term application development environment. The CLDC and MIDP standards are not intended to replace existing system software stacks or to serve as a complete operating system for small devices. Rather, the goal of these efforts is to define an environment that can be added flexibly on top of an existing system software stack to support third-party application development and secure, dynamic downloading of applications.

The CLDC and MIDP standardization efforts have slightly different but complementary goals. Connected, Limited Device Configuration is intended to serve as a generic, "lowest common denominator" platform that targets all kinds of small, connected devices?ndependent of any specific device category. Mobile Information Device Profile builds on top of CLDC and focuses on a specific category of devices: wireless, mobile, two-way communication devices such as cellular telephones and two-way pagers.

3.1.1 Dynamic Delivery of Java Applications and Content

One of the greatest benefits of Java technology in the small device space is the dynamic, secure delivery of interactive services and applications over different kinds of networks. Unlike in the past, when small devices such as cell phones and pagers came from the manufacturer with a hard-coded feature set, device manufacturers are increasingly looking for solutions that allow them to build extensible, customizable devices that support rich, dynamic, interactive content from third-party content providers and developers. With the recent introduction of Internet-enabled cell phones, communicators, and pagers, this transition is already under way. Several wireless device manufacturers are already offering cell phones that allow users to download new applications such as interactive games, screen savers, banking and ticketing applications, wireless collaboration tools, and so on ([Figure 3.1](#)).

Figure 3.1. Downloading customized services



Note that such customizability is not necessarily limited to communication devices such as cell phones or two-way pagers. For instance, it is quite possible to envision automobile engines that obtain new service programs and updates as they become available, washing machines that download new washing programs dynamically, electronic toys that automatically download updated game programs, and so on. The range of possible applications is virtually endless.

[\[Team LiB \]](#)

[!\[\]\(f02dd4fbd63775cd68de9741c4834ec6_img.jpg\) PREVIOUS](#) [!\[\]\(6c1be56a0b1f4970061499be2c64e55e_img.jpg\) NEXT](#) 

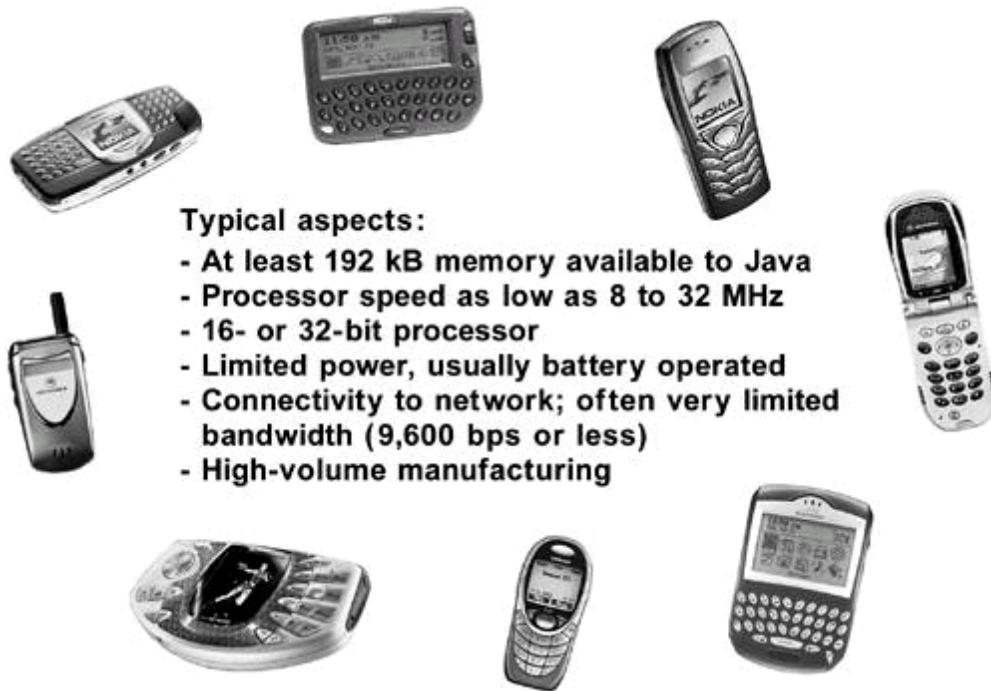
3.2 Target Devices

Potential target devices for the Mobile Information Device Profile (MIDP) include two-way communication devices such as cellular telephones, two-way pagers, and wireless personal organizers. The Connected, Limited Device Configuration (CLDC) can additionally be used to support other kinds of devices such as point-of-sale terminals, barcode scanners, inventory control devices, audio/video equipment, home appliances, and device controllers (for routers, sensors, vending machines, engines, and so forth). In general, CLDC is intended to be applicable to any resource-constrained device that might benefit from a portable, low-cost application development environment that supports secure, dynamic downloading of third-party applications.[\[2\]](#)

[2] Note that CLDC defines only general-purpose APIs, so additional APIs and/or J2ME profiles are typically necessary to address the specific needs of each device category.

The characteristics of the CLDC and MIDP target devices are summarized in [Figure 3.2](#). The technical hardware and software requirements of CLDC and MIDP target devices are defined in detail later in this chapter.

Figure 3.2. CLDC and MIDP target devices



The user interaction models of CLDC and MIDP target devices vary significantly. Consider, for example, how a user interacts with today's crop of mobile information devices.

The most common example is that of "one-handed" operation. Devices of this type are typical cellular phones with small LCD displays and a standard ITU-T telephone keypad (containing at least the keys 0?, *, and #). Users interact with this type of device with one hand, usually with their thumb only. With devices of this type, smaller is better, so these devices typically have a very limited display.

Other mobile information devices are operated in "two-handed mode." These devices have a small, conventional QWERTY keyboard. Data entry for these devices is considerably easier than for a cell phone, so they are used

[\[Team LiB \]](#)

[!\[\]\(951b479d3558e926bd35a732de0b68b5_img.jpg\) PREVIOUS](#) [!\[\]\(13c0d1e88e4cd9ebfe0232f854fac8f4_img.jpg\) NEXT](#) 

3.3 General Notes on Consumer Devices and Embedded Systems

Over the past thirty years, the performance of computer hardware has doubled approximately every 12 to 24 months. This has led to unprecedented miniaturization and widespread usage of computer technology. The price of the microprocessor in today's personal computers is only a few cents per megahertz, and the price is still decreasing rapidly. Today's small wireless devices such as cell phones and two-way pagers have microprocessors that are similar to those that were commonly used in high-end desktop computers less than fifteen years ago.

Ultimately, computing power and storage capacity will become extremely cheap. Inexpensive devices, home appliances, and consumer products such as microwave ovens, shopping carts, wallets, and toys will include powerful processors and megabytes of storage memory.

Thanks to Moore's Law, software developers for desktop computers and servers haven't had to worry very much about processing power or efficiency of their software. Many performance problems have been resolved simply by waiting for a more powerful computer to become available.

However, the consumer device space and embedded software development are different. Even though the processing power of consumer devices and embedded systems is increasing, the majority of these systems are still very resource-constrained and limited in processing power. And, unlike in the desktop and enterprise computing area, the situation is not likely to change dramatically in the next few years. There are several reasons for this:

-
- Moore's Law does not apply to the battery. Most consumer devices are battery-operated. Even though battery technology has advanced substantially in the past years, the advances in battery technology are not as dramatic and predictable as in microprocessor development. Low power consumption is very important in wireless devices, because 75 to 85 percent of the battery power typically needs to be reserved for the radio transmitter and RF signal processing. This places severe power constraints on the rest of the system.
-
- High-volume production. Typically, consumer devices such as cellular phones are manufactured in extremely large quantities (millions or even tens of millions of units). They are sold to price-conscious consumers at prices that are very low, often subsidized. To improve their profit margins, device manufacturers want to keep the per-unit costs of the devices as low as possible. Additional processing power or precious dynamic memory will not be added unless the consumers are willing to pay for the extra capabilities.
-
- Specialized nature of devices. Consumer devices are typically highly specialized. Each device is usually highly customized for its intended usage: cell phones for voice communication, pagers for alphanumeric message retrieval, digital cameras for photography, and so on. Again, to keep the price of the devices reasonable, device manufacturers are not willing to add general-purpose features and capabilities that would raise the cost of the device, unless the features are well-justified from the viewpoint of the target market and the consumer.

In general, a fundamental difference between desktop/enterprise computing and consumer devices is that in the consumer device space one solution does not fit all. Unlike personal computers, which can run tens of thousands of different kinds of applications, and are not customized particularly well for any single application, consumer devices

[\[Team LiB \]](#)

[!\[\]\(49395e7014fb3f8e802dc13ec735e3cb_img.jpg\) PREVIOUS](#) [!\[\]\(28c33a4e8f2c4eeae9e3705954383b78_img.jpg\) NEXT](#) 

3.4 Requirements

This section introduces the technical requirements and constraints that were followed in defining the CLDC and MIDP standards. We start with the hardware and software requirements, and then discuss the specific constraints that Java 2 Micro Edition imposes on its configurations and profiles.

It should be noted that the requirements of CLDC and MIDP are different. CLDC is not targeted to any specific device category, and therefore its requirements are broader than those of MIDP. MIDP is focused specifically on two-way wireless communication devices such as cell phones, and the requirements of MIDP are specific to its target market.

3.4.1 Hardware Requirements of CLDC

CLDC is intended to run on a wide variety of small devices. The underlying hardware capabilities of these devices vary considerably, and therefore the CLDC Specification does not impose any specific hardware requirements other than memory requirements. Even for memory limits, the CLDC Specification defines minimum limits only. The actual CLDC target devices may have significantly more memory than the minimum.

The CLDC Specification assumes that the minimum total memory budget available for the Java Virtual Machine, configuration libraries, profile libraries, and the applications is at least 192 kilobytes. More specifically, it is assumed that:

- At least 160 kilobytes of non-volatile[\[3\]](#) memory is available for the virtual machine and CLDC libraries.

[3] The term non-volatile is used to indicate that the memory is expected to retain its contents between the user turning the device "on" or "off." For the purposes of the CLDC Specification and MIDP Specification, it is assumed that non-volatile memory is usually accessed in read mode and that special setup might be required to write to it. Examples of non-volatile memory include ROM, flash, and battery-packed SDRAM. The CLDC Specification or MIDP Specification do not define which memory technology a device must have, nor do they define the behavior of such memory in a power-loss scenario.
- At least 32 kilobytes of volatile[\[4\]](#) memory is available for the virtual machine runtime (for example, the object heap).

[4] The term volatile is used to indicate that the memory is not expected to retain its contents between the user turning the device "on" or "off." For the purposes of the CLDC Specification and MIDP Specification, it is assumed that volatile memory can be read from and written to directly. The most common type of volatile memory is DRAM.

The ratio of volatile to non-volatile memory in the total memory budget can vary considerably depending on the target device and the role of the Java platform in the device. If the Java platform is used strictly for running system applications that are built in a device, then applications can be prelinked and preloaded, and a very limited amount of volatile memory is needed. If the Java platform is used for running dynamically downloaded content, then devices will need a higher ratio of volatile memory.

[\[Team LiB \]](#)

[!\[\]\(4c6e36f99fe374bedd022e132a02cf78_img.jpg\) PREVIOUS](#) [!\[\]\(121e64f25ddad2096c597b2caf1fbd73_img.jpg\) NEXT](#) 

3.5 Scope of the CLDC and MIDP Standards

3.5.1 Scope of CLDC

Based on the decisions of the JSR 30 and JSR 139 expert groups, the CLDC Specification addresses the following areas:

- Java language and virtual machine features
- core libraries (java.lang.* , java.io.* , java.util.*)
- input/output
- networking
- security
- internationalization

The CLDC Specification intentionally does not address the following functionality areas and features:

- application life-cycle management (installation, launching, deletion)
- user interface functionality
- event handling
- high-level application model (interaction between the user and the application)

These features can be addressed by profiles implemented on top of the Connected, Limited Device Configuration. In general, the CLDC expert group intentionally kept small the number of areas addressed by the CLDC Specification in order not to exceed the strict memory limitations or to exclude any particular device category.

3.5.2 Scope of MIDP

Chapter 4. Connected Limited Device Configuration

This chapter introduces the Connected, Limited Device Configuration (CLDC), one of the core building blocks of the Java 2 Platform, Micro Edition (J2ME). The goal of the Connected, Limited Device Configuration is to provide a standardized, highly portable, minimum-footprint Java application development platform for resource-constrained, connected devices. The CLDC target devices are characterized generally as follows:

- at least 192 kilobytes of total memory budget available for the Java platform,
- a 16-bit or 32-bit processor,
- low power consumption, often operating with battery power, and
- connectivity to some kind of network, often with a wireless, intermittent connection and with limited (often 9600 bps or less) bandwidth.

CLDC is core technology that can be used as the basis for one or more J2ME profiles. Cell phones, two-way pagers, personal digital assistants (PDAs), organizers, home appliances, low-end TV set-top boxes, and point-of-sale terminals are some, but not all, of the devices that might be supported by the Connected, Limited Device Configuration.

The material in this chapter comes from the CLDC Specification document that is available from the Java Community Process (JCP) web site.^[1] The CLDC Specification defines the minimum required complement of Java technology components and libraries for small connected devices. Java language and virtual machine features, core libraries, input/output, networking, and security are the primary topics addressed by the CLDC Specification.

[1] <http://jcp.org/en/jsr/detail?id=139>

[\[Team LiB \]](#)

[!\[\]\(b66725e52a8c81be973cd518958addb9_img.jpg\) PREVIOUS](#) [!\[\]\(f340bf72fef55de26e4bcc102533f45b_img.jpg\) NEXT](#) 

4.1 CLDC Expert Groups

4.1.1 CLDC 1.0 Expert Group

The CLDC Specification version 1.0, completed in May 2000, was the result of the work of a Java Community Process (JCP) expert group JSR 30 consisting of a large number of industrial partners. The following 18 companies (in alphabetical order) were members of the CLDC 1.0 expert group:

America Online

Oracle

Bull

Palm Computing

Ericsson

Research In Motion (RIM)

Fujitsu

Samsung

Matsushita

Sharp

Mitsubishi

Siemens

Motorola

Sony

Nokia

Sun Microsystems

NTT DoCoMo

Symbian

4.1.2 CLDC 1.1 Expert Group

The CLDC Specification version 1.1 was produced by the Java Community Process (JCP) expert group JSR 139 consisting of the following full members:

aJile Systems

Openwave Systems

Aplix Corporation

Oracle

France Telecom

Panasonic

Fujitsu

Research In Motion

[\[Team LiB \]](#)

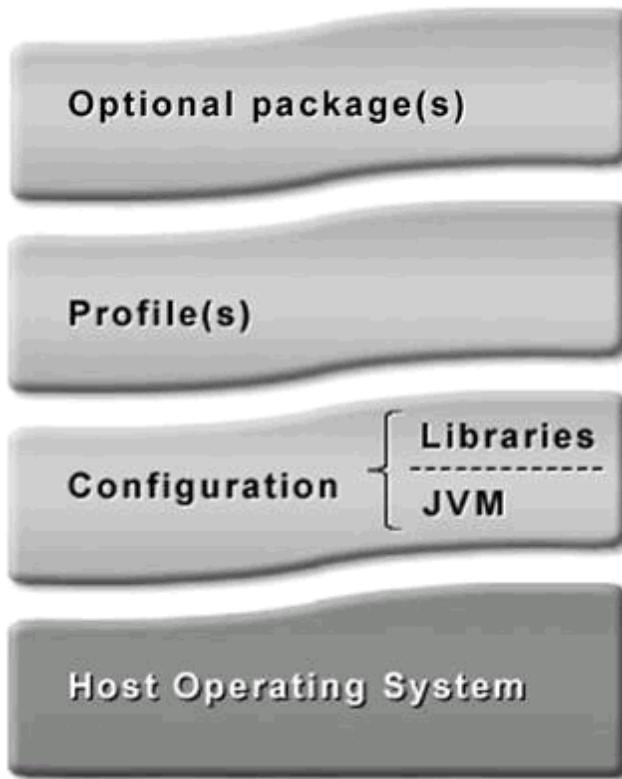
[!\[\]\(6450c25206cb41661f6831b5e94bfe70_img.jpg\) PREVIOUS](#) [!\[\]\(4d69078bb04248f1660cf89a1cb6647a_img.jpg\) NEXT](#) 

4.2 CLDC Architecture, Application Model, and Security

4.2.1 Architectural Overview

The high-level architecture of a typical CLDC device is illustrated in [Figure 4.1](#). At the heart of a CLDC implementation is the Java Virtual Machine, which, apart from specific differences defined later in this chapter, is compliant with the Java Virtual Machine Specification and Java Language Specification. The virtual machine typically runs on top of a host operating system that provides the necessary capabilities to manage the underlying hardware. As explained in [Section 3.4.3](#), "Software Requirements of CLDC," the CLDC Specification makes minimal assumptions about the capabilities of the host operating system.

Figure 4.1. Architecture overview of a CLDC target device



On top of the virtual machine are the Java libraries. These libraries are divided broadly into two categories:

1. those defined by the Connected, Limited Device Configuration (CLDC Libraries), and
2. those defined by profiles (such as MIDP) and optional packages.

Libraries defined by the Connected, Limited Device Configuration are discussed in [Chapter 5](#). Libraries defined by profiles are outside the scope of the CLDC Specification. The libraries supported by the Mobile Information Device Profile are discussed in [Chapters 8](#) through [20](#).

4.2.2 The Concept of a Java Application

[\[Team LiB \]](#)

[!\[\]\(c2628367a5f70c7f18c54368794fddcf_img.jpg\) PREVIOUS](#) [!\[\]\(5f61747860020efdbcd43a1f7139d1a4_img.jpg\) NEXT](#) 

4.3 Java Language Specification Compatibility

The general goal for a virtual machine conforming to CLDC is to be as compliant with the Java Language Specification as is feasible within the strict memory limits of CLDC target devices. This section summarizes the differences between a virtual machine conforming to CLDC and the Java Virtual Machine of Java 2 Standard Edition (J2SE). Except for the differences indicated herein, a virtual machine conforming to CLDC is compatible with The Java Language Specification, Second Edition, by James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha, Addison-Wesley, 2000, ISBN 0-201-31008-2.

Note

For the remainder of this book, the Java Language Specification is referred to as JLS. Sections within the Java Language Specification are referred to using the ?symbol. For example, (JLS [?2.4](#)).

4.3.1 No Finalization of Class Instances

CLDC libraries do not include the method `java.lang.Object.finalize()`. Therefore, a virtual machine conforming to CLDC does not support finalization of class instances (JLS [?2.6](#)). No application built to conform to the Connected, Limited Device Configuration can require that finalization be available.

4.3.2 Error-Handling Limitations

A virtual machine conforming to CLDC generally supports exception handling as defined in JLS [Chapter 11](#), with the exception that asynchronous exceptions (JLS [?1.3.2](#)) are not supported.

In contrast, the set of error classes included in CLDC libraries is limited, and consequently the error-handling capabilities of CLDC are considerably more limited. This is because of two reasons:

1.

In embedded systems, recovery from error conditions is usually highly device-specific. Application programmers cannot be expected to know about device-specific error-handling mechanisms and conventions.

2.

As specified in JLS [?1.5](#), class `java.lang.Error` and its subclasses are exceptions from which programs are not ordinarily expected to recover. Implementing the error-handling capabilities fully according to the Java Language Specification is rather expensive, and mandating the presence and handling of all the error classes would impose an overhead on the virtual machine implementation.

A virtual machine conforming to CLDC provides a limited set of error classes defined in [Section 5.2.7](#), "Exception and Error Classes." When encountering any other error, the implementation behaves as follows:



[\[Team LiB \]](#)

[!\[\]\(e8f0802ad8a7ddf23a327c70606505d1_img.jpg\) PREVIOUS](#) [!\[\]\(be431e403d6dcbf6a6bd501c4bd3fe09_img.jpg\) NEXT](#) 

4.4 Java Virtual Machine Specification Compatibility

The general goal for a virtual machine conforming to CLDC is to be as compliant with the Java Virtual Machine Specification as is possible within strict memory constraints of CLDC target devices. This section summarizes the differences between a virtual machine conforming to CLDC and the Java Virtual Machine of Java 2 Standard Edition (J2SE). Except for the differences indicated herein, a virtual machine conforming to CLDC is compatible with the Java Virtual Machine as specified in the The Java Virtual Machine Specification, Second Edition by Tim Lindholm and Frank Yellin. Addison-Wesley, 1999, ISBN 0-201-43294-3.

Note

For the remainder of this book, the Java Virtual Machine Specification is referred to as JVMS. Sections within the Java Virtual Machine Specification are referred to using the ?symbol. For example, (JVMS ?.4.3).

4.4.1 Features Eliminated from the Virtual Machine

A number of features have been eliminated from a virtual machine conforming to CLDC because the Java libraries included in CLDC are substantially more limited than libraries in Java 2 Standard Edition, and/or the presence of those features would have posed security problems in the absence of the full J2SE security model. The eliminated features include:

- user-defined class loaders (JVMS [?.3.2](#)),
- thread groups and daemon threads (JVMS ?.19, ?.12),
- finalization of class instances (JVMS ?.17.7), and
- asynchronous exceptions (JVMS ?.16.1).

In addition, a virtual machine conforming to CLDC has a significantly more limited set of error classes than a full J2SE virtual machine.

Applications written to conform to the Connected, Limited Device Configuration shall not rely on any of the features above. Each of the features in this list is discussed in more detail next.

User-defined class loaders

A virtual machine conforming to CLDC does not support user-defined, Java-level class loaders (JVMS [?.3](#), ?.17.2).

A virtual machine conforming to CLDC has a built-in "bootstrap" class loader that cannot be overridden, replaced, or

4.5 New for CLDC 1.1



The CLDC 1.1 (JSR 139) expert group members were generally satisfied with the CLDC Specification version 1.0 and did not see any need for radical changes in the new specification. Therefore, CLDC Specification version 1.1 is primarily an incremental release that is intended to be fully backwards compatible with CLDC Specification version 1.0. Some important new functionality, such as floating point support, has been added.

The list below summarizes the main differences between CLDC Specification versions 1.1 (JSR 139) and 1.0 (JSR 30). Those differences that are specific to Java libraries supported by CLDC are summarized in [Section 5.4, "New for CLDC 1.1."](#)

- Floating point support has been added.
 - All floating point bytecodes are supported by CLDC 1.1.
 - Classes Float and Double have been added, and various methods have been added to the other library classes to handle floating point values. (Refer to [Section 5.4, "New for CLDC 1.1,"](#) and the CLDC Almanacs at the end of this book for a detailed summary of the changes.)
- Weak reference support (small subset of the J2SE weak reference classes) has been added.
- Error-handling requirements have been clarified, and one new error class, NoClassDefFoundError, has been added.
- In CLDC 1.1, Thread objects have names like threads in J2SE do. The method Thread.getName() has been introduced, and the Thread class has a few new constructors that have been inherited from J2SE.
- In CLDC 1.1, Threads can be interrupted using the Thread.interrupt method.
- Minimum total memory budget for CLDC has been increased from 160 to 192 kilobytes, mainly because of the added floating point functionality.
- Much more detailed verifier specification ("CLDC Byte Code Typechecker Specification") is provided as an appendix of the CLDC Specification version 1.1.

Chapter 5. CLDC Libraries

Java 2 Platform, Standard Edition (J2SE) and Java 2 Platform, Enterprise Edition (J2EE) provide a very rich set of Java class libraries for the development of enterprise applications for desktop computers and servers. Unfortunately, these libraries require tens of megabytes of memory to run and are therefore unsuitable for small devices with limited resources. In this chapter we introduce the class libraries supported by the Connected, Limited Device Configuration.

5.1 Background and Goals

A general goal for designing the libraries for the Connected, Limited Device Configuration was to provide a minimum useful set of libraries for practical application development and profile definition for a variety of small devices. As explained earlier in [Section 3.1](#), "High-Level Goals," CLDC is a "lowest common denominator" standard that includes only minimal Java platform features and APIs for a wide range of consumer devices. Given the strict memory constraints and differing features of today's small devices, it is virtually impossible to come up with a set of libraries that would be ideal for all devices. No matter where the bar for feature inclusion is set, the bar is inevitably going to be too low for some devices and users, and too high for many others.

To ensure upward compatibility with the larger editions of the Java 2 Platform, the majority of the libraries included in CLDC are a subset of Java 2 Standard Edition (J2SE) and Java 2 Enterprise Edition (J2EE). While upward compatibility is a very desirable goal, J2SE and J2EE libraries have strong internal dependencies that make subsetting them difficult in important areas such as security, input/output, user interface definition, networking, and storage. These dependencies are a natural consequence of design evolution and reuse that has taken place during the development of the libraries over time. Unfortunately, these dependencies make it difficult to take just one part of the libraries without including several others. For this reason, some of them have been redesigned, especially in the area of networking.

The libraries defined by the CLDC Specification can be divided broadly into two categories:

1. those classes that are a subset of standard J2SE libraries, and
2. those classes that are specific to CLDC (but that can be mapped onto J2SE).

Classes belonging to the former category are located in packages `java.lang`, `java.util`, and `java.io`. These classes have been derived from Java 2 Standard Edition version 1.3.1. A list of these classes is presented in [Section 5.2](#), "Classes Derived from Java 2 Standard Edition."

Classes belonging to the latter category are located in package `javax.microedition`. These classes are discussed in [Section 5.3](#), "CLDC-Specific Classes."

[\[Team LiB \]](#)

[!\[\]\(4045b1beb079b18f7753b014534e8e7d_img.jpg\) PREVIOUS](#) [!\[\]\(fa08e6f57d1b84da923282eb95032d03_img.jpg\) NEXT](#) 

5.2 Classes Derived from Java 2 Standard Edition

CLDC provides a number of classes that have been derived from Java 2 Standard Edition (J2SE). The rules for J2ME configurations require that each class with the same name and package name as a J2SE class must be identical to or a subset of the corresponding J2SE class. The semantics of the classes and their methods included in the subset may not be changed. The classes may not add any public or protected methods or fields that are not available in the corresponding J2SE classes.

5.2.1 System Classes

J2SE class libraries include several classes that are intimately coupled with the Java Virtual Machine. Similarly, several commonly used Java tools assume the presence of certain classes in the system. For instance, the standard Java compiler (javac) generates code that requires certain methods of classes String and StringBuffer to be available.

The system classes included in the CLDC Specification are listed below. Each of these classes is a subset of the corresponding class in J2SE. Note that in CLDC 1.1, many of these classes contain additional methods that are not present in CLDC 1.0. Refer to CLDC library documentation in the end of this book for more details on each class, as well as for the differences between CLDC 1.0 and 1.1.

```
java.lang.Object  
java.lang.Class  
java.lang.Runtime  
java.lang.System  
java.lang.Thread  
java.lang.Runnable (interface)  
java.lang.String  
java.lang.StringBuffer  
java.lang.Throwable
```

5.2.2 Data Type Classes



The following basic data type classes from package java.lang are supported. Each of these classes is a subset of the corresponding class in J2SE. Classes Float and Double are new in CLDC 1.1.

```
java.lang.Boolean  
java.lang.Byte  
java.lang.Short  
java.lang.Integer  
java.lang.Long  
java.lang.Float  
java.lang.Double  
java.lang.Character
```

5.2.3 Collection Classes

The following collection classes from package java.util are supported.

[\[Team LiB \]](#)

[!\[\]\(7b2eb436f3582968c4312da7eb15ea56_img.jpg\) PREVIOUS](#) [!\[\]\(69a41d5e51cc33a48b478e70474f7a7c_img.jpg\) NEXT](#) 

5.3 CLDC-Specific Classes

This section contains a description of the Generic Connection framework for supporting input/output and networking in a generalized, extensible fashion. The Generic Connection framework provides a coherent way to access and organize data in a resource-constrained environment.

5.3.1 Background and Motivation

The class libraries included in Java 2 Standard Edition and Java 2 Enterprise Edition provide a rich set of functionality for handling input and output access to storage and networking systems. For instance, in JDK 1.3.1, the package `java.io` contained 59 regular classes and interfaces, and 16 exception classes. The package `java.net` of JDK 1.3.1 consisted of 31 regular classes and interfaces, and 8 exception classes. It is difficult to make all this functionality fit in a small device with only a few hundred kilobytes of total memory budget. Furthermore, a significant part of the standard I/O and networking functionality is not directly applicable to today's small devices, which often do not have TCP/IP support, or which commonly need to support specific types of connections such as Infrared or Bluetooth.

In general, the requirements for the networking and storage libraries vary significantly from one resource-constrained device to another. For instance, device manufacturers who are dealing with packet-switched networks typically want datagram-based communication mechanisms, while those dealing with circuit-switched networks require stream-based connections. Due to strict memory limitations, manufacturers supporting certain kinds of networking capabilities generally do not want to support any other mechanisms. All this makes the design of the networking facilities for CLDC very challenging, especially since J2ME configurations are not allowed to define optional features. Also, the presence of multiple networking mechanisms and protocols is potentially very confusing to the application programmer, especially if the programmer has to deal with low-level protocol issues.

5.3.2 The Generic Connection Framework

The challenges presented above have led to the generalization of the J2SE networking and I/O classes. The high-level goal for this generalized design is to be a precise functional subset of J2SE classes, which can easily map to common low-level hardware or to any J2SE implementation, but with better extensibility, flexibility, and coherence in supporting new devices and protocols.

The general idea is illustrated below. Instead of using a collection of different kinds of abstractions for different forms of communication, a set of related abstractions are used at the application programming level.

General form

```
Connector.open( "<protocol>:<address>;<parameters>" );
```

Note

These examples are provided for illustration only. CLDC itself does not define any protocol implementations (see [Section 5.3.3](#), "No Network Protocol Implementations Defined in CLDC"). It is not expected that a particular J2ME profile would provide support for all these kinds of connections. J2ME profiles may also support protocols not described here.

[\[Team LiB \]](#)

[!\[\]\(802a39c9ea561f713321b083fb86799b_img.jpg\) PREVIOUS](#) [!\[\]\(44a964ba92caebe4ca904a6eebd4b2ea_img.jpg\) NEXT](#) 

5.4 New for CLDC 1.1



The following list summarizes the Java library changes between CLDC Specification versions 1.0 and 1.1. For details, refer to "[CLDC Almanac](#)" on page 359.

- - Classes `Float` and `Double` have been added.
 - The following methods have been added to other library classes to handle floating point values:
 - `java.lang.Integer.doubleValue()`
 - `java.lang.Integer.floatValue()`
 - `java.lang.Long.doubleValue()`
 - `java.lang.Long.floatValue()`
 - `java.lang.Math.abs(double a)`
 - `java.lang.Math.abs(float a)`
 - `java.lang.Math.max(double a, double b)`
 - `java.lang.Math.max(float a, float b)`
 - `java.lang.Math.min(double a, double b)`
 - `java.lang.Math.min(float a, float b)`
 - `java.lang.Math.ceil(double a)`
 - `java.lang.Math.floor(double a)`

[\[Team LiB \]](#)

[!\[\]\(713932b028198c05e50737e04cd4e50a_img.jpg\) PREVIOUS](#) [!\[\]\(9b2a3f0578f0e269d8c0db36b6ef29ce_img.jpg\) NEXT](#) 

Chapter 6. Mobile Information Device Profile

The Mobile Information Device Profile (MIDP) for the Java 2 Platform, Micro Edition (J2ME) is an architecture and a set of Java libraries that create an open application environment for small, resource-constrained mobile information devices, or MIDs. Typical examples of MIDP target devices include cellular phones, two-way pagers, and wireless personal organizers. As summarized in [Chapter 3](#), MIDP 2.0 devices typically fulfill the following minimum requirements:

- - Memory:
 - 256 kilobytes of non-volatile memory for the MIDP components
 - 8 kilobytes of non-volatile memory for application-created persistent data
 - 128 kilobytes of volatile memory for the virtual machine runtime (for example, the object heap)
- - Display:
 - Screen-size: 96x54
 - Display depth: 1-bit
 - Pixel shape (aspect ratio): approximately 1:1
- - Input:
 - One or more of the following user-input mechanisms:
 - ⊕ "one-handed keypad"
 - ⊕ "two-handed keyboard"
 - ⊕ touch screen
- - Networking:

[\[Team LiB \]](#)

[!\[\]\(015737cb75488e290cfd7e881045e5c8_img.jpg\) PREVIOUS](#) [!\[\]\(5293baed919d09450e39359e49dcbc99_img.jpg\) NEXT](#) 

6.1 MIDP Expert Groups

6.1.1 MIDP 1.0 Expert Group

The MIDP Specification version 1.0 was produced by the Mobile Information Device Profile Expert Group (MIDPEG) as part of the Java Community Process (JCP) standardization effort JSR 37. The following 22 companies, listed in alphabetical order, were members of the MIDP 1.0 standardization effort:

America Online	DDI	Ericsson
Espial Group	Fujitsu	Hitachi
J-Phone	Matsushita	Mitsubishi
Motorola	NEC	Nokia
NTT DoCoMo	Palm Computing	RIM
Samsung	Sharp	Siemens
Sony	Sun Microsystems	Symbian
Telcordia Technologies		

6.1.2 MIDP 2.0 Expert Group

The MIDP Specification version 2.0 was produced as part of the Java Community Process (JCP) standardization effort JSR 118. More than 120 people (the largest expert group ever created in the Java Community Process) were members of the MIDP 2.0 expert group.

The expert group included people from the following 49 companies (alphabetical):

4thpass Inc.	AGEA Corporation	Alcatel
Aplix Corporation	AromaSoft Corp	Baltimore Technologies
CELLon France	Distributed Systems Technology Centre	Elatel PLC

6.2 Areas Covered by the MIDP Specification

The Mobile Information Device Profile Specification extends the functionality defined by the Connected, Limited Device Configuration (CLDC) Specification. The MIDP Specification defines a set of APIs that add a minimum set of capabilities that are common to various kinds of mobile information devices. The MIDP Specification version 2.0 extends the capabilities of the original MIDP Specification version 1.0 while retaining full backward compatibility. As already summarized earlier in [Section 3.5.2](#), "Scope of MIDP," the specific areas covered by the MIDP Specification version 2.0 include:

- application model (that is, defining the semantics of a MIDP application and how the application is controlled),
- user interface support (Limited Capability Device User Interface, LCDUI),
- networking support (based on the HTTP protocol and the Generic Connection framework introduced by CLDC),
- persistent storage support (Record Management System, RMS),
- sounds,
- 2D games,
- end-to-end security through HTTPS and secure sockets,
- MIDlet signing model for added security,
- application delivery and installation, and
- miscellaneous classes such as timers and exceptions.

In addition to these areas, the MIDP Specification defines an extension of the CLDC application model that allows for the execution and communication of applications called MIDlets. A MIDlet is the basic unit of execution in MIDP. One or more MIDlets packaged together are called a MIDlet suite. The MIDP application model and each of the areas mentioned above are discussed in detail in the subsequent chapters of this book. The concept of a MIDlet suite is discussed in detail in [Section 19.1](#), "MIDlet Suites."

Chapter 7. MIDP Application Model

Due to the strict memory constraints and the requirement to support application interaction and data sharing within related applications, the Mobile Information Device Profile does not support the familiar Applet programming model introduced by the Java 2 Platform, Standard Edition (J2SE). Rather, MIDP introduces a new application model that is better suited for the specific needs of small, resource-constrained devices. This application model is integrated closely with the MIDP user interface libraries (see [Chapters 8–11](#)), and it replaces the simple, command line-oriented application model supported by the Connected, Limited Device Configuration.

[\[Team LiB \]](#)

[!\[\]\(ae4572f4ce05dab8899601a7082846fd_img.jpg\) PREVIOUS](#) [!\[\]\(3613fd9019eb9c1fb92d33ae63b87a69_img.jpg\) NEXT](#) 

7.1 MIDlets

In MIDP, the basic unit of execution is a MIDlet. A MIDlet is a class that extends the class javax.microedition.midlet.MIDlet and implements a few methods?ncluding startApp, pauseApp, and destroyApp?o define the key behavior of the application.

As an example of programming with the MIDP application model, consider the following program that implements one of the simplest MIDlets possible: the canonical "Hello World" application.

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

public class HelloWorld extends MIDlet implements CommandListener {
    private Command exitCommand;
    private TextBox tb;

    // Constructor of the HelloWorld class
    public HelloWorld() {
        // Create "Exit" Command so the user can exit the MIDlet
        exitCommand = new Command("Exit", Command.EXIT, 1);

        // Create TextBox in which the output will be displayed
        tb = new TextBox("Hello MIDlet", "Hello, World!", 15, 0);

        // Add the Command and set the CommandListener
        tb.addCommand(exitCommand);
        tb.setCommandListener(this);

        // Set our TextBox as the current screen
        Display.getDisplay(this).setCurrent(tb);
    }

    // Called by the system when the MIDlet is started
    // for the first time, and when it is resumed after
    // being paused.
    protected void startApp() {}

    // Called by the system when the MIDlet is paused.
    // In this application, there is no need to do anything
    // when we are paused.
    protected void pauseApp() {}

    // Called when the MIDlet is destroyed. In this case,
    // there is no need to do anything when we are destroyed.
    protected void destroyApp(boolean force) {}

    // This method is called automatically by the system
    // in response to a user activating one of the Commands.
    public void commandAction(Command c, Displayable d) {
        // Upon receiving the "Exit" Command, destroy ourselves
        if (c == exitCommand) {
            destroyApp(false);
            notifyDestroyed();
        }
    }
}
```


7.2 MIDlet Suites

When MIDlets are delivered to devices for execution, they are packaged in a MIDlet suite. MIDlet suites provide a controlled way of downloading multiple MIDlets onto a device as well as support controlled sharing of data and resources between multiple, possibly simultaneously running MIDlets. MIDlet suites will be discussed in detail in [Chapter 19](#), "MIDlet Deployment."

7.3 New for MIDP 2.0

The MIDP application model has not changed since MIDP 1.0. However, the interactions with the user interface and the suggested application actions that are performed in these methods are now more precisely defined. By limiting the complexity of functions performed in these methods, the application can be more responsive to both system requests and user actions.

Chapter 8. MIDP User Interface Libraries

In light of the wide variations in cellular phones and other MIDP target devices, the requirements for user interface support are very challenging. MIDP target devices differ from desktop systems in many ways, especially in how the user interacts with them. The following UI-related requirements were seen as important when the user interface library of the MIDP Specification was designed:

- The devices and applications should be useful to users who are not necessarily experts in using computers.
- The devices and applications should be useful in situations where the user cannot pay full attention to the application. For example, many cellular phones and other wireless devices are commonly operated with one hand while the user is walking, cooking, fishing, and so forth.
- The form factors and user interface concepts of MIDP target devices vary considerably between devices, especially compared to desktop systems. For example, display sizes are much smaller, and input devices are much more limited or at least different. Most mobile devices have only numeric keypads, and few have pointing devices such as pen-operated touch screens.
- The Java applications for mobile information devices should have user interfaces that are consistent with the native applications so that the user finds them intuitive to use.

Given these requirements and the capabilities of devices that implement the MIDP (see [Section 3.2](#), "Target Devices"), a dedicated user interface toolkit is provided in MIDP in the `javax.microedition.lcdui` package.

8.1 MIDP UI Compared to Desktop AWT

The differences between mobile devices and desktop computers are evident when comparing the designs of the MIDP lcdui with the Abstract Windowing Toolkit (AWT) of the JavaTM 2 Platform, Standard Edition.

AWT was designed and optimized for desktop computers. The assumptions that are appropriate for desktop computers are not appropriate for mobile devices. For example, when a user interacts with AWT, event objects are created dynamically. These objects are short-lived and exist only until each associated event is processed by the system. The event object then becomes garbage and must be reclaimed by the garbage collector of the Java virtual machine. The limited CPU and memory subsystems of a mobile information device (MID) typically cannot afford the overhead of unnecessary garbage objects. To limit the creation of such objects, lcdui events are simply method calls that pass the event parameters directly as the parameters of the method.

AWT also assumes certain desktop environment specific user interaction models. For instance, the component set of AWT was designed to work with a pointing device such as a mouse or pen. As mentioned earlier, this assumption is valid only for a small subset of MIDs, since most of the MIDP target devices have just a keypad for user input. The MIDP user interface API uses a layer of abstraction to separate applications from concrete input methods used in different devices. For example, a MIDP device with touch input allows the user to use this input method on high-level user interface components but it is not visible to applications. The different concrete input methods are transparent to the application code.

AWT has extensive support for window management, such as overlapping windows, window resizing, and so forth. In contrast, MIDP target devices have small displays that are not large enough for multiple overlapping windows. The lcdui API of MIDP has no window management capabilities. A lcdui user interface is constructed using a set of Displayables. This [displayable model](#) and the class structure of user interface are described in the next section.

[\[Team LiB \]](#)

[!\[\]\(57ed95794e2f3b5e2bfe2abf8f3cdcbd_img.jpg\) PREVIOUS](#) [!\[\]\(b14d18570f2b300364a7bae24e27c80b_img.jpg\) NEXT](#) 

8.2 Structure of the MIDP User Interface API

The following sections provide an overview to the MIDP user interface API (`javax.microedition.lcdui`). The remainder of this chapter and the following two chapters concentrate on the high-level parts of the API. The low-level user interface API features are detailed in [Chapter 11](#), "MIDP Low-Level User Interface Libraries."

The separation of high-level and low-level API is often not very clear-cut. There are many classes that are used both in low-level and high-level APIs. For example, classes `Image` and `Font` are used both in high-level `Screen` components and low-level graphics drawing. Also, in MIDP 2.0 the new `CustomItem` class makes it possible to mix low-level graphics and high-level components within a `Form` object.



MIDP 2.0 also includes a new game-specific API provided in `javax.microedition.lcdui.game` package. The MIDP 2.0 Game API is discussed in [Chapter 12](#).

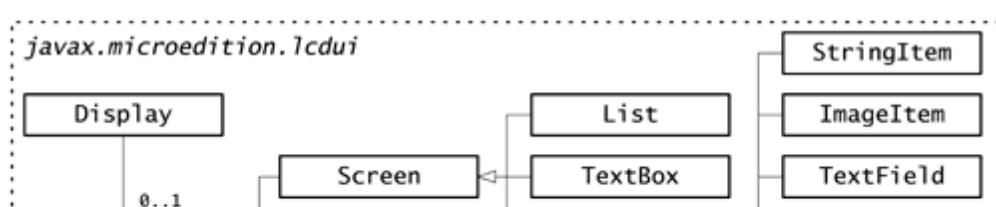
8.2.1 Displayable Model

The central abstraction of the MIDP user interface is class `Displayable`. Class `Displayable` helps organize the user interface into manageable pieces. This results in user interfaces that are easy to use and learn. Each MIDP application has a `Display` on which a single `Displayable` object is shown. The application sets and resets the current `Displayable` object on the `Display` for each step of the task, based on user interactions. The user tasks (interactions with the user interface) are implemented using `Command` objects. The application is notified automatically when a `Command` is selected by the user. As a result of these notifications the application often changes the current `Displayable` to some other `Displayable`. The device software manages the sharing of the physical display between the native applications and the MIDP applications.

The rationale behind the displayable-oriented approach is based on the wide variations in display and keypad configurations found in MIDP devices. Each device provides a consistent look and feel by handling the component layout, painting, scrolling, and focus traversal. If an application needed to be aware of these details, portability would be difficult to achieve, and smooth integration with the look and feel of the device and its native applications would place a heavy burden on application developers.

There is a hierarchy of `Displayable` subclasses called `Screens` in the `lcdui` API. Each `Screen` is a functional user interface element that encapsulates device-specific graphics rendering and user input handling. [Figure 8.1](#) shows the hierarchy of the user interface classes.

Figure 8.1. MIDP user interface class hierarchy (all classes not shown)



[\[Team LiB \]](#)

[!\[\]\(7dcbf2310f408fd495786ab3ad566012_img.jpg\) PREVIOUS](#) [!\[\]\(c235be60feca1ec79ad5a112cd550ce1_img.jpg\) NEXT](#) 

8.3 Display

The Display class is the central controller of the display of a MIDP application. An application gets a Display instance by calling the static method `getDisplay` and by passing the MIDlet instance to the method:

```
Display midletDisplay = Display.getDisplay(myMIDlet);
```

When the Display instance is obtained, the application can change the Displayables presented on the device using the methods of the Display class. There are three methods in class Display that allow the application to set the current Displayable instance presented on the device display. The most commonly used method is:

```
public void setCurrent(Displayable nextDisplayable)
```

The two other methods are provided for more specialized use:

```
public void setCurrent(Alert alert, Displayable nextDisplayable)
public void setCurrentItem(Item item)
```



The first of these specialized methods sets an Alert to be displayed and also provides the Displayable that is presented after the Alert is dismissed. See [Section 9.3](#), "Alert." The second of these specialized methods presents a specific Item attached to a Form instance. The Form that has the Item is set as the current displayable, and it is scrolled so that the Item becomes visible. Focus, if supported by device, moves to this Item.

8.3.1 Current Displayable

Technically, the methods just discussed set only the [current displayable](#) of class Display. During the lifecycle of the application the current Displayable never again is null after it has been initially set at application startup. The current Displayable is made visible by the device software referred to as application management software (AMS) in MIDP Specification (see [Section 19.2](#), "MIDP System Software.") The current Displayable is visible when the application is in the foreground.

There often are, however, situations when the application management software moves the application to the background, and thus the current Displayable is not actually visible. The current Displayable does not change in these cases. Applications are commonly moved to background upon system events such as incoming calls in a phone device or upon certain other system notifications. When the application management software decides that the application can again be brought to the foreground, the current Displayable is simply made visible.

The application can determine whether a Displayable is visible on the display by calling the method `isShown`. In the case of Canvas and CustomItem, the `showNotify` and `hideNotify` methods are called when the component is made visible and is hidden, respectively. Applications need to override `showNotify` and `hideNotify` methods in Canvas or CustomItem subclasses to be able receive these notifications.

[\[Team LiB \]](#)

[!\[\]\(da3dd1d44378ef625513225eb3da25fc_img.jpg\) PREVIOUS](#) [!\[\]\(ecc6f61e478cd037405c3ff4417ef3ab_img.jpg\) NEXT](#) 

8.4 Displayables

Displayable objects are the views in which the application interaction takes place. Application-defined operations in Displayable objects are implemented using Commands. The device implementation provides additional operations based on displayable type, as will be discussed in [Section 8.6.3](#), "Device-Provided Operations."

All Displayable objects have the following properties:

- Zero or more Commands for application specific user operations on the Displayable
- A CommandListener (set by the application with the method `setCommandListener[1]`) that gets notified when the user selects any of the Commands
 - [1] Note that only a single CommandListener can be added because of the unicast listener model is used in the lcdui API.
- A title[2] string (set with the `setTitle` method) that is presented to the user in a standard place, for example, as a header text for the Displayable
 - [2] In MIDP Specification version 2.0, the title and Ticker properties were added to class Displayable, which means that they are also available in class Canvas. In MIDP 1.0, these properties were only available in the Screen classes.
- A Ticker object (set with the `setTicker` method) that provides automatically scrolling text attached to the Displayable

Both the title and Ticker can be null, which means that no title text or Ticker is displayed. Depending on the user interface implementation of the device, the existence of a Ticker or title can consume space from the rest of the displayable content.



[\[Team LiB \]](#)

[!\[\]\(a1f787aff975ee7b4466d883820aa46c_img.jpg\) PREVIOUS](#) [!\[\]\(c71b15e6b819271e51cc0a122b551c34_img.jpg\) NEXT](#) 

8.5 Commands

Abstract Commands are the operations that the applications add to the user interface. Commands can be added to all Displayables, so they are part of the application when using both high-level or low-level API. The most common task for Commands is to provide users with a way to navigate between different Displayables of an application. Normally, if there are no Commands in a Displayable, the user has no way to proceed to other Displayables.

Commands are added to a Displayable using `addCommand` method. There is also `removeCommand` to remove already-added commands. This allows to dynamically change available operations within displayables.

When the user selects a Command from the user interface, the device calls the `CommandListener's commandAction` method and passes the `Command` object and the `Displayable` instance in which the Command was activated.

In the next sections, the idea of Command abstraction is described first. Then the proper use of Commands in applications is summarized.

8.5.1 Command Mapping to Device User Interface

Since the MIDP user interface API is highly abstract, it does not dictate any concrete user interaction technique such as softkeys,^[4] buttons, or menus. An abstract command mechanism allows an application to adjust to the widely varying input mechanisms of MIDP target devices and also allows it to be unaware of device specifics, such as number of keys, key locations, and key bindings. Low-level user interactions such as traversal or scrolling are not visible to the application, which helps the devices implement the high-level components in very different user interface styles. This also leverages application portability between devices.

[4] Softkeys are very common in mobile devices. Softkey is a keypad key positioned near the device display so that there is an changeable label present on the display that indicates the operation associated with the softkey.

MIDP applications define Commands, and the implementation can provide user control over these with softkeys, buttons, menus, or whatever mechanisms are appropriate for the device. Generally, Commands may be implemented using any user interface construct that has the semantics for activating a single action.

Commands are added to a Displayable (Canvas or Screen) with the method `addCommand` of class `Displayable`. The Commands can also be removed with the `removeCommand` method. This allows, for example, the application to control which Commands are available at a given time, based on user interaction. Some Commands can be paired with others or presented only in certain situations or context.

Each command has a type and a priority. Command types and priorities allow the device to place the commands to match the native user interface style of the device, and the device might put certain types of commands in standard places. For example, the "Go back" operation might always be mapped to the right softkey in a user interface style of a device. The device may decide to map a command with the BACK type to the right softkey. The `Command` class allows the application to communicate the abstract semantics of an action associated with a Command to the device so that action placements that are standard for that device can be used.

[\[Team LiB \]](#)

[!\[\]\(81fd4b9a321f9cc6bc9fcf2059ae2ebf_img.jpg\) PREVIOUS](#) [!\[\]\(1284087ebfed0d2a986ba4e0c867d73a_img.jpg\) NEXT](#) 

8.6 Advanced Topics

The following sections present more advanced topics related to the MIDP user interface libraries.

8.6.1 System Screens

Typically, the current screen of the foreground MIDlet is visible on the display. However, under certain circumstances, the system may create a screen that temporarily obscures the current screen of the application. These screens are referred to as system screens. This may occur if the system needs to show a menu or if the system requires the user to edit text on a separate screen instead of a `TextField` inside a `Form`. Even though the system screen obscures the application's screen, from the viewpoint of the application the notion of the current screen does not change. In particular, while a system screen is visible, a call to the method `getCurrent` will still return the application's current screen, not the system screen. The value returned by the method `isShown` is false while the current `Displayable` is obscured by a system screen.

If the system screen obscures a `Canvas`, the `hideNotify` method of the `Canvas` is called. When the system screen is removed, restoring the `Canvas`, its `showNotify` method and then its `paint` method are called. If the system screen was used by the user to issue a command, the `commandAction` method is invoked automatically after the `showNotify` method is called.

8.6.2 Adaptation to Device-Specific User Interface Style

The `lcdui` API contains several methods to query the user interface properties of the target device.

The `Display` class provides methods to query the color capability of the client device. The methods `isColor`, `numColors`, and `numAlphaLevels` provide information about whether the device has a color or monochrome display, the number of colors (or gray levels), and the number of supported alpha-channel levels in off-screen images, respectively.



The `Display` class also has methods to retrieve the prevailing foreground and background colors of the high-level user interface. These methods are useful for creating `CustomItem` objects that match the user interface of other items and for creating user interfaces within `Canvas` that match the user interface of the rest of the system. Implementations are not limited to using foreground and background colors in their user interfaces (for example, they might use highlight and shadow colors for a beveling effect), but the color values returned are those that match reasonably well with the color scheme of a device. An application implementing a custom item should use the background color to clear its region and then paint text and geometric graphics (lines, arcs, rectangles) in the foreground color.

The system colors can be queried with a method:

```
public int getColor(int colorSpecifier)
```


8.7 New for MIDP 2.0



We have marked a number of topics in this chapter as New! with an icon in the book margin. The user interface capabilities of the high-level user interface have been dramatically improved to allow increased control by the application developer over the layout and visual elements.

Class `Display` has been improved to allow:

- the querying of the colors used by the device for normal and highlighted colors for foreground, background, and borders (see the `getColor` method),
- the activation of the backlight and vibrator of the device as added effects to gain the attention of the user (see the `vibrate` and `flashBacklight` methods),
- the querying of the best width and height for Images used in Lists, Choice elements, and Alerts (see the `getBestImageWidth` and `getBestImageHeight` methods),
- the querying of the number of alpha levels supported (see the `numAlphaLevels` method), and
- the querying of the border style used for highlighted and normal elements (see the `getBorderStyle` method).

Class `Displayable` has been extended to allow:

- a title and Ticker to be set for any `Displayable` including `Canvas` (see the `setTitle` and `setTicker` methods), and
- the querying of the width and height of any `Displayable`; previously this was only possible for `Canvas` objects (see the `getWidth` and `getHeight` methods).

Abstract Commands have been enhanced to provide:

- the ability to create Commands with both a short label and long label; which one is used depends on the context and the available display area (see the constructor and `getLonglabel` methods).

[\[Team LiB \]](#)

[!\[\]\(7176a831836ec8f423c38e6250f870d7_img.jpg\) PREVIOUS](#) [!\[\]\(fa72f66d7098b5dd778d41096fe0c70a_img.jpg\) NEXT](#) 

[\[Team LiB \]](#)

[!\[\]\(64f5e75b13cd2ea173c1367c74de909f_img.jpg\) PREVIOUS](#) [!\[\]\(a0223f03ae64755501239d5f6c655962_img.jpg\) NEXT](#) 

9.1 List

The List class is a Screen that displays a list of choice items. The choice items are called elements. Each element has an associated string and may have an icon (an Image). There are three types of Lists: implicit, exclusive, and multiple choice. The type of the List is selected when the List is constructed and cannot be changed during the lifetime of the List object. The look and feel of Lists varies from one type of List to another, as illustrated in the examples below. Also, note that different MIDP implementations may render the Lists differently.

When a List is being displayed, the user can interact with it, for instance, by traversing from element to element. The traversal may cause the List to be scrolled. These traversing and scrolling operations are handled by the system and do not generate any events that are visible to the MIDP application. The system notifies the application only when a Command is fired. The notification to the application is carried out via the CommandListener of the Screen. This means that the application operations associated with list elements are added to the List as ITEM or SCREEN type commands, depending upon whether the operation affects only the selected element or the entire list.

In the List class there is also a notion of the select operation, which is central to the user's interaction with the List. The select operation is implemented differently from device to device. On devices that have a dedicated hardware key for selection operations, such as a "Select" or "Go" key, the select operation is invoked with that key. Devices that do not have a dedicated key must provide other means to perform the select operation; for example, using a labeled softkey. The behavior of the select operation within the three different types of lists is described in the following sections: [Section 9.1.1](#), "Implicit List," [Section 9.1.2](#), "Exclusive Choice List," and [Section 9.1.3](#), "Multiple Choice List."

Methods on List objects include insert, append and delete, and methods to get the String or the Image from any element. Elements can be added and removed from the List at any time. The selected elements in the List can be retrieved or changed. However, changing the List while the user is viewing it or trying to select an item is not recommended since it might cause confusion.

The majority of the behavior for managing a List is common with class ChoiceGroup; the common API is defined in the interface class Choice. Elements in the List are manipulated with the methods append, delete, getImage, getString, insert, and set. Item selection is handled with methods getSelectedIndex, setSelectedIndex, getSelectedFlags, setSelectedFlags, and isSelected.

Here is an example of List creation:



```
List list = new List("Fruits", . . .);
```


[\[Team LiB \]](#)

[!\[\]\(2903795c9b5aeb07867e5f4246b07acb_img.jpg\) PREVIOUS](#) [!\[\]\(fff4c5523a926e50d5a1332ca11a2485_img.jpg\) NEXT](#) 

9.2 TextBox

The TextBox class is a Screen that allows the user to enter and edit text. The input devices and input methods [1] for the text entry differ dramatically from device to device. Many devices have only a numeric keypad, which allows text entry by repetitive tapping of the number keys. Also, there can be separate user-changeable input modes: for example, separate modes for entering text and numbers.

[1] Input methods are software components that interpret user operations on input devices, such as typing keys, speaking, or writing using a pen device to generate text input.

Many numeric keypad devices have a more advanced input method called predictive input. [2] If there are several alternate input methods available, the user can often select which input mode to use in the editor. Some MIDP devices may have full-character keyboards, and devices with touch screen often have on-screen virtual keyboards. The input methods differ also from language to language. For example, in Chinese devices there are several different input modes. These different methods are not directly visible to applications, but developers should bear in mind that in many devices the input of long character strings might be cumbersome or rather slow. In MIDP 2.0, an application now has more concrete control on the input modes used, as described later in this subsection.

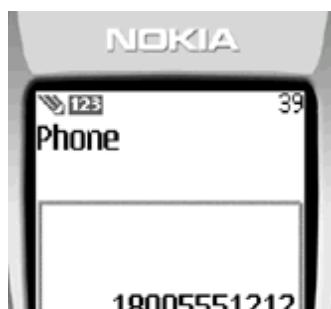
[2] Predictive input enables text entry with a numeric keypad using only one key press per letter in most common cases.

An application defines the maximum size of the TextBox, which is the maximum number of characters that the editor can contain. The application also sets the input constraints for the editor. These parameters are described in detail in the next section ([Section 9.2.1](#), "Input Constraints.") The characters in the TextBox can be modified and retrieved by the application either as a String or a sequence of characters.

The capabilities of the device determine a limit on the maximum size. The maximum size available may be smaller than the application had requested. The maximum size value is returned from both the setMaxSize and getMaxSize methods. The TextBox constructor and methods that change the text contents throw an IllegalArgumentException if the contents exceed maximum size. This means that when the application needs to use some initial value in the TextBox, it should compare the value returned from getMaxSize to the size of the String that it intends to use in the TextBox.

A TextBox must have Commands added to it. Otherwise, the user will not be able to trigger any action and will be stuck in the TextBox.

This sample code illustrates the creation of a TextBox with an initialized editable phone number and two commands.



[\[Team LiB \]](#)

[!\[\]\(b41fb99fc024f49cd078ae6414d5de01_img.jpg\) PREVIOUS](#) [!\[\]\(4c35d60f3c7e57a13f61c7ba1af3dbb8_img.jpg\) NEXT](#) 

9.3 Alert

Alerts are Screens that can be used to inform the user about errors and other exceptional conditions. Alerts can also be used as short informational notes and reminders. For the presentation of longer information, other Screen classes such as Form should be used.

An Alert shows a message and an optional Image to the user. An Alert is either presented for a certain period of time, or it can be modal with a special user operation to dismiss (close) the Alert.

The time the Alert is presented can be controlled with the setTimeout method. If the timeout is not set by application, the Alert timeout is the default timeout determined by the device. The default value can be inspected using the getDefaultTimeout method.

The Alert is set to be modal if the application sets the alert time to be infinite with the method call `setTimeout(Alert.FOREVER)`. A timed Alert is forced to be modal if the device cannot present the Alert contents without scrolling. In modal Alerts the device provides a feature that allows the user to dismiss the Alert, whereupon the next screen is displayed.

Timed Alerts can be used when the user does not need to be aware of the information presented and can safely ignore the Alert. Alerts without a timeout should be used when the user must be made aware of the information or condition.

When an Alert is dismissed either explicitly by the user or after a timeout, the current displayable is automatically changed to the next screen. When an Alert is displayed, the application can choose which Displayable is to be displayed after the Alert is complete. This is done by using a specialized setCurrent method of class Display:

```
public void setCurrent(Alert alert, Displayable nextDisplayable)
```

If the normal `setCurrent(Displayable nextDisplayable)` method is used, the display reverts to the current screen after an Alert exits.

The AlertType of an Alert can be set to indicate the nature of the information provided in the Alert. There are five AlertTypes; ALARM, CONFIRMATION, ERROR, INFO, and WARNING. ERROR and WARNING are used for error messages. WARNING should be used when the user has the possibility to prevent the error and ERROR when some non preventable error has already occurred. CONFIRMATION and INFO are used for more positive informational notes. CONFIRMATION is used when the user should confirm that some action has taken place. For example, after the user has selected some element from a settings List, a CONFIRMATION may be presented to the user to indicate that the setting has been changed. The INFO type is used for more generic positive informational notes. The ALARM type should be used when some notification is presented that interrupts the normal interaction in some other way. For example, a calendar application may use the ALARM type for notifying the user that a timed calendar event has been triggered.

The AlertType settings affect the presentation of the Alert. For example, if an application does not provide any image

9.4 Form

A Form is a Screen that may contain a combination of Items, including StringItems, ImageItems, editable TextFields, editable DateFields, Gauges, and ChoiceGroups. Any of the subclasses of Item defined by the MIDP Specification may be contained within a Form. The device handles layout, traversal, and possible scrolling automatically. The entire contents of the Form scroll up and down together. Horizontal scrolling is not generally available. This means that the MIDP Specification does not forbid a device to implement this, but it is not endorsed. When a device allows only vertical scrolling, the layout and traversal model is easier and matches the limited set of controls available on a typical mobile information device. Applications must be written with this limitation in mind. For example, a CustomItem subclass needs to be designed so that it works in varying screen widths without any horizontal scrolling support.



The methods for modifying the sequence of Items stored in a Form include insert, append, delete, get, set, and size. MIDP 2.0 devices also have a deleteAll method. Items within a Form are referenced by indices. The first Item in a Form has an index of zero and the last has the index size()-1.

The code sample below creates an empty Form container to be used as an options selection screen. This simple example has a title for the Form and Commands for accepting and canceling the Form contents. This kind of empty Form should be avoided in real applications.

Note that we will return back to this example in [Chapter 10](#) when we illustrate the use of the Form class in more detail. The code below serves as a starting point for an "Options" selection form of a photo album application. The example will continue in [Section 10.1](#), "Item," where we show how to add different Item class instances to this Form.



```
Form form = new Form( "Options" );
form.addCommand(backCommand);
form.addCommand(okCommand);
form.setCommandListener(this);
// Items are added later (see below)
display.setCurrent(form);
```

Refer to [Chapter 10](#), "MIDP High-Level User Interface ?Form" for more information on using the Form class.

[\[Team LiB \]](#)

[!\[\]\(2e04b8d4893bc7dea2c3662f2d8a6b43_img.jpg\) PREVIOUS](#) [!\[\]\(b2d1922445034a5a3e3f611dd4f80a28_img.jpg\) NEXT](#) 

9.5 New for MIDP 2.0



We have marked a number of topics in this chapter as New! with an icon in the book margin. Here is a summary of the most important new high-level user interface features of MIDP 2.0 described in this chapter.

Each of the Screen classes has been enhanced to provide more control over presentation and event handling.

The Font class has been extended to allow:

- the querying of the default Font for static text and input text (see the `getFont` method).

The List class has been extended to allow:

- the ability to get and set the Font of each element in the list (see the `setFont` and `getFont` methods),
- the control of the wrapping or truncation of text within elements (see the `setFitPolicy` and `getFitPolicy` methods),
- the ability to set the default command for selecting elements (see the `setSelectCommand` method),
- the ability to delete all of the elements in the list with a single method (see the `deleteAll` method), and
- the ability to set mutable Images as elements of a List (see the `append`, `insert`, and constructor methods).

The Alert class has been extended to allow:

- the Images in an Alert to be mutable (see the `setImage` method),
- a Gauge to be used as a progress indicator in the Alert (see the `setIndicator` and `getIndicator` methods),
- one or more Commands to be added and to replace the default dismiss command (see the `addCommand` and `removeCommand` methods), and
-

[\[Team LiB \]](#)

[!\[\]\(2ff6e28e83bf5bb93ee3bf3d1e233c1e_img.jpg\) PREVIOUS](#) [!\[\]\(776c3e79b2e438b459362762e6891a99_img.jpg\) NEXT](#) 

[\[Team LiB \]](#)

[!\[\]\(40c1ce98f6a8300334633b5896adcb33_img.jpg\) PREVIOUS](#) [!\[\]\(e410264afaecaa425a3f02b31ca3055d_img.jpg\) NEXT](#) 

10.1 Item

Class Item is a superclass of all those components that can be attached to a Form. An Item can be contained only in a single Form at a time. The application must remove the item from the Form in which it is currently contained before inserting it into another Form.

All Item objects can have a label field, which is a String representing the title of the Item. If the label String is null, then no label is presented. The label is typically displayed near the Item when the Item is visible on the screen. If the screen is scrolling, the implementation tries to keep the label visible at the same time as the Item.

10.1.1 ItemStateListener

When the user changes an editable Item in a Form, the application can be notified of the change by implementing the ItemStateListener interface and setting the listener on the Form. The itemStateChanged method of the ItemStateListener set on the Form is called automatically when the value of an interactive Gauge, ChoiceGroup, DateField, or TextField changes. The listener is set for the Form using the setItemListener method. It is not expected that the listener is called after every event that causes a change. However, if the value has changed, the listener is called sometime before it is called for another Item, or before a Command is delivered to the ItemStateListener of the Form.

Continuing the example started in [Section 9.4](#), "Form," the ItemStateListener would be set as follows:

```
form.setItemListener(this);
```

10.1.2 Item Commands



In MIDP 2.0, context-sensitive Commands can be added to Items. Commands added to an Item are made available to the user only when the focus is on that Item. As with Commands added to Displayables, the device implementation decides how the Commands are presented in the user interface. For example, the Commands might appear as a context-sensitive menu for the Item or as a normal menu when the Item is being manipulated by the user.

Context-sensitive Commands are used much like ordinary Commands in Displayables. The addCommand and removeCommand methods of the Item class control which Commands are associated with Items. There is a dedicated listener type called ItemCommandListener that is notified automatically when the user activates a Command of an Item. The listener of an Item is set with the setItemCommandListener method.

One of the Commands of an Item can also be chosen as the default Command using the method setDefaultCommand. A device may make this Command more easily accessible, for example, using the "Select" key of a device if such a key is available.

For an example on item command use, refer to the sample code in [Section 10.2.1](#), "Appearance Modes."

[\[Team LiB \]](#)

[!\[\]\(592d740df37e9ad0edffb31372889ebf_img.jpg\) PREVIOUS](#) [!\[\]\(b05363eb37c0e5a88f2a997e6732718e_img.jpg\) NEXT](#) 

10.2 StringItem

Read-only strings can be added to a Form either directly as Strings (Java String objects) or as StringItems. StringItem is a simple class that is used to wrap Strings so they can be treated consistently with the other Items. Strings are converted to StringItems automatically when they are appended to a Form. These kinds of StringItems have null labels. When an Item that was appended as a String is retrieved from a Form, it is returned as a StringItem.

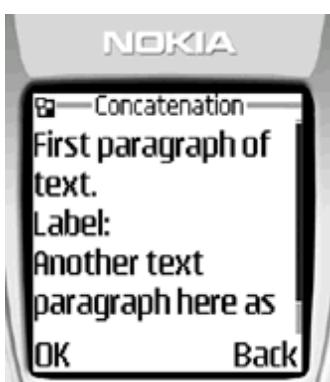
Continuing the example started in [Section 9.4](#), "Form," the following code illustrates how to add a new StringItem to the Form:



```
// Note: 'getImageName' is application-specific
String imageName = getImageName();
String si = new StringItem("Photo name:",
                           imageName);
form.append(si);
```

When a sequence of StringItems is displayed, the text is concatenated and wrapped to as many lines as needed. This has the effect that sequential StringItems are displayed as a paragraph of text. Any newline characters within StringItems are treated as row breaks (see [Section 10.9.1](#), "Row Breaks"). Also, a label may cause a row break either before or after the label, or both. Generally, the label is used as a paragraph title.

Here is an example that illustrates the use of concatenated StringItems:



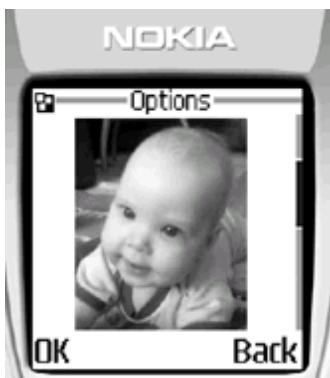
```
Form form = new Form("Concatenation");
... // Add Commands and CommandListener
form.append("First");
form.append("paragraph");
form.append("of text.");
Item si = new StringItem("Label", "Another");
```


10.3 ImageItem

Image objects can be added to a Form either directly as an Image or as an ImageItem. ImageItem is a simple class that wraps an Image. By default, the ImageItem is placed next to the previous StringItem or ImageItem. If an ImageItem is wider than the horizontal space available in the row, it is placed in a new row. Images wider than the Form width are clipped.

The ImageItem can be given a preferred layout policy with the setLayout method. The directive passed to this method defines whether the image is centered within the Form margins (LAYOUT_CENTER), left-justified (LAYOUT_LEFT), or right-justified (LAYOUT_RIGHT) and whether a forced line break is created before or after the image (LAYOUT_NEWLINE_BEFORE, LAYOUT_NEWLINE_AFTER). The layout is merely a hint for the device and might not be supported by the device. It also varies from device to device how the alignment directives are actually implemented. In some devices LAYOUT_LEFT might cause the ImageItem to float to the left margin, allowing nearby StringItems to wrap around the side of the image.

Continuing the example started in [Section 9.4](#), "Form," the following code adds an ImageItem to the Form.



```
Image image = Image.createImage(
    "/images/PhotoAlbum.png");
ImageItem imageItem =
    new ImageItem(null, // (no label)
        image,
        ImageItem.LAYOUT_CENTER |
        ImageItem.LAYOUT_NEWLINE_AFTER |
        ImageItem.LAYOUT_NEWLINE_BEFORE,
        "(preview image)");
form.append(imageItem);
```



There is also a LAYOUT_2 directive that causes an ImageItem to follow the strictly specified MIDP 2.0 Form layout rules (discussed in [Section 10.9](#), "Form Layout"). This means that the horizontal alignment directives LAYOUT_LEFT, LAYOUT_RIGHT, and LAYOUT_CENTER will behave the same way as in other Item classes. If LAYOUT_2 is not set, then the results are (for backwards compatibility reasons) implementation-specific since MIDP Specification version 1.0 did not define the behavior of the layout directives clearly for ImageItem.

10.4 TextField

A TextField is an editable text component that may be placed in a Form. The TextField API is exactly the same as that of TextBox. The only difference is the inherited functionality: TextField inherits from Item whereas TextBox inherits from Screen. Thus, as with TextBox, a TextField has a maximum size, input constraints and input mode (see [Section 9.2, "TextBox"](#)), a label, and a String value. The contents of a TextField can be either edited in-line in Form, or the Form may just present static text with a user operation to activate a separate editing view that allows the contents to be edited. The TextField contents within a Form are wrapped.

Devices dictate how many lines a TextField consumes. In some devices, the TextField height can grow dynamically based on the content.

An example:



```
TextField textField =
    new TextField("Photo caption:",
                 "", 32, TextField.ANY);
form.append(textField);
```

10.5 DateField

A DateField is a component that may be placed in a Form in order to present date and time (calendar) information. The user is able to select a date or time value for the field. Devices have different kinds of editors for this. Some may allow the user to select a value with a graphical date or time selector, and in others the value is simply entered using the numeric keypad. The value for a DateField can be initially set or left unset. If the value is not set, the DateField.getDate method returns null. The user interface must visually indicate that the date and time are unknown.

Each instance of a DateField can be configured to accept either date or time information, or both. This input mode configuration is done by choosing the DATE, TIME, or DATE_TIME mode of this class. The DATE input mode configures the DateField to allow only date information and TIME to allow only time information (hours, minutes). The DATE_TIME mode allows both clock time and date values to be set.

An example:



```
DateField date =
    new DateField("Date", DateField.DATE);
date.setDate(new java.util.Date());
// Set date to "now"
form.append(date);
```

10.6 ChoiceGroup

A ChoiceGroup defines a group of selectable elements that can be placed within a Form. A ChoiceGroup is similar to a List ([Section 9.1, "List"](#)), but it supports only the exclusive and multiple choice modes.



Also available is a ChoiceGroup-specific POPUP mode. The selection behavior of a pop-up choice is identical to that of an exclusive choice. The pop-up choice differs from an exclusive choice in presentation and interaction. Whereas the EXCLUSIVE type of ChoiceGroup presents all the elements in line as a radio button list, the POPUP type shows only the currently selected element in line in the Form. The other elements are hidden until the user performs an action to show them. When the user performs this action, all elements become accessible. For example, a device may use a pop-up menu to display the elements.

Generally, the device is responsible for providing the graphical representation of these ChoiceGroup modes and must provide a visually different representation for different modes. For example, it might use radio buttons for the exclusive choice mode, check boxes for the multiple choice mode, and pop-up menus without any specific graphics for pop-up mode.

An example:



```
ChoiceGroup choice =
    new ChoiceGroup("Size:",
                    Choice.EXCLUSIVE);
choice.append("Small", null);
choice.append("Large", null);
form.append(choice);
```

10.7 Gauge

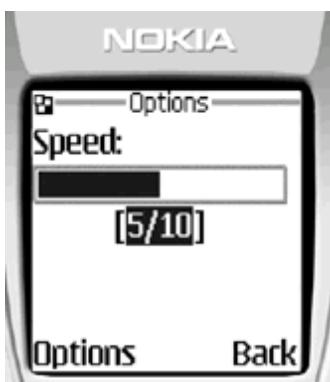
The Gauge class implements a graphical value display that may be placed in a Form. A Gauge displays a visual representation of a numeric value in the range between zero and the maximum value defined by the programmer. If the Gauge is set to be interactive, user actions can increase and decrease the value. Changes to the Gauge value are reported using an ItemStateListener. If the Gauge is set to be non-interactive, it represents a progress bar that can be used to provide feedback about the progress of long-running operations. The application can set and get the value of Gauge with the setValue(int) and getValue() methods. When Gauge is in non-interactive progress bar mode, the application must periodically update the value using the setValue method during the operation: the Gauge value should reach the maximum value just before the operation is finished.



Also available for a non-interactive Gauge is an indefinite range setting. This is used if there is no known endpoint to the activity. To define an indefinite Gauge, set the maximum value to the special INDEFINITE value. There are two kinds of indefinite Gauges available: CONTINUOUS_RUNNING and INCREMENTAL_UPDATING. These constants are used as Gauge range values. Continuous mode automatically animates the progress animation, whereas incremental mode requires that application call setValue(Gauge.INCREMENTAL_UPDATING) whenever there is an update in progress state.

For both of these modes there are separate idle modes that can be set with range values CONTINUOUS_IDLE and INCREMENTAL_IDLE, which indicate that no work is in progress. An application should use the idle modes with the correct pairs because the device progress graphics are typically designed so that, for example, CONTINUOUS_IDLE works well only in CONTINUOUS_RUNNING mode.

An example using interactive Gauge is:



```
Gauge gauge =
    new Gauge("Speed:", true, 10, 5);
form.append(gauge);
```

A non-interactive Gauge can also be placed in an Alert as an activity indicator. See [Section 9.3.2](#), "Activity Indicator," for details.

[\[Team LiB \]](#)

[!\[\]\(ee28bce606e2d37171120e83682c1db9_img.jpg\) PREVIOUS](#) [!\[\]\(15ae13bff6bb8baa2a0cd12872980d9a_img.jpg\) NEXT](#) 

10.8 CustomItem



The abstract class `CustomItem` allows applications to create new visual and interactive elements that can be used in Forms. To do this, a new subclass of `CustomItem` is created: it is responsible for creating the visual representation for the Item, including sizing and rendering. The subclass fully controls what is presented within item area. It defines the colors, fonts, and graphics that are used, including rendering of special highlight and focus states the item may have. Only the label of the item is rendered by the device, but the label is always rendered outside the `CustomItem` content area.

A `CustomItem` can trigger notifications to the `ItemStateListener` of a Form in which the `CustomItem` is contained. This is done by calling the `notifyStateChanged` method inherited from `Item`.

Like all Items, `CustomItems` have the concept of minimum and preferred sizes. These sizes pertain to the total area of the Item, which includes space for the content, label, borders, and so forth. (See [Section 10.9.4, "Item Sizing,"](#) for a full discussion of the areas and sizes of Items.) A `CustomItem` does not itself fully control this area. There is, however, a smaller region within the `CustomItem` called the content area that the `CustomItem` subclass paints and from which it receives input events.

A `CustomItem` subclass overrides the methods `getMinContentHeight`, `getMinContentWidth`, `getPrefContentHeight`, and `getPrefContentWidth` to return the minimum and preferred sizes for the content area. The device is responsible for calculating the minimum and preferred sizes for the whole item. The device also defines the actual content area space made available for `CustomItem`. The size of this area is given in calls to the methods `sizeChanged` and `paint`:

```
protected void sizeChanged(int w, int h)
protected abstract void paint(Graphics g, int w, int h)
```

The actual size might be smaller than the requested minimum content size if the device cannot allocate the space requested. For example, if no horizontal scrolling is available in a device, there is a device-specific maximum size for item width, which in turn affects the maximum content size for `CustomItems`.

If the minimum or preferred content area size changes, an application must call the method `invalidate`, and the subsequent calls to the size methods must return the new content area sizes. The method `invalidate` tells a device that it needs to perform its layout computation, which calls the content size methods to get new values based on the new contents of the `CustomItem`.

10.8.1 Interaction Modes

Each `CustomItem` is responsible for adapting its behavior to the interaction methods available on a target device. A `CustomItem` subclass needs to call the method `getInteractionModes` to inspect the interaction modes that are supported by a particular device. This method returns a bit mask of the supported modes with bits set or unset for the following modes:

[\[Team LiB \]](#)

[!\[\]\(ae969ebf1c1653141d65aca5e9b5b68f_img.jpg\) PREVIOUS](#) [!\[\]\(91ceaea7b8bf6b49653da08c23672fa9_img.jpg\) NEXT](#) 

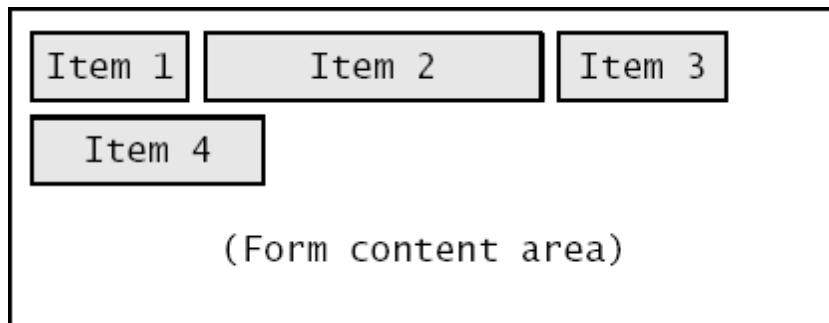
10.9 Form Layout

The Form class has a layout policy that is organized around rows. A Form grows vertically and scrolls vertically as necessary. The height of a Form varies depending upon the number of rows and the height of each row. The height of each row is determined by the items that are positioned on that row.



In MIDP 2.0, the Form layout algorithm considers each Item in turn, starting at Item zero and proceeding in order through each Item until the last Item in the Form has been processed. Items are laid out at the beginning of each row, proceeding across each row in the chosen layout direction, packing as many Items onto each row as will fit, unless a condition occurs that causes the packing of a row to be terminated early. A new row is then added, and Items are packed onto it as described above. Items are packed onto rows, and new rows are added below existing rows as necessary until all Items have been processed by the layout algorithm. This general principle of Form layout is illustrated in [Figure 10.1](#).

Figure 10.1. General principle of Form layout



An application can influence the layout of the Items using the following layout directives:

- LAYOUT_LEFT, LAYOUT_RIGHT, and LAYOUT_CENTER define the horizontal alignment of an item in a row.
- LAYOUT_TOP, LAYOUT_BOTTOM, and LAYOUT_VCENTER define the vertical alignment of an item in a row.
- LAYOUT_NEWLINE_BEFORE and LAYOUT_NEWLINE_AFTER force row breaks after and before an item.
- LAYOUT_SHRINK, LAYOUT_VSHRINK, LAYOUT_EXPAND, and LAYOUT_VEXPAND affect the way an item is sized.
- LAYOUT_2 indicates that new MIDP 2.0 layout rules are in effect for an item. LAYOUT_2 has no effect for StringItem, CustomItem, or Spacer.

[\[Team LiB \]](#)

[!\[\]\(f7e4bf0be545c5f16fafa1964a46cdd2_img.jpg\) PREVIOUS](#) [!\[\]\(31d9df6fe5f8d3868f68e1efb5758ccf_img.jpg\) NEXT](#) 

10.10 New for MIDP 2.0



We have marked a number of topics in this chapter as New! with an icon in the book margin. Here is a summary of the most important new high-level user interface features of MIDP 2.0 described in this chapter.

Classes TextBox and TextField have been improved to better control the display and input behavior by:

- adding the ability to set a hint for the initial input character set from an extensible set of Unicode subsets (see the setInitialInputMode method),
- adding a constraint to restrict the input of decimal numbers, including decimal point fraction digits (see the setConstraints method with the DECIMAL constraint),
- adding constraint modifiers PASSWORD, SENSITIVE, and NON_PREDICTIVE that restrict how the values are displayed or cached (see the setConstraints method), and
- adding constraint modifier hints to indicate that the first letter or each word or sentence should be capitalized (see the setConstraints method).

The functionality of the Form and Item classes has been significantly enhanced, as follows:

- There are more directives to specify the layout of each Item relative to the previous and successive Item. The options include positioning left, right, center, top, vcenter, and bottom; vertical and horizontal expansion and shrinking; and the control of line breaks (see the getLayout and setLayout methods).
- Spacer functionality has been added to provide a way to create empty spaces in the Form layout (see the Spacer class).
- New appearance modes plain, button, or hyperlink are available for StringItems and ImageItems.
- Commands can be added and removed from individual Items (see the addCommand and removeCommand methods).
- Interface ItemCommandListener can be used for defining context-sensitive command actions for Items (see the setItemCommandListener method).

Chapter 11. MIDP Low-Level User Interface Libraries

While the high-level user interface API provides maximum portability and development ease, some applications require greater control over the user interface.

The MIDP low-level user interface API is designed for applications that need precise placement and control of graphic elements as well as access to low-level input events. Typical examples of application components that might utilize the low-level user interface API are a game board, a chart object, or a graph.

Using the low-level user interface API, an application can:

- control precisely what is drawn on the display,
- handle primitive events like key presses and key releases, and
- access concrete keys and other input devices.

Applications that program to the low-level user interface API can be portable if the application uses only the standard features; applications should stick to the platform-independent part of the low-level API whenever possible. This means that applications should not directly assume the presence of any keys other than those defined in class `Canvas`. Also, applications should inquire about the size of the display and adjust their behavior accordingly.

[\[Team LiB \]](#)

[!\[\]\(05b32b046b7e22591c66224cbc2abd62_img.jpg\) PREVIOUS](#) [!\[\]\(8c1581abafdaa197848c32555f0402d3_img.jpg\) NEXT](#) 

11.1 The Canvas API

To directly use the display and the low-level user interface API, the developer uses the Graphics and Canvas classes. The Canvas class provides the display surface, its dimensions, and callbacks used to handle key and pointer events and to paint the display when requested. The methods of this class must be overridden by the developer to respond to events and to paint the screen when requested.

The Graphics class provides methods to paint lines, rectangles, arcs, text, and images to a Canvas or an Image.

The combination of the event-handling capabilities of the Canvas and the drawing capabilities of the Graphics class allows applications to have complete control over the application region of the screen. These low-level classes can be used, for example, to create new screens, implement highly interactive games, and manipulate images to provide rich and compelling displays for J2ME devices.

11.1.1 Canvas Dimensions

A Canvas does not necessarily have access to the entire screen of a J2ME device. Certain areas of the screen may be reserved for other purposes such as displaying network and battery information. In addition, the use of a title, Ticker, and Commands will further reduce the area available to a Canvas.

The current dimensions of a Canvas can be obtained by calling the methods getWidth and getHeight. Furthermore, the sizeChanged method of class Canvas is called whenever the dimensions of the Canvas change, thereby allowing it to update its layout based on the new size.



If an application needs to have as much screen space as possible, the setFullScreenMode method may be called to maximize the available area. When full screen mode is enabled, the Ticker and title are not shown even if they have been added to the Canvas. Commands are still shown, although the device may minimize the space used to display them; some devices may also hide system status indicators to maximize the area available for the Canvas.

11.1.2 Redrawing Mechanism

The application is responsible for redrawing the screen whenever repainting is requested. The application implements the paint method in its subclass of Canvas. Painting of the screen is done on demand so that the device can optimize the use of graphics and screen resources, for example, by coalescing several repaint requests into a single call to the paint method of class Canvas.

Requests to repaint the screen are made automatically by the device as needed; for example, when a Canvas is first shown or when a system dialog has been shown in front of the Canvas and then dismissed. The application itself may also request a repaint of the entire display by calling the repaint method of the Canvas object. If only part of the screen needs to be updated, another repaint method can be given the location, width, and height of the region that needs to be updated.

[\[Team LiB \]](#)

[!\[\]\(467fb4747609999a7912a4aba6cbb91c_img.jpg\) PREVIOUS](#) [!\[\]\(1ac5d5224e36a465440be81d736f671f_img.jpg\) NEXT](#) 

11.2 Low-Level API for Events in Canvases

11.2.1 Key Events

If the application needs to handle key events in its Canvas, it must override the Canvas methods `keyPressed`, `keyReleased`, and `keyRepeated`. When a key is pressed, the `keyPressed` method is called with the key code. If the key is held down long enough to repeat, the `keyRepeated` method is called for each repeated keyCode. When the key is released, the `keyReleased` method is called. Some devices might not support repeating keys, and if not, the `keyRepeated` method is never called. The application can check the availability of repeat actions by calling the method `Canvas.hasRepeatEvents`.

MIDP target devices are required to support the ITU-T telephone keys. Key codes are defined in the `Canvas` class for the digits 0 through 9, *, and #. Although an implementation may provide additional keys, applications relying on these keys may not be portable. For portable applications, the action key mappings described below should be used whenever possible, since other key codes are device-specific.

11.2.2 Action Keys

The `Canvas` class has methods for handling portable action events (also known as game actions) from the low-level user interface. The API defines a set of action events: UP, DOWN, LEFT, RIGHT, FIRE, GAME_A, GAME_B, GAME_C, and GAME_D. The device maps the action events to suitable key codes on the device; multiple keys may map to each game action. For example, a device with four navigation keys and a SELECT key in the middle could use those keys for mapping the action events, but it may also use the keys on the numeric keypad (such as 2, 4, 5, 6, and 8).

An application should determine the appropriate action for a given key code by calling the method `Canvas.getGameAction`. If the logic of the application is based on the values returned by this method, the application is portable and runs regardless of the keypad design.

The mapping between keys and the abstract action events does not change during the execution of an application. One of the device-specific key codes mapped to a particular action can be retrieved with the method `Canvas.getKeyCode`. However, since several keys may be mapped to a single action, comparing key events to these key codes is not recommended and `Canvas.getGameAction` should be used instead.

11.2.3 Pointer Events

The `Canvas` class has methods that the application can override to handle pointer events. Pointer events are events that are generated from a pointing device such as a stylus. If the application needs to handle pointer events, it must override and implement the methods `pointerPressed`, `pointerReleased`, and `pointerDragged`.

Not all MIDP target devices support pointer events, and therefore the pointer methods might never be called. The application can check whether the pointer and pointer motion events are available by calling `Canvas.hasPointerEvents` and `Canvas.hasPointerMotionEvents`.

[\[Team LiB \]](#)

[!\[\]\(b031865b899559cc68af31669ff7ece9_img.jpg\) PREVIOUS](#) [!\[\]\(e413adfbaed4b56f89d683a2f8c806dd_img.jpg\) NEXT](#) 

11.3 Graphics

All graphical operations are performed using a Graphics object. This object encapsulates a context for drawing to a particular target entity, including the current drawing color, clip region, line style, and so on.

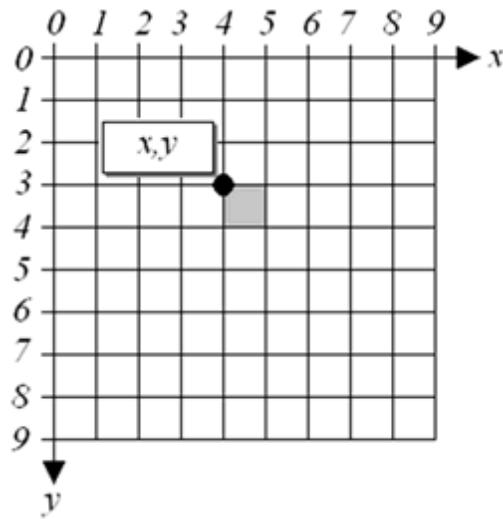
11.3.1 Coordinate System

The coordinate system's default origin is located at the upper-left corner, with the numeric values of the x-coordinates monotonically increasing from left to right, and the numeric values of the y-coordinates monotonically increasing from top to bottom.

Applications may assume that the horizontal and vertical values of the coordinate system represent equal distances on the actual device display. If the shape of the pixels of the device is significantly different from square, the device does the required coordinate transformation.

The coordinate system represents the locations of pixels, not the pixels themselves. Pixels have a width and height equal to 1 and extend down and to the left of their locating point. For example, the pixel in the upper-left corner of the display is defined as (0,0) and covers the area where $0 \leq x < 1$ and $0 \leq y < 1$. All coordinates are specified as integers. (See [Figure 11.1](#).)

Figure 11.1. Pixel coordinate system showing the pixel at (4,3)



11.3.2 Clipping

Each Graphics object has a single rectangular clip region; only those pixels within the clip region are modified by graphics operations. Operations are provided for intersecting the current clip rectangle with a given rectangle (`Graphics.clipRect`) and for setting the current clip rectangle outright (`Graphics.setClip`). The methods `Graphics.getClipWidth`, `Graphics.getClipHeight`, `Graphics.getClipX`, and `Graphics.getClipY` are used for obtaining the current clip region.

[\[Team LiB \]](#)

[!\[\]\(59aab71b78f6976850ef9b998f61c7a9_img.jpg\) PREVIOUS](#) [!\[\]\(ead91c80e7d9a1e1b6bf1ab9c4566bba_img.jpg\) NEXT](#) 

11.4 Creating and Using Images

11.4.1 Immutable Images

Images in a MIDP implementation may be either immutable or mutable. Immutable Images can be created directly from resource files, binary data, RGB data, or other Images. Once created, the contents of an immutable Image cannot be changed.

- The `Image.createImage(String name)` method is used to create an immutable Image from binary data in a resource file bundled with the application in the application's JAR file. The name must begin with "/" and include the full name of the binary file within the JAR file.
- The `Image.createImage(byte[], int offset, int length)` method is used to create an immutable Image from binary data contained in a byte array.
- The `Image.createImage(Image image)` method is used to create an immutable Image from another Image (which could be either mutable or immutable).
-  The `Image.createImage(java.io.InputStream stream)` method is used to create an immutable Image from a stream of binary data.
- The `Image.createImage(Image image, int x, int y, int width, int height, int transform)` method is used to create an immutable Image from the transformed region of another Image (which could be either mutable or immutable).
- The `Image.createRGBImage(int[] rgb, int width, int height, boolean alpha)` method is used to create an immutable Image from an array of RGB color values, either with or without alpha channel data.

If binary image data is used to create an Image, it must be in a format that is supported by the device. Though some devices may optionally support additional formats, all devices must support Portable Network Graphics (PNG) format as specified by the W3C-PNG (Portable Network Graphics) Specification, Version 1.0. W3C Recommendation, October 1, 1996. This specification is available at <http://www.w3.org/TR/REC-png.html> and as RFC 2083, available at <http://www.ietf.org/rfc/rfc2083.txt>.

11.4.2 Mutable Images

Mutable Images are created with specific dimensions and can be modified as needed. Mutable Images are created with the method `Image.createImage(int width, int height)` and are initially filled with white pixels. The `Image` has the

[\[Team LiB \]](#)

[!\[\]\(95ff6078a511e33b59a74f5036e1b370_img.jpg\) PREVIOUS](#) [!\[\]\(c3c98ae07a9ba888838e72649bbe0675_img.jpg\) NEXT](#) 

11.5 Drawing Primitives

The Graphics class provides various low-level drawing primitives. In general, drawing is started by calling the various methods of the Graphics object to set the color, translation, and clipping. Methods are available for drawing lines, various shapes, text, and Images. Each of the drawing primitives is explained next, along with a figure that illustrates their operation.

11.5.1 Drawing Lines

The Graphics.drawLine method draws a line from a starting point (x_1, y_1) to an ending point (x_2, y_2). The pixels are set to the value of the current color value. The lines are drawn with the current stroke style (see [Section 11.3.5](#), "Line Styles.")

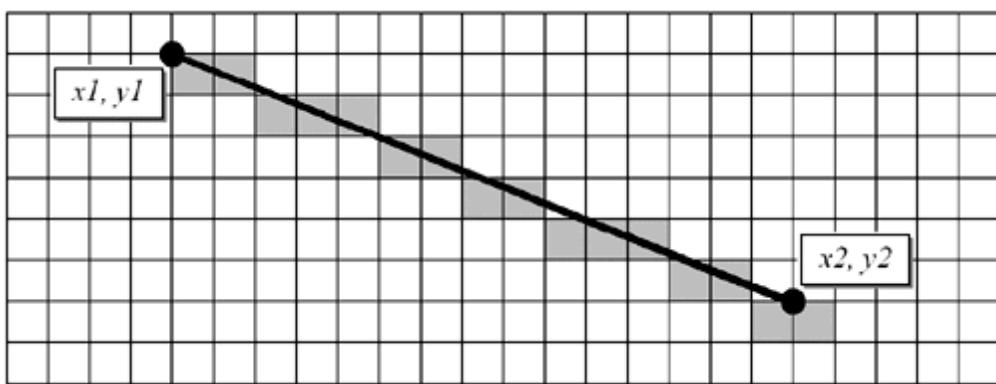
```
drawLine(int x1, int y1, int x2, int y2)
```

Since a pixel extends below and to the right of its point, the overall width and height of the line are one pixel larger than the difference between the two x values and the two y values, respectively. Thus, one pixel is drawn if (x_1, y_1) and (x_2, y_2) are identical.

[Figure 11.5](#) shows the result of the following code:

```
g.drawLine(4, 1, 19, 7);
```

Figure 11.5. Example of the drawLine method



11.5.2 Drawing and Filling Arcs

The Graphics.drawArc method draws the outline of a circular or elliptical arc bounded by the a rectangle whose origin is (x, y) and whose size is specified by the width and height. The arc is drawn using the current color and stroke style, starting at startAngle degrees and extending for arcAngle degrees. The start angle is interpreted such that 0 degrees is at the 3 o'clock position. A positive angle indicates a counter-clockwise rotation, while a negative value indicates a clockwise rotation. See the following example code and [Figure 11.6](#).

```
drawArc(int x, int y, int width, int height, int startAngle,  
       int arcAngle)
```


[\[Team LiB \]](#)

[!\[\]\(77e15936d2cadaff9885688dd2f51331_img.jpg\) PREVIOUS](#) [!\[\]\(e22d9b27f8f0ebd65edc98c5f85c5883_img.jpg\) NEXT](#) 

11.6 New for MIDP 2.0



We have marked a number of topics in this chapter as New! with an icon in the book margin. Here is a summary of the most important new low-level user interface features of MIDP 2.0.

The low-level user interface has been improved to allow more control and easier programming of Canvas and Image level actions. Significant additions are support for alpha blending and for manipulating Images as arrays of pixel values.

The Canvas class has been changed as follows:

- A full screen mode has been added that allows the maximum display to be used for application graphics (see the setFullScreenMode method.)
- The methods for setting the Ticker and title have been moved to Displayable so that these features can be used in Canvas objects as well.

The Graphics class has been improved with:

- the ability to draw a rectangular portion of the source Image with image transformation including rotation and mirroring (see the drawRegion method),
- the ability to copy of pixels within an off-screen Image (see the copyArea method),
- the ability to fill triangles, which is useful when filling irregularly shaped areas (see the fillTriangle method),
- the ability to draw pixels into the Image from an array of RGB pixel values, possibly with alpha values (see the drawRGB method), and
- the ability to return the color that will be used in the display (see the getDisplayColor method).

The Image class has been extended to allow easier creation of Image objects by adding methods for:

- creating an Image from a subregion of an existing Image and with a specified rotation and mirroring transform (see the appropriate createImage method),

Chapter 12. MIDP Game API



Game applications have proven to be one of the most popular uses for Java 2 Micro Edition, so a dedicated Game API is provided in Mobile Information Device Profile version 2.0 to facilitate game development. This API provides several benefits to the game developer. First, it simplifies game development and provides an environment that is more familiar to game developers. Second, it reduces application size and complexity by performing operations that would otherwise be implemented by the application. Third, because the API can be implemented using native code, it can also improve performance of the frequently used game routines.

As with the low-level user interface APIs, applications that use the Game API must be written carefully in order to ensure portability. Device characteristics such as screen size, color depth, and key availability must be accounted for by the developer if application portability is desired.

[\[Team LiB \]](#)

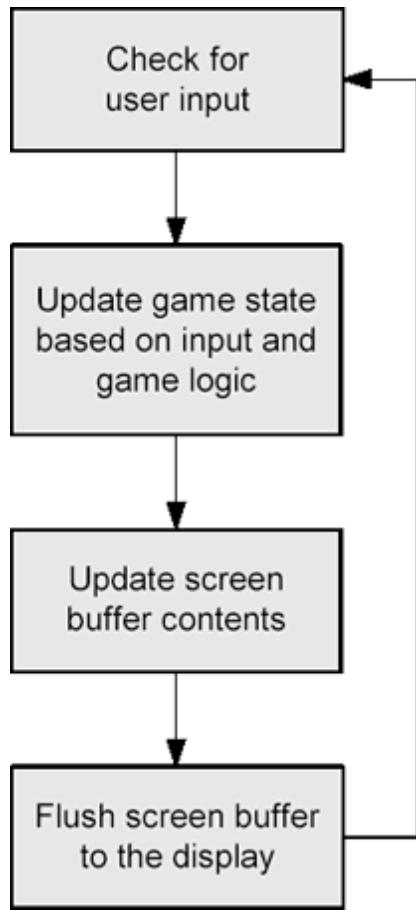
[!\[\]\(8aa63a504f4624af1c3d4fd5869c72ab_img.jpg\) PREVIOUS](#) [!\[\]\(b25477f54ce57ffd5a1664c125cab14b_img.jpg\) NEXT](#) 

12.1 The GameCanvas API

The Canvas class was designed for applications that are event-driven; that is, the user interface is updated in response to a user input, such as selecting an option or pressing a soft key. However, game applications are generally time-driven and continue to run and update the display whether or not the user provides input. To overcome this difference, the GameCanvas class provides a game-centric model for accessing the display and checking for key inputs.

Game applications typically employ a single thread to run the game. This thread executes a main loop, shown in [Figure 12.1](#), that repeatedly checks for user input, implements logic to update the state of the game, and then updates the user interface to reflect the new state. Unlike Canvas, the GameCanvas class provides methods that directly support this programming model.

Figure 12.1. Game loop flowchart



12.1.1 Key Polling

The GameCanvas class also provides the ability to poll the status of the keys instead of having to implement callback methods to process each key event. Key polling allows all of the application's key handling logic to be coded together and executed at the appropriate point in the game loop. It also enables simultaneous key presses to be detected on devices that support them.

12.2 Layers

The Layer class is an abstract class that represents a basic visual entity that can be rendered on the screen. A Layer has defined rectangular bounds, and it can be either visible or invisible. A Layer can be rendered using any Graphics object by calling its paint method.

The game API defines two Layer subclasses: Sprite and TiledLayer. Unfortunately, class Layer cannot be directly subclassed by the developers, since this would compromise the ability to improve performance through the use of native code in the implementation of the Game API.

Subclasses of class Layer can be drawn at any time using the paint method. The Layer will be drawn on the Graphics object according to the current state information maintained by the Layer object (that is, position, visibility, and so forth.) The Layer object is drawn at its current position relative to the current origin of the Graphics object. Erasing the Layer is always the responsibility of the code outside the Layer class.

[\[Team LiB \]](#)

[!\[\]\(613627aa28da42fdb2de1b788ed406ce_img.jpg\) PREVIOUS](#) [!\[\]\(4fa8f355c43bccfd3170859aaabea334_img.jpg\) NEXT](#) 

12.3 Sprites

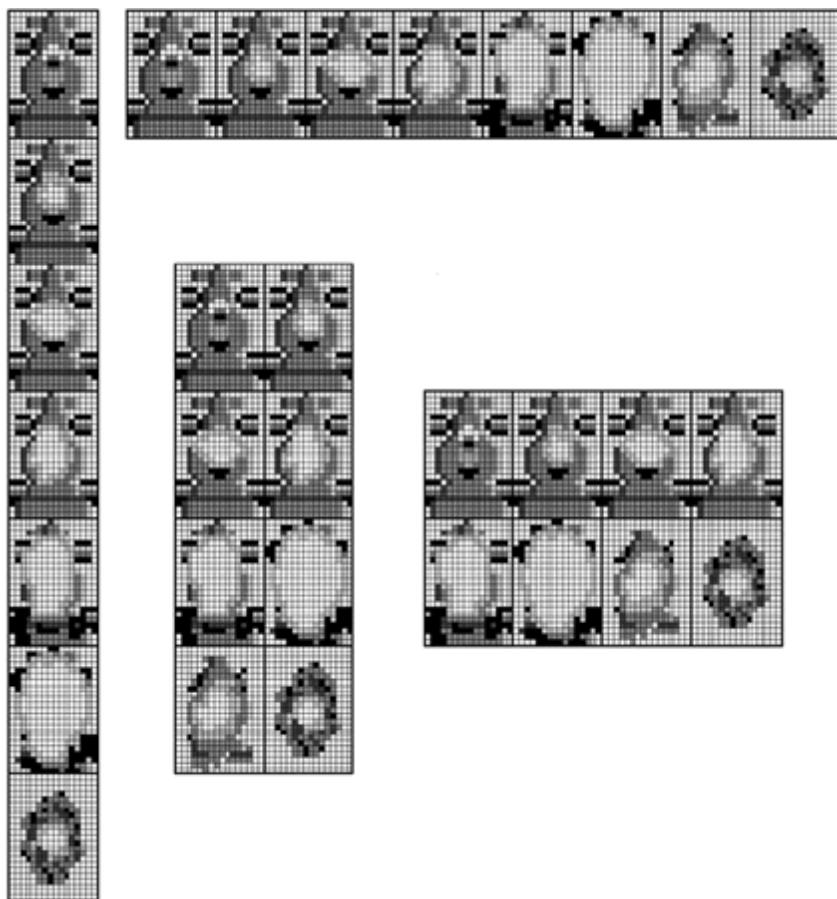
A Sprite is a basic visual element whose appearance is provided by one of several frames stored in an Image. The developer can control which frame is used to render the Sprite, thereby enabling animation effects. Several transforms such as flipping and rotation can also be applied to a Sprite to further vary its appearance. As with all Layer subclasses, a Sprite's location can be changed, and it can also be made visible or invisible.

12.3.1 Frames

The raw frames used to render a Sprite are provided in a single Image object, which may be mutable or immutable. If more than one frame is used, they are packed into the Image as a series of equally-sized frames. The frames are defined when the Sprite is instantiated; they can also be updated by calling the setImage method.

As shown in [Figure 12.2](#), the same set of frames may be stored in several different arrangements depending on what is most convenient for the game developer. The implementation automatically determines how the frames are arranged based on the dimensions of the frames and those of the Image. The width of the Image must be an integer multiple of the frame width, and the height of the Image must be an integer multiple of the frame height.

Figure 12.2. Frames can be arranged in various ways within an Image



The Image is divided into frames whose dimensions are dictated by the frameWidth and frameHeight parameters in the Sprite's constructor. Each frame within the Image is assigned a unique index number; the frame located in the upper-left corner of the Image is assigned an index of 0. (See [Figure 12.3](#).) The remaining frames are then numbered

[\[Team LiB \]](#)

[!\[\]\(c92eee6016fa8ffb7ac4ad43ef3cabed_img.jpg\) PREVIOUS](#) [!\[\]\(9399c8f4b8d4a71f80a092ff2e63447d_img.jpg\) NEXT](#) 

12.4 TiledLayer

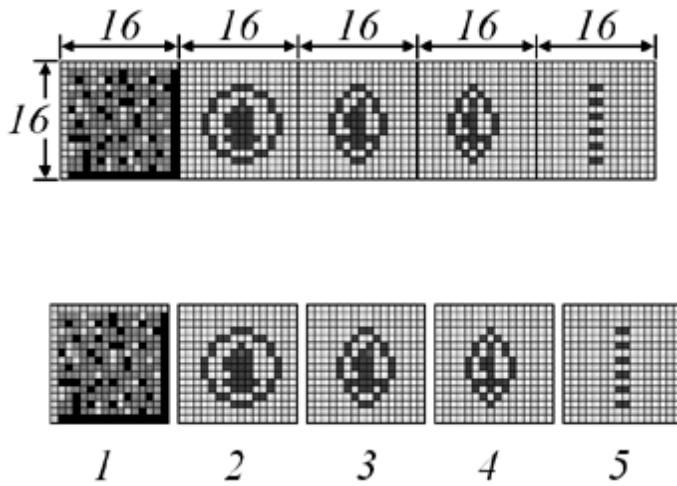
A TiledLayer is a Layer that is composed of a grid of cells that can be filled with a set of tile images. This technique is commonly used in 2D gaming platforms to create very large scrolling backgrounds without the need for an extremely large Image. The use of animated tiles allows numerous cells of a TiledLayer to be animated quickly and easily.

12.4.1 Tiles

As with a Sprite's frames, the tiles used to fill the TiledLayer's cells are provided in a single Image object which may be mutable or immutable. The Image is broken up into a series of equal-sized tiles whose dimensions are specified along with the Image.

Each tile is assigned a unique index number, as shown in [Figure 12.7](#). The tile located in the upper-left corner of the Image is assigned an index of 1 (unlike Sprite where the first frame is given an index of 0). The remaining tiles are then numbered consecutively in row-major order (indices are assigned across the first row, then the second row, and so on). These tiles are regarded as static tiles because there is a fixed link between the tile and the image data associated with it. The static tile set is created when the TiledLayer is instantiated; it can also be updated at any time using the setStaticTileSet method.

Figure 12.7. A TiledLayer's static tile set is obtained from an Image



In addition to the static tile set, the developer can also define several animated tiles. An animated tile's appearance is provided by the static tile to which it is linked. This link can be updated dynamically, thereby allowing the developer to change the appearance of several cells without having to update the contents of each cell individually. This technique is very useful for animating large repeating areas without having to explicitly change the contents of numerous cells.

Animated tiles are created using the createAnimatedTile method, which returns the index to be used for the new animated tile. The animated tile indices are always negative and consecutive, beginning with ?. The associated static tile that provides the animated tile's appearance is specified when the animated tile is created, and can also be dynamically updated using the setAnimatedTile method.

12.4.2 Cells

[\[Team LiB \]](#)

[!\[\]\(4a34c307d64a5099a208d4baa9103003_img.jpg\) PREVIOUS](#) [!\[\]\(1050cfd3c6b55b49971fbe53416940e0_img.jpg\) NEXT](#) 

12.5 LayerManager

The LayerManager manages a series of Layers and simplifies the rendering process by automatically drawing the correct regions of each Layer in the appropriate order.

The LayerManager maintains an ordered list to which Layers can be appended, inserted, and removed. A Layer's index correlates to its z-order; the layer at index 0 is closest to the user while the Layer with the highest index is furthest away from the user. The indices are always contiguous; that is, if a Layer is removed, the indices of subsequent Layers will be adjusted to maintain continuity.

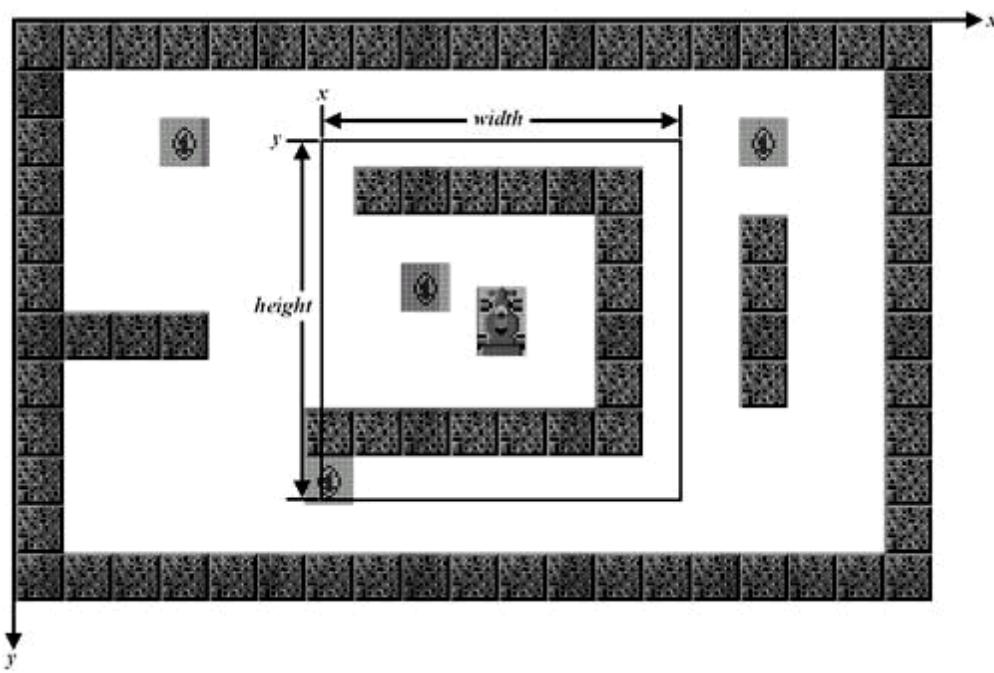
The LayerManager class provides several features that control how the game's Layers are rendered on the screen. The size of the view window controls how large the user's view will be, and is usually fixed at a size that is appropriate for the device's screen.

The position of the view window can also be adjusted relative to the LayerManager's coordinate system. Changing the position of the view window enables effects such as scrolling or panning the user's view. For example, to scroll to the right, simply move the view window's location to the right.

In the example shown in [Figure 12.11](#), the view window is set to a region that is 120 pixels wide, 120 pixels high, and located at (102, 40) in the LayerManager's coordinate system.

```
layerMgr.setViewWindow(102, 40, 120, 120);
```

Figure 12.11. Setting the view window



The paint method of LayerManager class includes an (x,y) location that controls where the contents of the view window are rendered relative to the screen. Changing these parameters does not change the contents of the view window itself, only the location where it is drawn on the screen. Note that high values in

[\[Team LiB \]](#)

[!\[\]\(bddeedb2edb7599e21d020205801653f_img.jpg\) PREVIOUS](#) [!\[\]\(3ba2797f7f1be336a106b626c259a0aa_img.jpg\) NEXT](#) 

12.6 Collision Detection

Collision detection is a key feature of the Game API, enabling the developers to quickly and easily determine whether two elements of a game have collided or not.

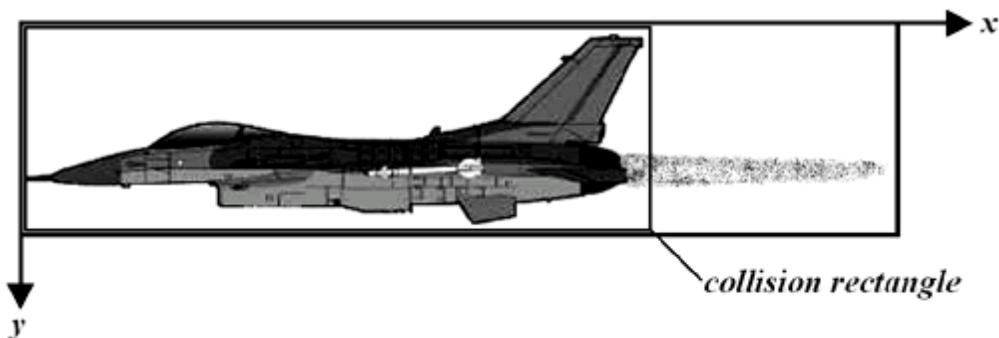
The collision detection methods are provided in the Sprite class, and they allow the developer to detect a collision between a Sprite and either a TiledLayer, an Image, or another Sprite.

12.6.1 Collision Rectangle

Sprites have a defined collision rectangle that indicates the region to be included for collision detection. By default, this region corresponds to the overall bounds of the Sprite. However, it can be redefined to cover any rectangular region as needed.

A special collision rectangle allows areas of a Sprite to be excluded from the collision detection process. For example, a jet fighter Sprite may include a vapor trail coming from the engines; however, a missile colliding with this part of the Sprite should not be considered as hitting the aircraft. By defining a collision rectangle that includes only aircraft and excludes the vapor trail (see [Figure 12.13](#)), selective collision detection can be easily achieved.

Figure 12.13. Setting a collision rectangle



The collision rectangle is automatically updated to reflect any transforms that are applied to the Sprite. Therefore, there is no need to modify the collision rectangle when a transform operation is performed.

12.6.2 Pixel-Level Detection

The collision detection methods allow the developer to specify boundary-level or pixel-level detection. Boundary-level detection checks if the collision rectangles of the two elements are intersecting or not, and it is the simplest and fastest detection scheme. However, many elements are likely to be non rectangular in nature, so boundary-level detection may not be sufficient.

To overcome this problem, the developer may request pixel-level detection. Used in conjunction with transparent image data, this detection scheme checks whether an opaque pixel in the first element is touching an opaque pixel in the second element. Thus, the transparent area around the opaque area is excluded from the detection process, and a

[\[Team LiB \]](#)

[!\[\]\(582fa7b4d5971f89a4d5428b540d25f9_img.jpg\) PREVIOUS](#) [!\[\]\(5dc7544209ea9645d477b9896d06f8f1_img.jpg\) NEXT](#) 

12.7 Sample Code: A Simple Game

This example illustrates how the APIs discussed in this chapter can be used to form a simple game. The user controls a car that is always moving forward at a constant speed and can be turned 90 degrees clockwise or counterclockwise. The object of the game is to drive around the track and collect as many coins as possible without hitting any of the walls. (See [Figure 12.16](#) on the next page.)

Figure 12.16. A screen shot from the simple game



The car is implemented using a Sprite that can be rotated in 90-degree increments using the appropriate transform. Special frame sequences are used to show just the first frame or to animate through the explosion frames.

The track and the coins are implemented using TiledLayers. Most of the cells in the tiled layers are left empty so that the dark gray background is seen. A static tile is used to form the walls of the track, and an animated tile is used to display the spinning coins.

Two separate TiledLayers are used to simplify the collision detection code. Collision detection provides a simple way to determine whether the car has collided with a non-empty cell in either of the TiledLayers. If the collision was with the coins, some simple code determines which cells containing the coins can be removed and the score can be accurately updated.

Note that this code example is simplified to demonstrate the use of the game API; a real application would require additional code to appropriately handle application starting and pausing, game restarts, displaying the score, multiple levels, and so on.

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import javax.microedition.midlet.*;
import java.io.*;

/**
 * Simple car game that demonstrates
 * the MIDP 2.0 Game API
 */
public class SimpleGame extends MIDlet {
```


12.8 New for MIDP 2.0



The Game API is an entirely new feature in MIDP 2.0. With the Game API, application developers can simplify the development of game applications by using a programming paradigm that is well suited to game applications. Since the Game API provides many operations that would otherwise be implemented by the application, the API can significantly reduce the size and complexity of game applications.

Chapter 13. MIDP Sound API



This chapter introduces the Sound API of MIDP 2.0. In the first version of the MIDP Specification, sound support was minimal. With the release of the MIDP Specification version 2.0, this area has now been significantly enhanced. A well-defined, scalable architecture for sound and multimedia has been created for devices running J2ME, with primary emphasis on mobile phones using the Mobile Information Device Profile. In addition to offering basic sound programming support in MIDP 2.0, an optional package called the Mobile Media API has been created to enable generic multimedia functionality. A diverse set of applications will benefit from sound and media support in J2ME devices. Examples of such applications include games, speech, and multimedia applications.

The purpose of this chapter is to give the reader an overview of the MIDP 2.0 Sound API and the complementary optional package, the Mobile Media API (MMAPI), created by the Java Community Process effort JSR 135.

[\[Team LiB \]](#)

[!\[\]\(9afc4eb38b2f1d5b954b397015b659a7_img.jpg\) PREVIOUS](#) [!\[\]\(0c14d1920edc4133a59b1fee9267be5e_img.jpg\) NEXT](#) 

13.1 Overview of the MIDP 2.0 Sound API

13.1.1 MIDP Media Support

MIDP devices range from low-end mass-market phones with basic sound support to high-end media phones with advanced audio and video rendering capabilities. To accommodate these diverse configurations and multimedia processing capabilities, an API with a high level of abstraction and support for optional features is needed. The central goal of the specification work for the Sound API was to address the wide range of application areas. As a result, two API sets were developed:

-
- MIDP 2.0 Sound API
-
- Mobile Media API (MMAPI)

The division into two interoperable API sets arose from the fact that some devices are very resource-constrained. It may not be feasible to support a full range of multimedia types, such as video, on some mobile phones. As a result, not all MIDP 2.0 target devices are required to support the full generality of the Mobile Media API, such as the ability to support custom protocols and content types.

MIDP Sound API

The MIDP 2.0 Sound API is intended for resource-constrained devices such as mass-market mobile phones. The basic level of sound support mandated for all devices is the presence of monophonic buzzer tones. This API is a directly compatible subset of the full Mobile Media API. Furthermore, due to its building block nature, this subset API can be adopted to other J2ME profiles that require sound support.

Mobile Media API

The full Mobile Media API is intended for J2ME devices with advanced sound and multimedia capabilities. The target devices for the Mobile Media API include powerful mobile phones, as well as PDAs and set-top boxes.

Applications

Sound and media are a key feature in many applications. Games, music applications, user interface tones, and alerts are examples of application areas that will benefit from the MIDP Sound API.

In the following sections, the MIDP Sound API is discussed in more detail. In [Section 13.4, "Enhanced Media Support Using the Mobile Media API,"](#) a more detailed look at the MMAPI is provided.

13.1.2 Design Goals

[\[Team LiB \]](#)

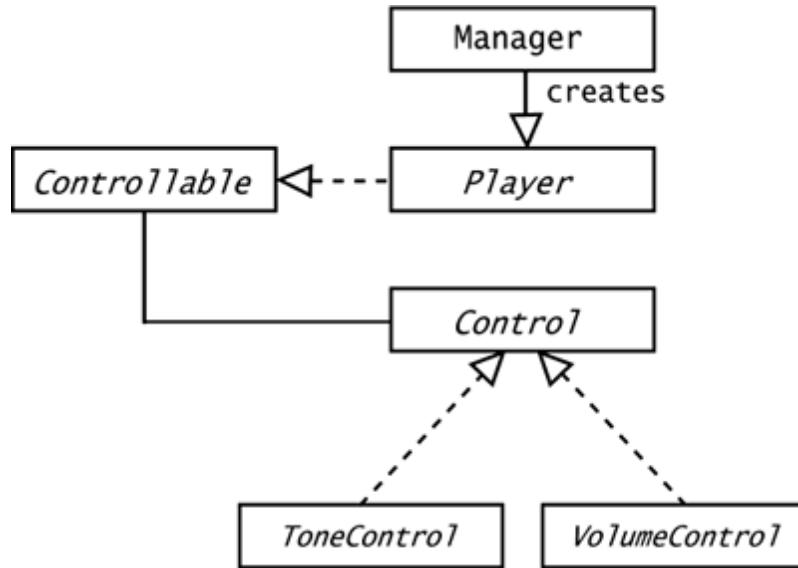
[!\[\]\(2f2b2c158ae762d905dcf6da4fb6fd2a_img.jpg\) PREVIOUS](#) [!\[\]\(cf4e49a1af813bfb7fded6b674a8040c_img.jpg\) NEXT](#) 

13.2 Player Creation and Management

In this section, the process of creating and managing sound in a MIDP 2.0 application is described. The functionality of classes Manager and Player are described, and simple application examples are provided to illustrate the details of the API.

[Figure 13.2](#) shows the class diagram of the MIDP 2.0 Sound API. (Interfaces are shown in italic font.)

Figure 13.2. Class diagram of the MIDP Sound API



13.2.1 Managing a Media Framework

The entry point to the Sound API is the Manager class. Class Manager provides access to an implementation-specific mechanism for constructing Player objects. In addition, class Manager provides means for property queries of content and protocol support. For convenience, it also provides a simplified method to generate simple tones.

Property queries

The MIDP sound API provides a mechanism for querying the type of content and protocols that the device supports. The property queries are enabled by the methods `Manager.getSupportedContentTypes` and `Manager.getSupportedProtocols`.

For example, if the given protocol is "http:", then the supported content types that can be played back with the http protocol will be returned. If null is passed in as the protocol, all the supported content types for this device will be returned. The returned array must be non-empty; in other words, at least one content type has to be supported.

13.2.2 Creating Players for Media Data

[\[Team LiB \]](#)

[!\[\]\(c48726687b6f6495339e9234477012a7_img.jpg\) PREVIOUS](#) [!\[\]\(ba77fb0e1e0cc06fb19baa19038459af_img.jpg\) NEXT](#) 

13.3 Media Controls

Control objects are used for controlling content-type-specific media processing operations. The set of operations are usually functionally related. Thus a Control object provides a logical grouping of media-processing functions.

All Control classes implement the interface Controllable, and the Player interface is a subinterface of Controllable. Therefore, a Player implementation can use the Control interface to extend its media-processing functions. For example, a Player can expose a VolumeControl to allow the volume level to be set.

The javax.microedition.media.control package specifies a set of predefined Controls classes. In MIDP 2.0, two controls are specified and required: ToneControl and VolumeControl.

The Controllable interface provides a mechanism for obtaining the Controls from an object such as a Player. It provides methods to get all the supported Controls and to obtain a particular Control based on its class name.

13.3.1 ToneControl

ToneControl is an interface that enables the playback of a user-defined monotonic tone sequence. A tone sequence is specified as a list of tone duration pairs and user-defined sequence blocks. The list is packaged as an array of bytes. The setSequence method is used for defining the sequence for the ToneControl.

Here is the syntax of a tone sequence, described using an Augmented BNF notation. (See <http://www.ietf.org/rfc/rfc2234> for details of the Augmented BNF notation.)

```
sequence          = version *1tempo_definition
                  *1resolution_definition
                  *block_definition 1*sequence_event

version          = VERSION version_number
VERSION          = byte-value
version_number   = 1           ; version # 1

tempo_definition = TEMPO tempo_modifier
TEMPO            = byte-value
tempo_modifier   = byte-value
; multiply by 4 to get the tempo (in bpm) used
; in the sequence.

resolution_definition = RESOLUTION resolution_unit
RESOLUTION        = byte-value
resolution_unit   = byte-value

block_definition  = BLOCK_START block_number
                  1*sequence_event
                  BLOCK_END block_number
BLOCK_START       = byte-value
BLOCK_END         = byte-value
block_number      = byte-value
; block_number specified in BLOCK_END has to be
; the same as the one in BLOCK_START
```


[\[Team LiB \]](#)

[!\[\]\(39deee626765a13e6a5afa0112ed8550_img.jpg\) PREVIOUS](#) [!\[\]\(a72c5c659b6c9e53ad54d578f65cd8c3_img.jpg\) NEXT](#) 

13.4 Enhanced Media Support Using the Mobile Media API

As discussed earlier in this chapter, MIDP Sound API is a subset of the Mobile Media API (MMAPI) defined by the Java Community Process effort JSR 135. MMAPI includes a lot of additional functionality that is not present in the MIDP 2.0 Sound API. The additional features introduced by the Mobile Media API include the following:

- Custom protocols through the DataSource
- Synchronization of multiple Players using the TimeBase interface
- Interactive synthetic audio
- Video and camera functionality
- Capture and recording
- Metadata

The scalability and optional use of the API is crucial for the successful deployment of the API in mobile devices. It is clear that not all implementations of MMAPI will support all multimedia types and input protocols. For example, some of the devices may support only playback of local audio files. The design of the MMAPI allows implementations to provide optional support for different media types and protocols.

As with the MIDP 2.0 Sound API, class Manager provides the getSupportedContentTypes and getSupportedProtocols methods to query for supported media types and protocols. An application can also attempt to create a Player from class Manager given the media input. If the content type or the protocol specified is not supported, the Player creation methods will throw an exception.

Because MMAPI is an optional package, it does not mandate any particular media types or protocols. Required types and protocols are to be determined by the appropriate profiles adopting MMAPI (for example MIDP 2.0). However, an implementation must guarantee support of at least one media type and protocol.

MMAPI also allows implementations to support optional features. For example, some implementations may support only media playback but not recording. MMAPI allows implementations to expose these optional features as applicable.

Optional features are organized as Control classes. A Player can be queried for all of its supported Controls using the method getControls or a particular type of Control using the method getControl.

13.5 New for MIDP 2.0



In the MIDP Specification version 1.0, sound support was minimal. With the release of the MIDP Specification version 2.0, sound support has been significantly extended. A well-defined, scalable Sound API has been created for devices supporting the Mobile Information Device Profile. A diverse set of applications?ncluding games and media applications?ill benefit from the MIDP 2.0 Sound API.

Chapter 14. MIDP Persistence Libraries

The MIDP Specification provides a mechanism for MIDlets to persistently store data and retrieve it later. This persistent storage mechanism, called the Record Management System (RMS), is modeled after a simple, record-oriented database. This chapter introduces the key features and characteristics of the Record Management System.

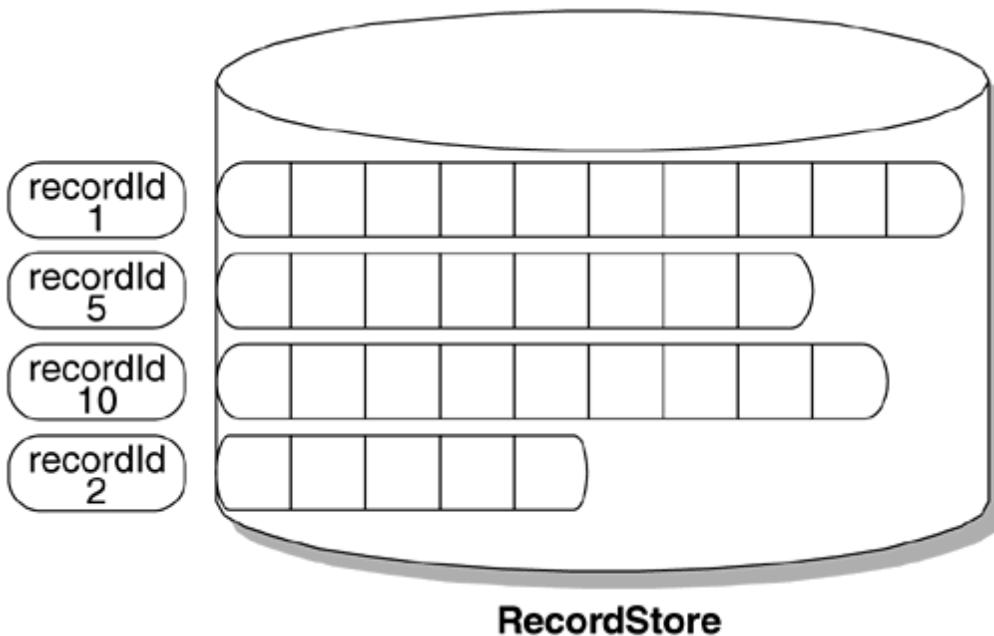
[\[Team LiB \]](#)

[!\[\]\(9eebd7279f79276be4325b3be5e72137_img.jpg\) PREVIOUS](#) [!\[\]\(1198f1cc428b9f11bf8c55b9c578fae3_img.jpg\) NEXT](#) 

14.1 The Record Management System

Conceptually, RMS provides records and record stores, as shown in [Figure 14.1](#). A record store is a collection of records that is persistent across multiple invocations of a MIDlet. Each record is an array of bytes. Each record in a record store can be of a different length and can each store data differently. For example, in [Figure 14.1](#) the topmost record, Record ID 1, may contain a String followed by an int, while the second record, Record ID 5, may contain an array of short numbers.

Figure 14.1. Structure of a record store



Associated with each record in a record store is a unique identifier called the recordId. This identifier can be used to retrieve a record from the record store (see `getRecord` in [Section 14.2.3](#), "Manipulating Records in a Record Store"). In [Figure 14.1](#), "adjacent" records in the record store do not necessarily have consecutive recordIds. The underlying implementation may in fact reorganize records within the store or place two consecutively placed records in "far" apart locations. All that is guaranteed is that if a record is added to the record store, it will retain its recordId until deleted. A recordId is assigned via a monotonically, increasing-by-one algorithm. There is no provision for "wrap around" of recordIds. However, since the data type for recordIds is an integer that is capable of storing a number as large as 2,147,483,647, it is highly unlikely that a given recordId will ever be reused within a record store, especially since mobile information devices usually have limited persistent storage.

MIDlets within a MIDlet suite can access one another's record stores directly; however, no locking operations are provided by the RMS. Therefore, if a MIDlet suite uses multiple MIDlets or threads to access a record store, it is the developer's responsibility to coordinate these access operations with the synchronization primitives provided by the Java programming language. Note that the system software of the mobile information device is responsible for maintaining the integrity of RMS record stores throughout the normal use of the platform, including reboots, battery changes, and so forth. Therefore, the programmer does not have to worry about these issues.



The sharing of RMS record stores is controlled by the MIDP application model (see [Section 19.1.4](#), "MIDlet Suite Execution Environment"). Under this model, record stores are shared

[\[Team LiB \]](#)

[!\[\]\(4f3a446dee1ce83fc938f7e3fce55fb8_img.jpg\) PREVIOUS](#) [!\[\]\(937289c0ea25797dd54854fa29a34f29_img.jpg\) NEXT](#) 

14.2 Manipulating Record Stores and Records

Record stores have two basic types of operations: those that deal with manipulating the record store as a whole and those that deal with manipulating the records within the record store. A summary of different record store and record manipulation operations is provided in the following sections.

14.2.1 Manipulating a Record Store

The programmer accesses record stores by name. These names are case-sensitive and may contain between 1 and 32 Unicode characters. The name space for record stores is a flat, non-hierarchical space.[\[1\]](#)

[1] Note: The name space of record stores is separate from that of resources (resources are accessed by calling the method `java.lang.Class.getResourceAsStream()`.) Thus, the name of a record store can be lexically equivalent to that of a resource, but the two are separate entities.

Within a MIDlet suite, record store names are unique. In other words, MIDlets within a MIDlet suite are not allowed to create more than one record store with the same name. On the other hand, no such restriction applies to different MIDlet suites: a MIDlet within one MIDlet suite is allowed to have a record store with the same name as a MIDlet in another MIDlet suite. In that case, the record stores are distinct and separate. A MIDlet can obtain a list of all the record stores owned by the containing MIDlet suite by calling the class static method `listRecordStores` of class `RecordStore`.

In order to access a record store, the record store must be opened first. A MIDlet can open one of its record stores with the method `openRecordStore` that takes two parameters: a String containing the name of the record store, and a boolean indicating whether the record store should be created if it does not exist yet. The method `openRecordStore` returns an object of type `RecordStore`, which provides access to the following methods that return information about the associated record store:

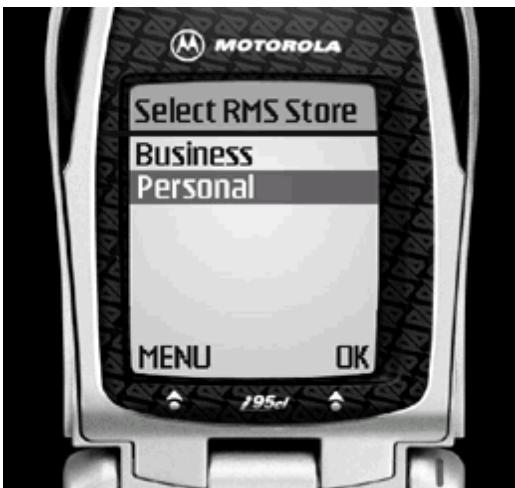
- getName: returns a String that represents the name with which this record was opened or created.
 -
 - getNumRecords: returns an int that represents the number of records currently in the record store.
 -
 - getSize: returns an int that represents the current number of bytes contained in the record store.
 -
 - getSizeAvailable: returns an int that represents the number of remaining bytes in the device that the record store could potentially occupy. Note that the number returned by this method is not a guarantee that the record store can actually grow to this size, since another MIDlet may allocate space for another record store.
 -
 - getNextRecordID: returns an int that represents the recordId for the next record added to the record store.
 -
 - getVersion: returns an int that represents the version of the record store. Each time a record is added

[\[Team LiB \]](#)

[!\[\]\(eccbef7b5322ee5df131bd84e6e8ab3e_img.jpg\) PREVIOUS](#) [!\[\]\(f9db0da7de14e91ae6aa86c7f3a4bb1a_img.jpg\) NEXT](#) 

14.3 Sample Code (RMSMIDlet.java)

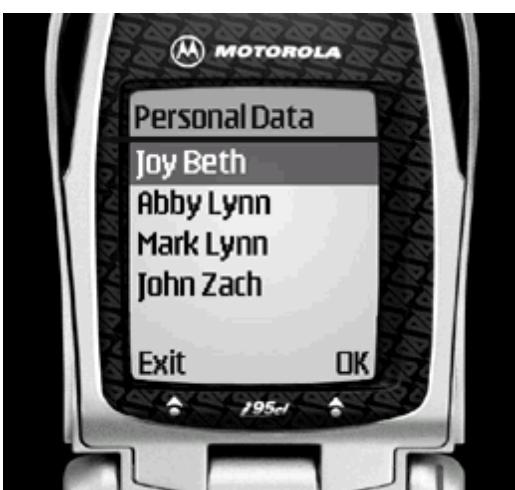
The following small sample application, RMSMIDlet.java, demonstrates some of the capabilities of RMS. Only the main class and one non-public implementation class of this application are shown.



The RMSMIDlet example creates two record stores called Personal and Business and allows the user to view the contents of the record store and the details of the individual records using the operations discussed in [Section 14.2.1](#), "Manipulating a Record Store").

When the MIDlet begins, the application opens the screen that is depicted in the figure above.

If the user selects the OK command, then the content of the selected database (in this case, Personal) is shown, as depicted in the figure to the right.



If the user navigates through the MENU command and selects the DETAIL command, then the detail screen is shown. (See the figure on the next page.)

Class RMSMIDlet is implemented as a MIDlet, so its structure and functionality follows the general guidelines defined for MIDlet classes. This means that the class must implement the methods `startApp`, `pauseApp`, and `destroyApp` to define the behavior upon application startup, pausing, and exit, respectively.

14.4 New for MIDP 2.0



The most requested feature for RMS in MIDP 2.0 was the ability to share record stores between MIDlet suites. This was addressed by adding an access mode to class RecordStore so that the owner of the RecordStore can allow shared or private access, and can differentiate between read-only or read-write access modes. The MIDlet suite that creates the RecordStore owns it and controls access to it. Other MIDlet suites can open shared record stores by giving the MIDlet suite name and the record store name. This new functionality makes it possible to share persistent data between MIDlet suites.

Chapter 15. MIDP Networking and Serial Communications

MIDP target devices operate on a wide variety of wireless and wired networks with incompatible transport protocols. In addition, wireless networks have less bandwidth, more latency, and less stable connections than wired networks. The MIDP Specification version 2.0 must accommodate these differences and allow for future network technologies. To do so, MIDP uses standard network programming mechanisms and practices and takes advantage of the existing network infrastructure.



MIDP specifies that HTTP and HTTPS are required protocols for all devices. In addition, it defines APIs for optional protocols: datagrams, sockets, secure sockets, and serial communication ports. (See [Chapter 16](#), "Secure Networking" for more information on the secure networking protocols.)

15.1 Characteristics of Wireless Data Networks

The MIDP Specification version 2.0 addresses two types of wireless data networks: circuit-switched data (CSD) and packet-switched data (PSD). A wireless circuit-switched network (also called a session-based network) assigns a user a radio channel, which it dedicates to the connection until the data transfer is done. During the data exchange, the channel cannot be used for anything else. The GSM cellular phone network is a circuit-switched data network.

Data over a circuit-switched network is usually billed on a time basis, like voice transmissions. Using a circuit-switched wireless network for data transfer can be expensive because data rates in today's wireless networks are rather low. In a GSM phone, for example, the data rate is usually about 9.6 kilobits per second (kbps).

The other type of wireless data network is packet-switched. In packet-switched data transmissions, the data exchange is broken up into small packets that usually have a fixed length. Packets from several different data exchanges, each from different users, can be sent in time slots of the same radio channel. The packets are mixed during transmission and reassembled in the correct order by the recipients. Packet-switched data uses the radio spectrum more efficiently than circuit-switched data. In addition, packet-switched data is usually billed by the amount of data sent, so packet-switched networks are generally cheaper for data transfer.

The next-generation wireless-network technologies promise much greater data rates than the networks in widespread use today. The second-and-a-half generation (2.5G) wireless technologies, such as General Packet Radio Service (GPRS), and third-generation (3G) technologies, such Wideband Code Division Multiple Access (WCDMA), are generally packet-switched technologies. For example, the theoretical maximum data transfer rate of GPRS is 171.2 kilobits per second, and third generation wireless networks offer theoretical data rates up to a few megabits per second.

15.2 Network Interface Considerations

MIDP devices must support the HTTP protocol, a rich, widely used protocol that can be implemented relatively easily over a variety of wireless networks. HTTP lets devices take advantage of the extensive server-side infrastructure already in place for wired networks. Furthermore, because no unrequested HTTP packets can arrive at the device, implementing the networking interface is easier.

Supporting the HTTP protocol does not imply that the device must support a particular Internet Protocol (IP). A device can implement HTTP using IP protocols such as TCP/IP, or non-IP protocols such as WAP or i-Mode. If a device uses a non-IP protocol, a computer called a gateway bridges the wireless protocol to and from HTTP servers on the Internet and provides IP-based network facilities such as server name resolution (for example, DNS).



Many wireless networks have evolved to support IP-based protocols, so the MIDP Specification version 2.0 defines the corresponding APIs and behaviors. The new MIDP 2.0 APIs are `SocketConnection` ([Section 15.5](#), "SocketConnection"), `ServerSocketConnection` ([Section 15.6](#), "ServerSocketConnection"), and `UDPDatagramConnection` ([Section 15.7](#), "UDPDatagramConnection"). In addition, the `CommConnection API` ([Section 15.8](#), "CommConnection") supports serial communications. Support for these protocols is optional, but most networks are expected to support them. They enable MIDlets using the protocols to run consistently across implementations.

Implications for application developers



When designing MIDP applications, it is important to keep in mind that many networking and communications API calls can be time-consuming and can potentially block the application thread. For instance, networking operations such as opening connections or receiving data from a server can cause the application thread to be blocked while waiting for the network activity to be completed.

Application developers should design their applications so that the applications do not invoke any time-consuming network operations in the callback functions that are invoked as a result of user interface actions. For instance, `CommandListener` functions should not generally invoke networking functions, as listeners require a separate thread to keep the user interface functionality from interfering with the responsiveness of the application. In addition, upon invoking time-consuming networking operations, the application should always display a progress screen, such as `Gauge`, and include a "Cancel" command to allow the user to escape from potentially indefinitely blocking network activity.

[\[Team LiB \]](#)

[!\[\]\(04acd4df06df8bdae88f714041d63e47_img.jpg\) PREVIOUS](#) [!\[\]\(e20c53682c6f03be51b985238d43667f_img.jpg\) NEXT](#) 

15.3 The HttpConnection Interface

The interface javax.microedition.io.HttpConnection defines the MIDP HTTP API. This interface is unchanged from MIDP 1.0. The interface extends the interface javax.microedition.io.ContentConnection to provide additional fields and methods that parse URLs, set request headers, and parse response headers. (Refer to the documentation of [interface ContentConnection](#) on page 69.)

15.3.1 States of an HTTP Connection

An HTTP connection exists in one of three states: setup, connected, or closed.

A connection is in the setup state after it is opened and before the request has been made to the server. The application sets information needed to connect to the server, such as the request parameters and headers. (They must be set before the request is sent.) The methods `setRequestMethod` and `setRequestProperty` set the values; the `getRequestMethod` and the `getRequestProperty` methods retrieve them.

A connection enters the connected state when the request is sent. This occurs when the application calls a method that requires a response value or closes the output stream, if any. These methods indicate the end of the request.

A connection enters the closed state when the connection is closed. Although none of the methods of the `HttpConnection` object can be used when the connection is closed, its `InputStream` or `OutputStream` might still contain data. The streams may be used until they are closed. The following code fragment shows an input stream being used after its connection has been closed:

```
// Open the connection and input stream
HttpConnection c = (HttpConnection)Connector.open(...);
InputStream is = c.openInputStream();

...
// Close the connection. Can't call HttpConnection methods
// after this, but InputStream methods are still available.
c.close();
while ((int ch = is.read()) != -1) {
    ...
}
// Close the input stream. Can't call InputStream methods after this
is.close();
```

15.3.2 Security of HTTP

Access to the HTTP connection may be restricted by device policy. Trusted MIDlet suites request access with the `javax.microedition.io.Connector.http` permission, as described in [Section 18.3](#), "Trusted MIDlet Suite Security Model."

The device may try to authorize the MIDlet when the MIDlet calls the `Connector.open` method with a valid URL. If the authorization fails, the MIDlet is not allowed to use the HTTP API. (See [Section 18.1](#) for more information about device policies.)

[\[Team LiB \]](#)

[!\[\]\(c32818e9d2907cf53e5172e2c942d1a2_img.jpg\) PREVIOUS](#) [!\[\]\(2eeeefcb904b43a483756a786ccacf10_img.jpg\) NEXT](#) 

15.4 Sample Code (NetClientMIDlet.java)

The following application, NetClientMIDlet, demonstrates how HTTP can be used to log in to a server, send data, and receive a response.



The MIDlet begins by displaying the instructional screen shown in the top figure. If the user selects OK, the MIDlet sends a POST request that contains a user identifier and password for a basic authentication scheme. It also sends the message *Esse quam videri* ("To be rather than to seem.")



After sending the message, the MIDlet reads the returned values. A screen full of the values are shown in the next figure: a string identifying itself (such as "NetServer Servlet" as shown in the bottom figure), followed by a date string, the original POST message body, the message body reversed, and a complete list of headers read from the POST request. (The figure only shows the first header.)

The example consists of three classes. The NetClientMIDlet class extends the MIDlet class. It implements the MIDlet life cycle methods and has the device display the screens shown in the figures. The ConnectionManager class opens the HTTP connection to a server, sends the information, and reads the response. The BasicAuth class transforms the user identifier and password into a base64 encoded string.

NetClientMIDlet.java

[\[Team LiB \]](#)

[!\[\]\(4198493efa7bf64a39c25537403cd5a5_img.jpg\) PREVIOUS](#) [!\[\]\(08cef83cbce6572a409f80267b05cd55_img.jpg\) NEXT](#) 

15.5 SocketConnection



The `SocketConnection` interface extends interface `javax.microedition.io.StreamConnection` to define the API to TCP/IP or similar socket streams. (Refer to the documentation of [interface StreamConnection](#) on page 69.)

Applications that need to accept connections from other applications must use `ServerSocketConnection` instead. See [Section 15.6](#), "ServerSocketConnection" for more information.

An application opens a socket on a target host by calling the `Connector.open` method. The application must provide a URI string that starts with 'socket://' and is followed by a host name and port number separated by a colon. For example, 'socket://host.com:79' has the correct syntax to open a socket connection on the host.com system at port 79.

In the string the MIDlet can specify the host as a fully qualified host name or IPv4 number. Note that RFC1900 [[reference 2](#)] recommends the use of host names rather than IP numbers. Using the host name means the MIDlet can still make the connection, even if the host's IP number is reassigned. For example, 'socket://host.com:79' is preferred to 'socket://183.206.241.133:79'. If the connection string uses incorrect syntax, `Connector.open` throws an `IllegalArgumentException` exception.

The `Connector.open` method returns a `SocketConnection` instance. It has methods that enable the MIDlet to get the local and remote addresses and the local and remote port numbers, and to control socket options.

15.5.1 Getting the Local Address and Port Number

An application can get the local host address and port number for an open connection. (This information is needed primarily by applications that use a `ServerSocketConnection` to allow clients to make inbound connections to a system-assigned port. See [Section 15.6](#), "ServerSocketConnection" for more information.) The `getLocalAddress` method returns the local IP number to which the socket is bound. Calling `System.getProperty("microedition.hostname")` returns the host name of the device, if it is available. The `getLocalPort` method returns the local port to which this socket is bound.

If an application sends address and port information to a remote application so that it can connect to the device, the application should try to send the host name of the device. IP addresses can be dynamically assigned, so a remote application with only an IP address will not be able to make contact if the device IP number changes. If a host name is available, sending it would help the remote application be more robust.

15.5.2 Getting the Remote Host Address and Port

An application can get the remote host address and port number for an open connection. The `getAddress` method returns the remote host (as either a host name or an IP address) to which the socket is bound. The `getPort` method returns the remote port to which this socket is bound. The methods get their return values from the string passed to `Connector.open`.

[\[Team LiB \]](#)

[!\[\]\(212a47b2e9df3e42cf438809edaab8cf_img.jpg\) PREVIOUS](#) [!\[\]\(41afc8587fb165d306acec98b8e78a70_img.jpg\) NEXT](#) 

15.6 ServerSocketConnection



The ServerSocketConnection interface defines the server socket stream API. A server socket connection is not a socket or stream connection; it only establishes that the application is accepting incoming connections. The ServerSocketConnection interface extends the interface javax.microedition.io.StreamConnectionNotifier, which listens for incoming connections. (Refer to the documentation of [interface StreamConnectionNotifier](#) on page 69.)

An application establishes a server socket connection by calling the Connector.open method. The application must supply a URI string with the host omitted. For example, "socket://:7" defines an inbound server socket on port 7 (the echo port). If the application also omits the port number, the device dynamically selects an available port. For example, "socket://" defines an inbound server socket on a port allocated by the system. If the connection string is invalid, the Connector.open method throws an IllegalArgumentException.

When the application needs to connect to the socket (for example, to handle an incoming request) it invokes the acceptAndOpen method of the server socket connection. The acceptAndOpen method returns a SocketConnection instance. See [Section 15.5](#), "SocketConnection," for information on the returned instance. As with other blocking network calls, the application should invoke these methods only in a thread designated for the purpose; use of the user interface callbacks to invoke these methods will block the user interface.

15.6.1 Getting the Local Address and Port Number

An application can get the address and port on which the device will accept connections. The getAddress method returns the IP address of the device. The getPort method returns the assigned port number. See [Section 15.5.1](#), "Getting the Local Address and Port Number" for more information, but remember that the values returned from the ServerSocketConnection versions of the methods are not the address and port of an active SocketConnection.

15.6.2 Security of ServerSocketConnection

Access to a server socket connection is restricted to prevent unauthorized transmission or reception of data. MIDlet suites needing to use server sockets must request the javax.microedition.io.Connector.serversocket permission for Trusted MIDlet suites as described in [Section 18.3](#), "Trusted MIDlet Suite Security Model."

The device tries to authorize the MIDlet when the MIDlet calls the Connector.open method with a valid server socket connection string. If the authorization fails and the MIDlet is not allowed to use the server socket API, the Connector.open method throws a java.lang.SecurityException. The device also checks the MIDlet's permission when the MIDlet calls the acceptAndOpen method and might check when the MIDlet calls the openInputStream, openDataInputStream, openOutputStream, or openDataOutputStream methods.

15.6.3 Example

The following example shows a ServerSocketConnection providing an echo service. (It does not, however, use the

[\[Team LiB \]](#)

[!\[\]\(0b44df6d5955a5c3c125ef44231840d1_img.jpg\) PREVIOUS](#) [!\[\]\(27f6bcf52251eb534b327366c15d6953_img.jpg\) NEXT](#) 

15.7 UDPDatagramConnection



The UDPDatagramConnection interface extends the interface `java.microedition.io.DatagramConnection` to define a UDP datagram connection. (Refer to the documentation of [interface DatagramConnection](#) on page 69.) The UDP datagram protocol is defined by the IETF standard RFC768 [[reference 1](#)]. The protocol is message-oriented; delivery and duplicate protection are not guaranteed. Applications use UDP datagram connections to exchange messages with minimal overhead, but if ordered reliable delivery of streams of data is required, a `SocketConnection` is a better choice.

An application opens a UDP datagram connection by calling the `Connector.open` method. The application must provide a URI string that starts with "datagram://" and is followed by a host name and port number separated by a colon. If the string has neither the host nor port fields, the device allocates an available port. If the connection string is invalid, an `IllegalArgumentException` is thrown.

15.7.1 Getting the Local Address and Port Number

An application can get the local host address and port number for an open connection. (This information is needed primarily for applications that use `UDPDatagramConnections` to allow clients to send datagrams to a dynamically assigned port.) The `getLocalAddress` method returns the local IP number to which the socket is bound. Calling the method `System.getProperty("microedition.hostname")` returns the host name of the device, if it is available. The `getLocalPort` method returns the local port to which this socket is bound.

If an application sends address and port information to a remote application so that it can connect to the device, the application should try to send the host name of the device. IP addresses can be dynamically assigned, so a remote application with only an IP address will not be able to make contact if the device IP number changes. If a host name is available, sending it would help the remote application be more robust.

15.7.2 Datagrams

The `UDPDatagramConnection` interface inherits methods to create, send, and receive datagrams from the interface `javax.microedition.io.DatagramConnection`. Once created, a datagram can be used and reused for reading or writing. A Datagram consists of an address and a data buffer. The address is a URL string in the same syntax as described for the method `Connector.open`. If the datagram will be sent, the application uses the `Datagram.setAddress` method to assign a destination address. (The address must include both the host and port.) If the Datagram was received, the address contains the source address.

The data buffer is a byte array with an offset and a length. An application can access the byte array either directly or indirectly through the datagram's `DataInputStream` for reading and `DataOutputStream` for writing. The datagram's `getOffset` and `setOffset` methods are the accessor methods for the offset of the first byte of data. The `getLength` and `setLength` methods are the accessor methods for the number of bytes of data in the datagram. The data buffer, offset, and length must be set before the Datagram is sent. Before receiving a Datagram, the buffer, offset, and length must also be set to indicate the bytes available to store the incoming message.

[\[Team LiB \]](#)

[!\[\]\(6c3c9e773a1e4002d96ad9db84e0c6c6_img.jpg\) PREVIOUS](#) [!\[\]\(5850ed41dc11d7a523ff84824eb1a22d_img.jpg\) NEXT](#) 

15.8 CommConnection



The CommConnection interface defines an API for a logical serial port connection. The CommConnection extends the StreamConnection interface, through which bytes are transferred using input and output streams. The underlying operating system defines the logical serial ports that can be used. Such a port does not need to correspond to a physical RS-232 serial port. For instance, IrDA IRCOMM ports are commonly configured as logical serial ports within the device.

An application opens a serial port by calling the Connector.open method. The application must provide a URI string that begins with "comm:" followed by a port identifier, followed optionally by a semicolon and a semicolon-separated list of parameter-value pairs. The name of a parameter is separated from its value with an equals sign (=). Spaces are not allowed in the string. For example, "comm:COM0;baudrate=19200" is a valid string.

The port identifier is a logical device name defined by the device. Port identifiers will probably be device-specific, so applications should use them with care. (See [Section 15.8.3](#), "Recommended Convention for Naming Ports.") Applications can get a comma-separated list of valid port identifiers for a particular device and operating system by calling the System.getProperty method with the key microedition.commports.

Any additional parameters must be separated by a semicolon. If a particular optional parameter is not applicable to a particular port, the device is permitted to ignore it. Legal parameters are listed in [Table 15.3](#). If the application supplies an illegal or unrecognized parameter, the Connector.open method throws an IllegalArgumentException. If the device supports a parameter value, it must be honored. If the device does not support a parameter value, the Connector.open method throws a java.io.IOException. If an option duplicates a previous option, the last value overrides the previous value.

Table 15.3. CommConnection parameters for opening serial connections

Parameter	Default	Description
baudrate	platform dependent	Speed of the port in bits per second (such as 9600 or 19200)
bitsperchar	8	Number of bits per character (7 or 8)
stopbits	1	Number of stop bits per char (1 or 2)
parity	none	Parity (even, odd, or none)
blocking	on	Whether to wait for a full buffer. (on, which means wait, or off, which means do not wait)

15.9 New for MIDP 2.0



MIDP Specification version 2.0 introduced several new network and communication APIs. The `SocketConnection` API can be used for reliable connections with servers without the overhead of HTTP. The `ServerSocketConnection` API allows a device to handle incoming connections. The `UDPDatagramConnection` API provides for low overhead but unreliable delivery of datagrams and has certain device and network caveats. The `CommConnection` API defines operations for accessing serial ports on a device if they are available.

It must be kept in mind that there are many mobile devices that do not provide support for sockets, and therefore the availability of the networking protocols will vary from one device to another. The networks provided by different network operators may also have limitations in the support of specific networking protocols. Also, access to some of the protocols may be prohibited because of the security policies of the device or the network. Close cooperation with the device manufacturer and the network operators' developer programs will be helpful in understanding how to use these APIs and how they may be enabled in particular networks.

Chapter 16. Secure Networking



Many mobile applications, especially those intended for mobile commerce, need secure transport protocols in order to ensure the integrity and confidentiality of transactions that are initiated by the applications. Secure protocols such as Secure Sockets Layer (SSL) and Transport Layer Security (TLS) have been in common use on the Internet for some time. The MIDP Specification version 2.0 now brings secure networking protocols also to the developers of J2ME applications.

The MIDP Specification version 2.0 defines two secure networking interfaces: `HttpsConnection` and `SecureConnection`. The `HttpsConnection` interface extends the interface `HttpConnection`, and the `SecureConnection` interface extends the interface `SocketConnection`. When a MIDlet opens an HTTPS connection or a secure socket connection, the client device and the server establish a secure link by negotiating the secure protocol and cipher suite, and by exchanging credentials.

Because the client and server components of mobile applications are usually designed and deployed together to satisfy a particular business need, the server-side infrastructure can define the confidentiality and integrity requirements. The client-side implementations needs to verify only that the client is communicating with the correct site and protocols.

Secure networking protocols supported by MIDP 2.0 include SSL and TLS as well as the Wireless Application Protocol (WAP) secure protocol WTLS. Each protocol places requirements on certificates used for secure networking, as described in [Section 16.4](#), "MIDP X.509 Certificate Profile."

[\[Team LiB \]](#)

[!\[\]\(a4263b17173e81f2c06ea852e4fde19a_img.jpg\) PREVIOUS](#) [!\[\]\(8f08ebb58cf5340686512b635ad80c1e_img.jpg\) NEXT](#) 

16.1 Checking the Security Properties of a Connection

The device and the server negotiate the secure connection parameters based on mutually agreeable secure suites. After the connection has been established, the application can retrieve a `SecurityInfo` object and verify the integrity and server credentials of a negotiated connection. A MIDlet using an HTTPS or secure connection gets the object by calling the `getSecurityInfo` method on the open connection. If the MIDlet's connection is not secure enough, it can report an error and close the connection.

The application can use the `SecurityInfo` object to get the protocol name, protocol version, cipher suite, and certificate for a secure network connection. [Table 16.1](#) lists the names and versions of the protocols available by calling the `getProtocolName` and `getProtocolVersion` methods of the `SecurityInfo` interface.

Table 16.1. Secure protocols and version numbers

Secure Protocol	Protocol Name	Version
SSL V3	"SSL"	"3.0"
TSL 1.0	"TLS"	"3.1"
WTLS (WAP-199)	"WTLS"	"1"
WAP TLS Profile and Tunneling Specification	"TLS"	"3.1"

The `getCipherSuite` method of the `SecurityInfo` interface returns the name of the cipher suite used for the connection. The name is usually one of the standard ciphers listed in the `CipherSuite` column of the `CipherSuite` definitions table in Appendix C of RFC 2246 [[reference 4](#)]. Non-TLS implementations should select the cipher suite name according to the key exchange, cipher, and hash combination used to establish the connection. This will ensure that equivalent cipher suites have the same name regardless of protocol. If the `getCipherSuite` method returns a non-null name that is not in Appendix C of RFC 2246 [[reference 4](#)], the cipher suite's contents are specific to the device and server.

16.1.1 Server Certificate

An application calls the `getServerCertificate` method of the `SecurityInfo` object to get the server's certificate. A `Certificate` object contains a subject, issuer, type, version, serial number, signing algorithm, dates of valid use, and serial number.

The subject is the entity that vouches for the identity of the server; the issuer is the distinguished name of the entity that vouches for the identity of the subject. An application gets the subject from the `getSubject` method of the `Certificate` object and the issuer from the `getIssuer` method. The `getIssuer` method never returns null. The subject and issuer strings are formatted using the rules for the printable representation for X.509 distinguished names given in [Section 16.1.2](#), "Printable Representation for X.509 Distinguished Names."

[\[Team LiB \]](#)

[!\[\]\(22e4f0e2f6f889ffa0cdf99995f93aca_img.jpg\) PREVIOUS](#) [!\[\]\(dc80d8e5a7d80b13c5b2606f3fa827cc_img.jpg\) NEXT](#) 

16.2 HttpsConnection

HTTPS is the secure version of the HTTP protocol [[reference 9](#)]. (See [Section 15.3](#), "The `HttpConnection` Interface," for more information.) The MIDP Specification version 2.0 requires a secure HTTP connection to comply with one or more of the following specifications:

- HTTP over TLS: documented in both RFC 2818 [[reference 11](#)] and TLS Protocol Version 1.0 as specified in RFC 2246 [[reference 4](#)].
- SSL V3: specified in The SSL Protocol Version 3.0 [[reference 12](#)].
- WTLS: specified in the Wireless Application Protocol Wireless Transport Layer Security document WAP-199 [[reference 13](#)].
- WAP TLS Profile and Tunneling Specification WAP-219-TLS [[reference 14](#)].

The `HttpsConnection` interface extends the interface `HttpConnection` to define MIDP HTTPS API. The `HttpsConnection` interface has the same methods and behaviors as its superinterface, and adds information about the security parameters of the connection. An application accesses an `HttpsConnection` by calling the `Connector.open` method. The application must supply a URI with the scheme "https:". RFC 2818 [[reference 11](#)] defines the scheme.

A secure link must be established so the request headers can be sent securely. The secure link may be established as early as in the invocation of the `Connector.open` method. It may also be established when the application opens the connection's input or output stream or calls methods that cause a transition to the connected state. (See [Section 15.3](#), "The `HttpConnection` Interface," for information on these methods.) Any method that can cause a secure link to be established may experience security-related failures; such methods throw a `CertificateException` to indicate these failures.

16.2.1 Security of `HttpsConnection`

Access to the secure HTTP connection may be restricted to prevent unauthorized transmission or reception of data. Trusted MIDlet suites needing to use HTTPS must request the `javax.microedition.io.Connector.https` permission as described in "[Trusted MIDlet Suite Security Model](#)" on page 308.

The device may try to authorize the MIDlet when the MIDlet calls the `Connector.open` method with a valid HTTPS connection string. If the authorization fails and the MIDlet is not allowed to use the HTTPS API, the `Connector.open` method throws a `SecurityException`. The device might also check the MIDlet's permission when the MIDlet calls the `openInputStream`, `openDataInputStream`, `openOutputStream`, and `openDataOutputStream` methods.

16.2.2 Example

[\[Team LiB \]](#)

[!\[\]\(8c4347170d053e4fec3ad1de801736ab_img.jpg\) PREVIOUS](#) [!\[\]\(05142ce97d746447b2acb5a8a81b953a_img.jpg\) NEXT](#) 

16.3 SecureConnection

The SecureConnection interface defines the API for a secure socket connection. Interface SecureConnection, like its superinterface SocketConnection, allows a client to connect to a server and has the added benefit of a secure transport. The MIDP Specification version 2.0 requires that a SecureConnection comply with one or more of the following specifications:

- TLS Protocol Version 1.0: specified in RFC 2246 [[reference 4](#)].
- SSL V3: specified in The SSL Protocol Version 3.0 [[reference 12](#)].
- WAP(TM) TLS Profile and Tunneling Specification: specified in WAP-219-TLS-20010411-a [[reference 14](#)].

There is no support for the complementary server-side secure connection in the MIDP Specification version 2.0.

An application opens a secure connection on a target host by calling the Connector.open method. The application must supply a GCF string with the scheme ssl, a host, and a port number. The host may be a fully qualified host name or an IPv4 number. For example, "ssl://host.com:79" defines a target socket on the host.com system at port 79. RFC 1900 [[reference 2](#)] recommends the use of names rather than IP numbers for best results in the event of IP number reassignment. Some networks and devices, however, may support only IP addresses since DNS name resolution may introduce additional latency and costs.

Unlike an HTTPS connection, the Connector.open method always opens the secure connection. If the secure connection cannot be established due to errors related to certificates, the Connector.open method throws a CertificateException.

16.3.1 Security of SecureConnection

Access to secure connections may be restricted to prevent unauthorized transmission or reception of data. Trusted MIDlet suites needing to use an SSL/TLS connection must request the javax.microedition.io.Connector.ssl permission as described in [Section 18.3](#), "Trusted MIDlet Suite Security Model."

The device may try to authorize the MIDlet when the MIDlet calls the Connector.open method with a valid connection string. If the authorization fails and the MIDlet is not allowed to use the secure socket API, the Connector.open method throws a SecurityException. The device might also check the MIDlet's permission when the MIDlet calls the openInputStream, openDataInputStream, openOutputStream, and openDataOutputStream methods.

16.3.2 Example

[\[Team LiB \]](#)

[!\[\]\(4e9e071b1582a4a7aec52b54bf0b7ae0_img.jpg\) PREVIOUS](#) [!\[\]\(26f0a0fa22ee9bf9d4e3aee31499eb88_img.jpg\) NEXT](#) 

16.4 MIDP X.509 Certificate Profile

MIDP 2.0 devices are expected to use standard Internet and wireless protocols and techniques for their transport and security. The protocols and techniques are based on the following Internet standards for public key cryptography:

- WAP Certificate Profile Specification WAP-211-WAPCert-20010522-a [[reference 15](#)] (WAPCert).
 - Devices must conform to all mandatory requirements in WAPCert, Appendix C, and should conform to all of its optional requirements except:
 - WAPCert [Section 6.2](#), User Certificates for Authentication.
 - WAPCert Section 6.3, User Certificates for Digital Signatures.
- RFC 2437 ?PKCS #1 RSA Encryption Version 2.0 [[reference 6](#)].
- RFC 2459 ?Internet X.509 Public Key Infrastructure [[reference 7](#)].
 - MIDP 2.0 implementations must support X.509 Certificates and are permitted to support other certificate formats. RFC 2459 [[reference 7](#)] contains sections that are not relevant to a MIDP 2.0 implementation, however. The WAP Certificate Profile does not mention these functions. Implementations should exclude:
 - Requirements from Paragraphs 4 of [Section 4.2](#) ?Standard Certificate Extensions. A MIDP 2.0 implementation does not need to recognize extensions that must or may be critical, including certificate policies, name constraints, and policy constraints.
 - [Section 6.2](#) Extending Path Validation. Support for Policy Certificate Authority or policy attributes is not required.

16.4.1 Certificate Extensions

A MIDP implementation must consider a version 1 X.509 certificate equivalent to a version 3 certificate with no extensions.

A MIDP 2.0 device must recognize key usage (see RFC 2459 [[reference 7](#)] section 4.2.1.3) and basic constraints (see RFC 2459 [[reference 7](#)] section 4.2.1.10). It must be able to process certificates with unknown distinguished name attributes and unknown non-critical certificate extensions. It must also recognize the serialNumber attribute defined by WAPCert [[reference 15](#)] in distinguished names for Issuer and Subject.

16.5 New for MIDP 2.0



The secure networking functionality described in this chapter is new in MIDP 2.0. The secure networking operations ensure the integrity and privacy of communications using standard secure network protocols. Interface HttpsConnection provides the application developer with the familiar popular network programming protocol HTTPS that integrates easily with the existing network servers and services. The SecureConnection interface extends the flexibility of sockets to the network infrastructure built on the reliable and robust SSL and TLS protocols. Both secure connection types make it easy to write robust and secure network applications.

Chapter 17. Event-Driven Application Launch



The MIDP Specification version 2.0 defines an option to automatically launch a MIDlet when a notification is received. Notifications can be network- or alarm-based. Network notifications, also known as push notifications, can occur when the device receives an incoming network connection for one of its MIDlets. Alarm-based notifications can occur at a time set by a MIDlet.

Automatic launching is not a required device feature. Devices can support both types of notifications, only one type, or neither.

Notifications can cause MIDlets to be launched without direct user intervention; users could be unaware of the activity. For example, if a device launches a MIDlet at a preset time, the user may not be present or paying attention.

A number of security concerns arise from the potential to launch applications without direct user intervention. Security permissions and policy provide safeguards, however. They ensure that MIDlets can perform only activities allowed by the user. (MIDlets, whether launched automatically or at the request of the user, are always subject to the security policy of the device and any permissions the user has granted. See [Section 18.3](#), "Trusted MIDlet Suite Security Model," for more information.) For example, a user can require the device to ask permission before automatically launching a MIDlet.

17.1 Alarm-Based MIDlet Launch

It can often be useful for a MIDlet to be launched at a preset time to provide information to the user in a timely fashion. For example, a MIDlet that sells movie tickets might set an alarm to warn the ticket-holding user about the impending start time of the movie.

To use alarm-based notification, a MIDlet calls the PushRegistry.registerAlarm method. The MIDlet must supply the name of the MIDlet to launch (it must be a MIDlet in the same MIDlet suite) and a launch time. The named MIDlet must have been registered in the JAD file or the JAR manifest with a MIDlet-n attribute. Providing a launch time of zero removes any alarm.

The push registry supports one alarm per MIDlet. If a registration occurs for a MIDlet that already has an alarm set, the new registration is used. The PushRegistry.registerAlarm method returns the old registration time in milliseconds since January 1, 1970, 00:00:00 GMT. Note that the time returned could be in the past. It returns zero for the MIDlet's first alarm registration.

Errors may occur during alarm registration. If the runtime system does not support alarm-based application launch, the registerAlarm method throws a ConnectionNotFoundException. If the MIDlet class name provided is null or is not in a MIDlet-n attribute in the MIDlet suite's JAR manifest or JAD file, the registerAlarm method throws a ClassNotFoundException. If the MIDlet does not have permission to register an alarm, the registerAlarm method throws a SecurityException. (See [Section 17.5](#), "Security of the Push Registry," for more information.)

A device could be unable to launch the MIDlet for the alarm. For example, a MIDlet cannot be launched if the device is off at the wake-up time. Similarly, the MIDlet cannot be launched if it is running at the wake-up time. As a result, MIDlets should be written to check the current time against the scheduled wake-up time. They can use class java.util.TimerTask to manage time-based events while they are running. (See [Section 20.1](#), "Timer Support," for more information.)

As a general rule, do not use the alarm to periodically launch a MIDlet to check for new information. Polling is an inefficient use of phone and network resources. It can run down the battery if it activates the device too often and can incur extra network-usage costs, while providing little or no value to the consumer. Instead, use push notification: have the network service send messages to the MIDlet when new information should be presented to the user.

If push notification cannot be used (for example, if the device does not support it), MIDlets that use alarm-based notification should be written to collapse or compress any duplicate events for MIDlets in the suite. This will help to avoid the usability problems associated with starting the MIDlet too frequently.

[\[Team LiB \]](#)

[!\[\]\(fa68265322926688c1b38a4aadac9a78_img.jpg\) PREVIOUS](#) [!\[\]\(a88b1cb3cef69d8cc149cd8d8f48e417_img.jpg\) NEXT](#) 

17.2 Network-Based MIDlet Launch

Devices can use protocols from the Generic Connection Framework (GCF) for their push functionality. (For more information, refer to [Chapter 15](#), "MIDP Networking and Serial Communications.") Launching a MIDlet when a network connection is received is an efficient mechanism to alert users to time-critical information.

Not all protocols are appropriate for use as push connections. Even if a device supports a protocol for inbound connections, the device is not required to use that protocol in its push mechanism. For example, a device might support server socket connections in a MIDlet, but might not support launching a MIDlet in response to receiving a server socket connection.

The three tasks that enable a MIDlet to be started to handle a push notification are registration, listening, and processing of the connection. During the registration phase, the system is given the expected network connection, the MIDlet to run, and a source-address filter. Listening is done by the Application Management Software (AMS). (See [Section 19.2](#), "MIDP System Software," for more information.) The AMS listens for the registered connections associated with MIDlets that are not running. Processing occurs when a registered connection is made: the AMS will run the corresponding MIDlet, and the MIDlet then assumes control of its connections. The steps for registration, listening, and processing are described in more detail below.

MIDlets use the methods of the javax.microedition.io.PushRegistry class to find and open connections. It uses the normal GCF classes to read, write, and close the connection.

When the MIDlet suite is active it is responsible for handling incoming connections. When the MIDlet suite is destroyed, AMS resumes listening for registered connections.

17.2.1 Registering a MIDlet to Be Launched

MIDlet suites must register for push notifications. They can register either statically or dynamically. Static registration is done by adding attributes to the application descriptor, the JAR manifest, or both. It is called static registration because it uses information that is present at installation time. Static registrations remain valid until the MIDlet suite is removed from the device. Dynamic registration is done with the methods of the PushRegistry class. It is called dynamic registration because MIDlets can enable and disable push notifications as needed. They can also modify registrations as needed.

If a MIDlet requires a particular connection in order to function correctly and will not need to change that connection while the MIDlet is on the device, the application developer should use static registration. Two MIDlet suites on a device cannot use the same connection. If two MIDlet suites have a push connection in common, they cannot be on the device at the same time and still function correctly. The device checks for connection conflicts at installation. If all the static push declarations can be fulfilled, then installation proceeds. If not, the user is notified that there are conflicts, and the MIDlet suite is not installed. (Refer to [Chapter 19](#), "MIDlet Deployment," for connection-conflict errors.) The end-user typically uninstalls one MIDlet in order to successfully install the other.

If a MIDlet can be flexible in its push connections, the application developer should use dynamic registration. That is, dynamic registration is for MIDlets that can use (but do not require) a connection, require a connection for only a

17.3 Listening and Launching

The Application Management Software (AMS) listens for registered connections and manages the launch of and transition to the appropriate MIDlet. It and the other software on the device provide a seamless integration of native and MIDP applications. The AMS handles the interactions among incoming connections, interrupting the user, starting and stopping applications, and handing off connections to MIDlets. It also enforces the device's security policy and constraints.

For example, if a native application is running when a notification arrives, the AMS must interrupt it. It must give the user control over the interruption (for example, user settings determine whether the interruption is to launch the new MIDlet or to get permission to launch it) and any transition to the MIDlet that responds to the network event. To make the transition, the AMS stops or pauses the running application. It and the other software on the device must work together to make the consumer experience smooth and natural.

The AMS would do similar things if a MIDlet was running when a notification arrives. It would call either the pauseApp method or the destroyApp method to change to the MIDlet that responds to the connection. The MIDlet life cycle defines the duties of the interrupted MIDlet. (Refer to "[MIDlet States](#)" on page 82.)

The implementation handles incoming connections in a way that allows them to be held and passed on to the MIDlet when it is started. The requirements for buffering of connections and messages are protocol-specific. There is a general requirement, though, that if a device buffers data, it must provide the buffered information to the MIDlet. The device should supply any buffered data when the launched MIDlet opens the related connection.

When a device supports datagram connections for push notification, it must buffer at least the datagram that caused the startup of the MIDlet. It must also guarantee that when it starts a MIDlet in response to the incoming datagram, the buffered data will be available to the MIDlet when the MIDlet opens the UDPDatagramConnection.

The requirement for devices that support socket connections for push notification is different. The device must guarantee that when it starts a MIDlet in response to an incoming socket connection, the MIDlet can accept the connection by opening the ServerSocketConnection, provided that the connection has not timed out.

17.4 Handling Connections after Launch

The MIDlet is responsible for handling incoming connections once it has been started. When a MIDlet associated with a push connection starts, it should first check why it was started. That is, it should get the list of available connections with input available. (See [Section 17.2.3](#), "Mechanics of Dynamic Push Registration," for information on the listConnections method.) If there are such connections, the MIDlet can assume it was started to handle them.

The AMS holds onto the incoming connections until the MIDlet opens them. (This is known as handing off the connections to the MIDlet.) The MIDlet opens a connection by passing a connection string returned by the listConnections method directly to the Connector.open method.

To prevent data loss, a MIDlet is responsible for all I/O operations on the connection from the time it calls the Connector.open method until it closes the connection. Application developers should avoid closing the connection too early. If the connection is closed but the MIDlet is still running, neither the AMS nor the application will be listening for push notifications. Inbound data could be lost if the application closes the connection before all data has been received.

Application developers should also use a separate thread to perform input and output (I/O) operations and avoid blocking (and possibly deadlocking against) interactive user operations.

A push application should behave in a predictable manner when handling asynchronous data it obtains through the push mechanism. A well-behaved application should inform the user that data has been processed. While it is possible to write applications that do not use any user visible interfaces, launching an application that only performs a background function could be a confusing user experience.

17.5 Security of the Push Registry

The PushRegistry class is protected using the security framework and permissions. A MIDlet suite must have the javax.microedition.io.PushRegistry permission to register an alarm based launch and to statically or dynamically register for network notifications. Push registry permissions are also used to determine whether the user needs to be prompted before launching a MIDlet suite in response to a network or alarm-based notification. The security framework defines the general behavior for user permissions with the interaction modes of oneshot, session, and blanket. Their behavior is specialized for the PushRegistry and AMS, as described in the following list:

-
- Oneshot: prompt the user before launching a MIDlet suite in response to a network or alarm-based notification, and for each push registry request (such as registering an alarm or connection).
-
- Session: prompt the user before launching a MIDlet suite in response to a network or alarm-based notification, or before the first push registry request (such as registering an alarm or connection) after each launch. The user is not prompted on subsequent uses of the push registry by the MIDlet.
-
- Blanket: prompt the user once when the MIDlet is being installed, before the first time the MIDlet suite is launched in response to a network or alarm-based notification, and before it uses the push registry.

In addition to needing permission to use the push registry, the MIDlet must also have permission to use server protocols. That is, the push mechanism uses protocols in which the device is acting as the server; it can accept connections from other entities on the network. To use the push mechanisms and accept connections, the MIDlet suite needs permission to use the protocols that it will use when it acts as a server. For example, a chat program that can be started in response to a socket connection notification would need to have the javax.microedition.io.Connector.serversocket permission. The chat program might use the following attributes in the manifest to request the permissions and static registrations it requires:

[\[View full width\]](#)

```
MIDlet-Push-1: socket://:79, com.sun.example.SampleChat, *
MIDlet-Permissions: javax.microedition.io.PushRegistry, javax.microedition.io.Connector.
➥ serversocket
```

[\[Team LiB \]](#)

[!\[\]\(8a3dd3e8437de609e7ea50c8d2699553_img.jpg\) PREVIOUS](#) [!\[\]\(f779c183d7d365f1a192cd55f09a6bed_img.jpg\) NEXT](#) 

17.6 Sample Usage Scenarios

This section provides two examples of MIDlets that are launched in response to a network notification. The first example is a Chat application that uses a well-known port for communication. The second example is a Ping application that dynamically allocates a port the first time it is started.

17.6.1 Sample Chat Application

The Chat application uses push to receive a message on a well-known port. In order to concentrate on the push functionality, it does not show any corresponding MIDlet that would enable the user to reply to the message.

Note

This sample is appropriate for bursts of datagrams because it loops on the connection, processing received messages.

The JAD file for the Chat application's MIDlet suite includes a static push connection registration shown in boldface. It also includes an indication that this MIDlet requires permission to use a datagram connection for inbound push messages. The following example code shows the JAD file:

[\[View full width\]](#)

```
MIDlet-Name: SunNetwork - Chat Demo
MIDlet-Version: 1.0
MIDlet-Vendor: Sun Microsystems, Inc.
MIDlet-Description: Network demonstration programs for MIDP
MicroEdition-Profile: MIDP-2.0
MicroEdition-Configuration: CLDC-1.0
MIDlet-1: InstantMessage, /icons/Chat.png, example.chat.SampleChat
MIDlet-Push-1: datagram://:79, example.chat.SampleChat, *
MIDlet-Permissions: javax.microedition.io.PushRegistry, javax.microedition.io.Connector.
➥datagramreceiver
```

The following example code shows the Chat MIDlet. The startApp method launches a thread to handle the incoming data. (An inner class implements the Runnable interface.) Using a separate thread to handle network I/O is the recommended practice. It avoids conflicts between blocking I/O operations and user-interaction events. The thread receives messages until the MIDlet is destroyed.

```
public class SampleChat extends MIDlet {
    // Current inbound message connection
    DatagramConnection conn;

    // Flag to terminate the message reading thread
    boolean done_reading;

    public void startApp() {
        // List of active connections
        String connections[];

        // Check to see if this session was started due to
        // (initial) connection establishment
```


17.7 New for MIDP 2.0



Event-driven application launch is a new feature in MIDP 2.0. This feature makes it possible to create applications that can be launched at a preset time or that can respond to events generated by network services.

The ability to define applications that will be launched on a specific network protocol event enables the creation of new kinds of services that can monitor information in the network and can notify a MIDlet suite automatically when the information changes.

The event-driven launch of MIDlets is subject to the security policy of a device, and this feature can be enabled and disabled at the discretion of the user.

[\[Team LiB \]](#)

[!\[\]\(43ac5e5c84f8ef971b819581af9c4682_img.jpg\) PREVIOUS](#) [!\[\]\(d05f842aa9190cf330109c27625c6387_img.jpg\) NEXT](#) 

Chapter 18. Security for MIDlet Suites

Security in mobile devices presents many interesting and significant challenges. Mobile devices must enable users to take advantage of the applications and services provided on the Internet while still protecting them and their data.

Many parties are interested in the security of mobile devices. Network operators have a strong interest in ensuring the integrity of the data that is sent over their networks so they can provide friendly, useful, lucrative, and reliable services to their customers. Mobile device manufacturers want to create usable, useful devices that provide a compelling experience for consumers. Software developers want to tap into the large and vibrant consumer market to sell games, entertainment, and productivity applications. Consumers want a device that is simple, fun, convenient, and easy to use.

Consumers do not want to be inconvenienced, and they are generally discouraged by the presence of security mechanisms that make the devices more difficult to operate. The big challenge is to keep consumer confidence and convenience in balance. If the consumers do not believe that they can use mobile devices easily, safely, and reliably to access services, they will not use them.

To be confident users of a device, consumers need to know their responsibilities and be aware of the limits on their liabilities. They need to be confident that they are using reliable technology that will not fail or compromise their privacy or security. They need to know that the companies they deal with have confidence in the device and network, and that the transactions that they initiate are safe. They need to have confidence in knowing that their personal information will not be used in unanticipated ways. They need to know when their use of the device is costing them money. These opportunities and constraints come together in the MIDP 2.0 security model.

The MIDP Specification version 2.0 extends the sandbox security model used by MIDP 1.0. It introduces a security model that can safeguard the functions of the device that can expose the user to risk, such as the risk of incurring costs associated with wireless network use or the risk to personal privacy. The MIDP 2.0 security model is also extensible to optional APIs, such as SMS.

The MIDP 2.0 security model tries to reconcile the competing goals of security and usability. Security concerns usually require frequent checking and rechecking that an action is allowed. Improving the usability of applications usually means reducing the number of steps and choices the user needs to make. Often the users first get annoyed and then habituated to repetitive prompts, and become accustomed to responding without really reading or making a conscious choice each time. This reduces or eliminates the effectiveness of security mechanisms that ask the user to confirm each and every access of protected functions.

The MIDP 2.0 security model enables the device to enforce security policy based both on the authentication of the MIDlet suite (verification that it is genuine and has not been tampered with) and on the user choices. The device uses authentication information to define which permissions are allowed and which are delegated to the user. Only when the access permissions are delegated to the user must the implementation protect the resource by requiring an explicit confirmation from the user. This can reduce the number of prompts a user might receive, improving the effectiveness of the security mechanism.

18.1 Assumptions

The following assumptions contributed to the design of the security mechanism and ensured ample flexibility in its implementation:

- MIDlets do not need to be aware of the security policy except for security exceptions that may occur when using APIs.
- A MIDlet suite is subject to a single protection domain and its permissible actions. (See [Section 18.3.3, "Protection Domain,"](#) for an explanation of protection domains.)
- The internal representation of protection domains and permissions is implementation specific. (See [Section 18.3.1, "Permissions,"](#) for an explanation of permissions.)
- The user interface for presenting configuration settings and the results of authentication attempts to the user is implementation-dependent and outside the scope of the MIDP Specification version 2.0.
- The device must protect its security policy and its information on protection domains so that they are safe from viewing or modification except by authorized parties.
- If the security policy for a device is static and disallows use of some functions of the security framework, then the implementation of unused and inaccessible security functions may be removed.
- Security policy allows an implementation to restrict access but must not be used to avoid implementing security-sensitive functionality. For example, unimplemented protocols under the Generic Connection framework must throw a `ConnectionNotFoundException`, not a security-related exception.

18.2 Sandbox for Untrusted MIDlet Suites

The MIDP Specification version 1.0 constrained MIDlet suites to operate in a sandbox that prevented access to sensitive APIs and functions of the device. This model was useful for many applications, and is included and formalized in the MIDP Specification version 2.0.

A MIDlet suite compliant with MIDP 1.0 must be able to run on a MIDP 2.0 device as an untrusted MIDlet suite. An untrusted MIDlet suite is a MIDlet suite for which the origin and the integrity of the JAR cannot be reliably determined by the device. Untrusted MIDlet suites execute in a restricted environment where access to the protected APIs or functions is either not allowed or allowed only with explicit user permission. The restricted environment is the untrusted domain.

The untrusted domain must allow, with explicit confirmation by the user, access to the protected APIs and functions in [Table 18.1](#). Other restricted APIs on the device may be available to untrusted MIDlet suites depending on the security policy of the device.

Table 18.1. APIs for which untrusted MIDlet suites require confirmation

API	Protocol
javax.microedition.io.HttpConnection	HTTP
javax.microedition.io.HttpsConnection	HTTPS

[\[Team LiB \]](#)

[!\[\]\(bd20f95b322669fb975a9574b885f884_img.jpg\) PREVIOUS](#) [!\[\]\(09b6cd0548e0cf09dded039f88d7b2b5_img.jpg\) NEXT](#) 

18.3 Trusted MIDlet Suite Security Model

Today, many functions are provided by applications built into the device. Trust in the applications comes from trust in the device manufacturer and network operator who are clearly identified.

Consumers also need to know and trust downloadable applications and their sources. The MIDP Specification version 2.0 introduces the concept of trusted MIDlet suites, which provides a level of authentication that allows the users to have such confidence. A trusted MIDlet suite is one for which the source, authenticity, and integrity of the JAR can be reliably determined by the device.

The procedure for determining whether a MIDlet suite is trusted is device-specific. Some devices might trust only MIDlet suites obtained from certain servers. Other devices might support only untrusted MIDlet suites. It is also possible for a device to authenticate MIDlet suites with the same strong security mechanisms that web sites rely on everyday: the Public Key Infrastructure (PKI). See [Section 18.5](#), "Establishing Trust for MIDlet Suites by Using X.509 PKI," for more information.

Once the device trusts the MIDlet suite, it can grant the MIDlets in the suite permission to use protected functions. Determining whether to grant a requested permission is known as authorization.

18.3.1 Permissions

Devices use permissions to protect access to APIs or functions that require authorization before being invoked. The permissions described in this section refer only to those APIs and functions that need security protection. They do not apply to APIs that do not require authorization and can be accessed by both trusted and untrusted MIDlet suites. ([Section 18.4](#), "APIs That Are Not Security Sensitive," covers the non-security-sensitive APIs.)

A MIDP implementation typically gives trusted applications more access to security-sensitive APIs than untrusted MIDlets. To ensure that access is granted appropriately, the implementation checks permissions prior to each invocation of a protected function. MIDlet suites must have their permission request granted before the implementation provides the function.

[Table 18.2](#) summarizes the permissions that apply to the restricted APIs.

Table 18.2. Summary of Permissions defined by the MIDP Specification

Permission	Restricted API
javax.microedition.io.Connector.http	javax.microedition.io.HttpConnection
javax.microedition.io.Connector.https	javax.microedition.io.HttpsConnection

18.4 APIs That Are Not Security Sensitive

APIs and functions that are not security sensitive are accessible by all MIDlet suites. These APIs are listed in [Table 18.3](#).

Table 18.3. APIs freely permitted for all MIDlet suites

API	Description
javax.microedition.rms	RMS APIs
javax.microedition.midlet	MIDlet Life-cycle APIs
javax.microedition.lcdui	User Interface APIs
javax.microedition.lcdui.game	The Game APIs
javax.microedition.media javax.microedition.media.control	The multimedia APIs for playback of sound

[\[Team LiB \]](#)

[!\[\]\(9613852665ad78e04ab99a1a69e741a7_img.jpg\) PREVIOUS](#) [!\[\]\(ef580aba806fa57c042dedbce78bb658_img.jpg\) NEXT](#) 

18.5 Establishing Trust for MIDlet Suites by Using X.509 PKI

The mechanisms for signing and authenticating MIDlet suites are based on Internet standards for public key cryptography RFC 2437 [[reference 6](#)], RFC 2459 [[reference 7](#)], RFC 2560 [[reference 8](#)], and WAPCERT [[reference 15](#)].

A developer can put a digital signature of the MIDlet suite's JAR and a certificate in the MIDlet suite's application descriptor. A digital signature is a piece of data, created with the private key of a private-public key pair, that ensures the JAR has not been tampered with.

A certificate contains the corresponding public key that can be used to check the digital signature in the JAR and the signature of an entity that vouches that the public key in the certificate is genuine. The application descriptor can also have another certificate that contains the public key of the vouching entity and the signature of the entity that vouches for it. This can continue until there is a certificate for a well-known and trusted entity that does not need another entity to vouch for it; it vouches for itself. Such an entity is called a certificate authority (CA), and the certificate is said to be self-signed. Self-signed certificates are also called root certificates. Root certificates reside on the device, not in the application descriptor. A series of certificates that begins with the JAR signer and ends with a CA is called a certificate chain.

Normally the development environment for MIDP applications will include tools to handle signing of a MIDlet suite and putting the appropriate certificates into the suite's application descriptor, so application developers should not have to understand these details. However, developers may still need to be aware of the protection domains on their target devices so that they can obtain the appropriate certificates.

When a user tries to install a signed MIDlet suite, the device authenticates the suite by verifying the signer certificates and JAR signature. Authentication enables the device to assign the MIDlet suite to a protection domain and appropriately authorize the MIDlets. Devices associate protection domains with CAs. For example, the device might assign the MIDlet suites that use the device manufacturer for their root certificate to one domain and the MIDlet suites that use the MIDP implementor for their root certificate to another. The root certificate associated with a protection domain is called the protection domain root certificate.

In the case where there is a trusted relationship (possibly bound by legal agreements), the owner of a protection domain root certificate may delegate signing MIDlet suites to a third party and, in some circumstances, the author of the MIDlet.

18.5.1 Signing a MIDlet Suite

The signer of a MIDlet suite might be the developer or an entity responsible for distributing, supporting, and perhaps billing for the use of the MIDlet suite. The signer is responsible for ensuring that the MIDlet suite's applications are not malicious and will not harm assets or capabilities. The signer must exercise due-diligence in checking the MIDlet suite before signing it.

The signer must have a private and public key pair. The signer uses the private key to sign the MIDlet suite, and provides the public key in an RSA X.509 certificate for the MIDlet suite's application descriptor. That certificate

18.6 Recommended Security Policy for GSM/UMTS Devices

The MIDP Specification defines the security model and mechanisms to determine whether applications can be trusted, and for them to request specific permissions. The specification does not mandate a specific policy or how those mechanisms are used. The Recommended Security Policy for GSM/UMTS Devices (see the MIDP Specification version 2.0) describes a policy recommended by the GSM operators. The purpose of the policy is to allow consistent application behavior across GSM and UMTS devices and networks.

18.7 New for MIDP 2.0



The entire security model for MIDlet suites is new in MIDP 2.0. Unlike in MIDP 1.0, the MIDlets are no longer restricted to run only in a sandbox. Flexible mechanisms are now provided to establish a security policy for a MIDP device. The new model is extensible for use with J2ME optional packages and other APIs such as the Wireless Messaging API and the Mobile Media API. The new model can take advantage of the existing PKI infrastructure to ensure the integrity and authentication of the source of MIDlet suites. Through the protection domains, the new model enables the implementation of a variety of security policies that meet the needs of security-critical mobile applications and business models.

Chapter 19. MIDlet Deployment

Deploying MIDlets involves more than simply writing Java code using the CLDC and MIDP APIs. To transfer the MIDlet to a wireless device, it has to be packaged with the resources it requires, configured for the locale in which it will be used, and placed on a server that supports the over-the-air (OTA) protocol specification so that it can be downloaded by mobile devices. The first thing to understand is the concept of a MIDlet suite.

[\[Team LiB \]](#)

[!\[\]\(2350cffe4f5e24b1a8be233e2075a268_img.jpg\) PREVIOUS](#) [!\[\]\(1f314b57f06601659230d134ded865ff_img.jpg\) NEXT](#) 

19.1 MIDlet Suites

One of the central goals for the MIDP application model is to provide support for the controlled sharing of data and resources between multiple, possibly simultaneously running MIDlets. To accomplish this goal while retaining a secure environment, the MIDP Specification requires that any MIDlets needing to interact and share data beyond what is enabled with the RMS record sharing in MIDP 2.0 (refer to [Section 14.2.2](#), "Shared Record Stores"), the MIDlets must be placed into a single JAR. This collection of MIDlets encapsulated in a JAR is referred to as a MIDlet suite.

The MIDlets within a MIDlet suite share a common name space (for persistent storage), runtime object heap, and static fields in classes. In order to preserve the security and the original intent of the MIDlet suite provider, the MIDlets, classes, and individual files within the MIDlet suite cannot be installed, updated, or removed individually; they must be manipulated as a whole. In other words, the basic unit of application installation, updating, and removal in MIDP is a MIDlet suite.

A MIDlet suite can be characterized more precisely by its packaging and its runtime environment. These characteristics are discussed in more detail next.

19.1.1 MIDlet Suite Packaging

A MIDlet suite is encapsulated within a JAR.^[1] A MIDlet suite provider is responsible for creating a JAR that includes the appropriate components for the target user, device, network, and locale. For example, since the CLDC Specification does not include the full internationalization and localization support provided by Java 2 Standard Edition, a MIDlet suite provider must tailor the JAR components to include the necessary additional resources (strings, images, and so forth) for a particular locale.

[1] JAR format specifications are available at <http://java.sun.com/products/jdk/1.4.1/docs/guide/jar/index.html>. Refer to the JDK JAR and manifest documentation for syntax and related details.

The contents of the MIDlet suite's JAR include the following components:

- the class files implementing the MIDlet(s),
- any resource files used by the MIDlet(s)? for example, icon or image files, and so forth, and
- a manifest describing the JAR contents.

All the files needed by the MIDlet(s) are placed in the JAR using the standard structure based on mapping the fully qualified class names to directory and file names within the JAR. Each period is converted to a forward slash, '/'. For class files, the .class extension is appended.

19.1.2 Application Descriptor

19.2 MIDP System Software

In the preceding sections and chapters, frequent references were made to the piece of software called the "system," "MIDP system," or the "execution environment." The MIDP Specification calls this software the application management software, or the AMS. In some documents, application management software is also referred to as the Java Application Manager, or the JAM. The two terms (AMS and JAM) are equivalent.

The application management software in a MIDP implementation consists of those pieces of software on the device that provide a framework in which MIDlet suites are installed, updated, removed, started, stopped, and in general, managed. Furthermore, the application management software provides MIDlets with the runtime environment discussed in [Section 19.1.4](#), "MIDlet Suite Execution Environment."

[\[Team LiB \]](#)

[!\[\]\(08527893d055e02779ecb163c9c36b9e_img.jpg\) PREVIOUS](#) [!\[\]\(2cca41fcb2fb66f1242aef59a355a830_img.jpg\) NEXT](#) 

19.3 Over-the-Air User-Initiated Provisioning

MIDP devices without installed MIDlets would not be all that useful. Granted, a device may provide a way to download MIDlets through serial cable or similar mechanism, which would enable the installation of MIDlets. However, one of the biggest advantages of recent devices is the availability of wireless data connectivity, whereby the device can connect with Internet servers and services to download MIDlets over-the-air (OTA). In this way, the user can have access to new MIDlets while in the airport, train station, at a friend's house, in a meeting, or generally anywhere where wireless coverage is available.

In the MIDP Specification version 2.0, the OTA download specification is based on the HTTP protocol.^[3] While the most common way a user will discover MIDlet suites for downloading will be through the WWW, WAP, or iMode browser resident on the device, other solutions are possible. For the specification and this book, we will use the more generic term discovery agent (DA) to indicate the application used for discovering and downloading a MIDlet suite.

[3] The term HTTP protocol is used generically in this chapter to mean HTTP and HTTPS, as well as the WAP protocol (WSP) with the http:// schema.

For the sake of simplicity in the rest of this discussion, we will ignore the intermediate wireless base stations, network nodes, and other network components, and consider the OTA transaction as a direct connection between the mobile device and a server.

A typical transaction is illustrated in the event transition diagram in [Figure 19.2](#), and it consists of the following steps:

1.

Using the discovery agent (DA), the user chooses a MIDlet to install. The MIDlet will be referenced by a URL that points to either an Java Application Descriptor File (JAD) or a JAR.

2.

If the URL references an application descriptor, the device downloads it and examines its contents as a preflight check. In this way it ensures that the device has suitable capabilities and resources to run the MIDlet before attempting the download of the JAR, which is typically much larger.

3.

If the URL that the user chose represents a JAR, or if the application descriptor preflight check was successful, the device downloads the JAR and installs it.

4.

The device sends a notification to the server indicating the installation status, whether successful or not.

Figure 19.2. OTA download sequence. *Italic items are optional.*



19.4 New for MIDP 2.0



The MIDP Specification version 2.0 defines precise guidelines for the deployment and over-the-air (OTA) provisioning of MIDP applications, based on the experience that was gained after releasing the MIDP Specification version 1.0 and publishing the recommended practice document for the OTA delivery of MIDP 1.0 applications.

The result of this work is a streamlined specification that can be implemented on a wide range of devices and networks. Modifications have been made to the MIDlet suite model to handle the downloading and upgrading of signed MIDlet suites, and to handle failures related to the registration of MIDlet suites for event-based launching (using the PushRegistry class.) Support for cookies to identify sessions across download requests has been removed because that feature was difficult to support on some networks. Session information can be maintained using URL rewriting instead. Notifications can be requested by a provisioning server to indicate successful or unsuccessful installations and deletions. Attributes for the application descriptor and manifest are specified to allow the MIDlet suite to request permissions needed by the security model. The deployment specification allows a wide range of device implementations and provisioning servers to interoperate smoothly.

Chapter 20. Additional MIDP APIs

In addition to the functional categories discussed elsewhere in this book, the MIDP Specification defines some additional APIs that do not fill their own chapters. Also, the MIDP Specification intentionally modifies, refines, or augments the behavior of certain functions that it inherits from CLDC.

This chapter introduces the MIDP Timer APIs. Furthermore, this chapter discusses the system properties and resource access mechanisms defined by the MIDP Specification, as well as explains how the MIDP Specification overrides the behavior of the `System.exit` method inherited from CLDC.

[\[Team LiB \]](#)

[!\[\]\(2c0f488574f9647fe79294204eab9500_img.jpg\) PREVIOUS](#) [!\[\]\(ff87b232db2fbb0647e779e527f535b2_img.jpg\) NEXT](#) 

20.1 Timer Support

MIDlets may need to delay or schedule activities to be performed at a later time. The MIDP Specification provides two mechanisms to enable this:

- Timers are used for time-based events while a MIDlet is running. For example, a timer used to change the frame being shown in an animation sequence. Timers are covered in this section.



Alarms provide a one-shot time for MIDlets to be launched if they are not already running. Automatic launching can be convenient, for example, to register the next appointment in a calendar. Alarm-based MIDlet launching is covered in [Section 17.1](#), "Alarm-Based MIDlet Launch."

The MIDP Specification provides two classes: Timer and TimerTask, that include functions for several types of timers. The general setup of a timer is illustrated next. In order to define a task to run, the programmer first defines a new timer task class by inheriting class TimerTask. The code to be executed by the timer is defined in the run method of the TimerTask subclass (see class MyTask below). In order to set up the actual timer, instances of class Timer and the new timer task class are created (objects myTimer and myTask in this example), and the schedule method of the timer object is then called with the task object and the appropriate time values as parameters. In this example, the timer will start executing in 10 milliseconds and will then repeat every 500 milliseconds.

```
class MyTask extends TimerTask {  
    public void run() {  
        ...  
    }  
}  
...  
myTimer = new Timer();  
myTask = new MyTask();  
myTimer.schedule(myTask, 10, 500);
```

There are two basic types of timers in MIDP: one-shot timers and repeating timers. The key difference between these two types of timers is the scheduling approach. A one-shot timer executes the specified task only once. A repeating timer continues executing the specified task repeatedly at a specific interval.

Timers can be used by a MIDlet in both the Active and Paused states, although use in the Paused state is discouraged.

20.1.1 Using One-Shot Timers

A one-shot timer can be set up to execute a task in two different ways. In the first approach, a java.util.Date object is passed into the schedule method of the Timer object. When the time specified by the Date object occurs, the timer task is run. If the Date is modified, the Timer object does not cancel the timer task; it is immediately triggered again.

20.2 System Properties

MIDP inherits the system property mechanism from CLDC (see [Section 5.2.10](#), "Property Support"). In addition to the properties provided by the CLDC, the MIDP Specification defines the following additional properties that can be retrieved by calling the method `java.lang.System.getProperty` (see [Table 20.1](#)).

The `microedition.locale` property is a String that consists of the language, country code, and variant separated by "-" (Unicode U+002D), for example, "fr-FR" or "en-US". Note that this is different from the J2SE definition for Locale printed strings where fields are separated by "_" (Unicode U+005F).

Table 20.1. System properties defined by MIDP

System Property	Description
<code>microedition.locale</code>	The current locale of the device; may be null.
<code>microedition.profiles</code>	A blank (Unicode U+0020) separated list of the J2ME profiles that this device supports. For MIDP 2.0 devices, this property MUST contain at least "MIDP-2.0".
 <code>microedition.commports</code>	A comma separated list of ports that can be combined with a "comm:" prefix as the URL string to be used to open a serial port connection (see Section 15.8 , "CommConnection").
<code>microedition.hostname</code>	The local hostname (if available).

This locale string can be used by a MIDlet to determine the locale for which the underlying device is configured.

The `microedition.profiles` property can be used by a MIDlet to ensure that the underlying platform supports the necessary profile(s) and profile version(s) that the application requires.

20.3 Application Resource Files

MIDP provides the capability of storing resources in the application's JAR that can be accessed by a MIDlet at run time. An example of such a resource might be a locale bundle, an icon, or a graphical image; however, any thing that can be accessed as an InputStream can also be stored. These resources are accessed using the method `getResourceAsStream(String name)` of class `java.lang.Class`.

The rules and the parameter string format for accessing resources inside a JAR file have been discussed in [Section 19.1.4](#), "MIDlet Suite Execution Environment." As a general rule, if the parameter string to the `getResourceAsStream` method begins with a '/', the search for the resource begins at the "root" of the JAR file; however, if it does not begin with a '/', the resource is searched for along a path relative to the class instance retrieving the resource.

20.4 Exiting a MIDlet

The CLDC Specification states that a call to the method `java.lang.System.exit` is effectively equivalent to calling the method `Runtime.getRuntime().exit`. This means that in a CLDC program, calling the `System.exit` method will exit the virtual machine. However, since MIDP defines its own application model, this behavior is not desirable. For instance, it is not desirable that a single MIDlet in a MIDlet suite, upon exiting, would shut down the entire virtual machine (and therefore shut down the other MIDlets as well.) Rather, a well-behaved MIDlet should exit by calling the `destroyApp` and `notifyDestroyed` methods.

In order to prevent MIDP applications from accidentally shutting down the entire virtual machine, the MIDP Specification redefines the behavior of the `system.exit` method. In MIDP, calling this method will always throw a `java.lang.SecurityException`.

[\[Team LiB \]](#)

[!\[\]\(a0c34e8af886b5f7ad053d94c1a15e97_img.jpg\) PREVIOUS](#) [!\[\]\(21939286a8313ee8c52a572263a3a0f4_img.jpg\) NEXT](#) 

Chapter 21. Summary

In the past few years, we have witnessed the beginning of an exciting new era in the history of computing. Wireless, handheld computing and communication devices, or mobile information devices (MIDs), have taken the role that desktop computers had earlier as the forefront of computing technology. As a result of this transition, computing and communication devices are becoming more and more mobile, personal, and ubiquitous. The number of mobile information devices has already surpassed the number of conventional personal computers, and this trend is likely to continue and strengthen in the future. The vast majority of people in the world will probably never use a traditional desktop computer. However, it is very likely that a very large percentage of them will eventually be using some kind of a mobile information device.

At the same time, the rapid emergence of the Internet is playing an increasingly visible role in the development and evolution of mobile information devices. People have become dependent on the information that is available on the Internet, and they will also want access to that information from mobile information devices. This will place much more emphasis on the ability to customize and personalize mobile information devices according to the needs of the individual users. Unlike in the past, when wireless devices typically came from the factory with a hard-coded feature set, the devices will become much more dependent on dynamically downloaded software. Consequently, there is a need for technologies that open up the mobile information devices for third-party software development, and make it possible to extend and customize the features of mobile information devices for Internet access and Internet-based services in a dynamic, secure fashion.

As discussed in this book, we believe that the Java programming language is ideally suited to become the standard application development language for wireless devices. After all, Java technology provides an unsurpassed combination of benefits for device manufacturers, wireless network operators, content providers, and individual application developers. These benefits include the dynamic delivery of interactive content, security, cross-platform compatibility, enhanced user experience, and the power of a modern object-oriented programming language with a very large established developer base.

In this book, we have summarized the recent advances in creating a portable, secure, small-footprint Java application development environment for small, connected devices. The initial work in this area started in early 1998 at Sun Microsystems Laboratories (Sun Labs) with the creation of a small, new Java execution engine, the K Virtual Machine (KVM). This work led to a number of collaborative research and development efforts with major consumer device manufacturers and other companies. Eventually, a number of standardization efforts were started to harmonize the key features and libraries across a wide variety of possible target devices.

This book has presented the overall architecture of Java 2 Platform, Micro Edition (J2ME), focusing specifically on two key J2ME standards: Connected, Limited Device Configuration (CLDC) version 1.1 and Mobile Information Device Profile (MIDP) version 2.0. Connected, Limited Device Configuration is intended to serve as a generic, "lowest common denominator" platform or building block that targets all kinds of small, connected devices that have at least 192 kilobytes of memory available for the Java environment and applications. Mobile Information Device Profile builds on top of CLDC and adds valuable application programming interfaces for a specific category of devices: wireless, mobile, two-way communication devices such as cellular telephones and two-way pagers.

A key characteristic of the CLDC and MIDP standards is the incremental and complementary nature of the platforms that they define. A central goal in creating the CLDC and MIDP standards was to avoid any conflicts with existing

[\[Team LiB \]](#)

[!\[\]\(4498ea06d630fb18b29bf1ac9fe56e1c_img.jpg\) PREVIOUS](#) [!\[\]\(981be0e003e7a93bf2ce519375c1071e_img.jpg\) NEXT](#) 

References

[reference 1] User Datagram Protocol. The Internet Engineering Task Force. August 1980. <<http://www.ietf.org/rfc/rfc768.txt>>

[reference 2] Renumbering Needs Work. The Internet Engineering Task Force. February 1996. <<http://www.ietf.org/rfc/rfc1900.txt>>

[reference 3] Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. The Internet Engineering Task Force. November 1996. <<http://www.ietf.org/rfc/rfc2045.txt>>

[reference 4] The TLS Protocol Version 1.0. The Internet Engineering Task Force. January 1999. <<http://www.ietf.org/rfc/rfc2246.txt>>

[reference 5] Uniform Resource Identifiers (URI): Generic Syntax. The Internet Engineering Task Force. August 1998. <<http://www.ietf.org/rfc/rfc2396.txt>>

[reference 6] PKCS #1: RSA Cryptography Specifications Version 2.0. The Internet Engineering Task Force. October 1998. <<http://www.ietf.org/rfc/rfc2437.txt>>

[reference 7] Internet X.509 Public Key Infrastructure Certificate and CRL Profile. The Internet Engineering Task Force. January 1999. <<http://www.ietf.org/rfc/rfc2459.txt>>

[reference 8] X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. The Internet Engineering Task Force. June 1999. <<http://www.ietf.org/rfc/rfc2560.txt>>

[reference 9] Hypertext Transfer Protocol ?HTTP/1.1. The Internet Engineering Task Force. June 1999. <<http://www.ietf.org/rfc/rfc2616.txt>>

[reference 10] HTTP Authentication: Basic and Digest Access Authentication. The Internet Engineering Task Force. June 1999. <<http://www.ietf.org/rfc/rfc2617.txt>>

[reference 11] HTTP Over TLS. The Internet Engineering Task Force. May 2000. <<http://www.ietf.org/rfc/rfc2818.txt>>

[reference 12] Freier, Alan O., Philip Karlton, Paul C. Kocher. The SSL Protocol Version 3.0. November 1996 <<http://home.netscape.com/eng/ssl3/draft302.txt>>

Appendix A. CLDC Application Programming Interface

This appendix contains the application programming interface documentation in Almanac format for the CLDC. For a description of this format, refer to "[Almanac Legend](#)" on page 356.

Full CLDC javadocs with detailed comments are available in the CLDC Specification (see "[Related Literature and Helpful Web Pages](#)" on page xxv) or as part of the CLDC reference implementation software that can be downloaded from Sun's web site (<http://www.sun.com/software/communitysource/j2me/cldc/>).

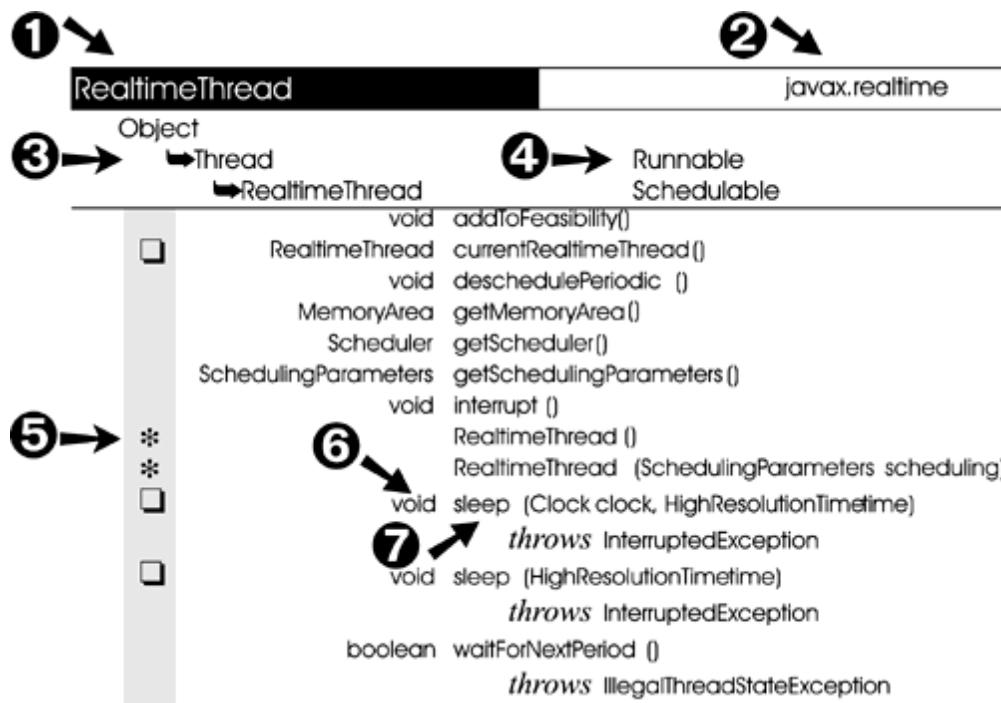
[\[Team LiB \]](#)

[!\[\]\(3fabc5daa534284a62133e7e08e4adaa_img.jpg\) PREVIOUS](#) [!\[\]\(d7bb9f8813cddba3b1bbaa7753f72543_img.jpg\) NEXT](#) 

Almanac Legend

The Almanac format presents all classes and interfaces in alphabetic order. Each class displays a list of its members in alphabetic order, mixing fields, methods, and constructors together.

The Almanac format used in this book is modeled after the style introduced by Patrick Chan in his excellent book, JavaTM Developers Almanac.



1.

The name of the class or interface. If the name refers to an interface, its name is printed in italics.

2.

The name of the package containing the class or interface.

3.

The inheritance chain of superclasses. A class is a subclass of the one above it. Inheritance hierarchies for interfaces are not displayed, because interfaces can potentially inherit multiple superinterfaces.

4.

The names of the interfaces implemented by the class to its left on the same line.

5.

Icons that indicate modifiers or members. If the "protected" symbol does not appear, the member is public. Private and package-private members have no symbols and are not shown.



abstract



final



static

[\[Team LiB \]](#)

[!\[\]\(e535ed85c7044d8fde266b330da82514_img.jpg\) PREVIOUS](#) [!\[\]\(a0feed22b371dfedf12710abddb09059_img.jpg\) NEXT](#) 

CLDC Almanac

ArithmeticsException

java.lang

[\[View full width\]](#)

```
object
  ↪ Throwable
    ↪ Exception
      ↪ RuntimeException
        ↪ ArithmeticsException
```

*

```
ArithmeticsException()
ArithmeticsException(String s)
```

ArrayIndexOutOfBoundsException

java.lang

[\[View full width\]](#)

```
Object
  ↪ Throwable
    ↪ Exception
      ↪ RuntimeException
        ↪ IndexOutOfBoundsException
          ↪ ArrayIndexOutOfBoundsException
```

*

```
ArrayIndexOutOfBoundsException()
ArrayIndexOutOfBoundsException(int index)
ArrayIndexOutOfBoundsException(String s)
```

ArrayStoreException

java.lang

[\[View full width\]](#)

```
Object
  ↪ Throwable
    ↪ Exception
      ↪ RuntimeException
        ↪ ArrayStoreException
```

*

```
ArrayStoreException()
ArrayStoreException(String s)
```

Boolean

java.lang

[\[View full width\]](#)

Appendix B. MIDP Application Programming Interface

This appendix contains the application programming interface documentation in Almanac format for the MIDP. For a description of this format, refer to "[Almanac Legend](#)" on page 356.

Full MIDP javadocs with detailed comments are available in the MIDP Specification (see "[Related Literature and Helpful Web Pages](#)" on page xxv) or as part of the MIDP reference implementation software that can be downloaded from Sun's web site (<http://www.sun.com/software/communitysource/j2me/midp/>).

[\[Team LiB \]](#)

[!\[\]\(b28cbb2b8c30ca6980a21dcf12aa6c31_img.jpg\) PREVIOUS](#)

MIDP Almanac

Alert

javax.microedition.lcdui

[\[View full width\]](#)

Object
 ↳ Displayable
 ↳ Screen
 ↳ Alert

```
void addCommand(Command cmd)
*
*
AlertType alertType)
MIDP 2.0 ↳
MIDP 2.0
MIDP 2.0

void addCommand(Command cmd)
    Alert(String title)
    Alert(String title, String alertText, Image alertImage,
AlertType alertType)
Command DISMISS_COMMAND
    int FOREVER
    int getDefaultTimeout()
    Image getImage()
    Gauge getIndicator()
    String getString()
    int getTimeout()
    AlertType getType()
    void removeCommand(Command cmd)
    void setCommandListener(CommandListener l)
    void setImage(Image img)
    void setIndicator(Gauge indicator)
    void setString(String str)
    void setTimeout(int time)
    void setType(AlertType type)
```

AlertType

javax.microedition.lcdui

[\[View full width\]](#)

Object
 ↳ AlertType

```
AlertType ALARM
    AlertType()
AlertType CONFIRMATION
AlertType ERROR
AlertType INFO
    boolean playSound(Display display)
AlertType WARNING
```

Canvas

javax.microedition.lcdui

[\[View full width\]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[\[Team LiB \]](#)

[Team LiB]

[[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

[Abstract Windowing Toolkit](#)

[addCommand](#)

[addTicker](#)

[Alert 2nd](#)

[Screen:Alert](#)

[Ticker on Alert](#)

[AlertType](#)

[alpha-channel](#)

[AMS](#)

[animated Gauge](#)

[ANY](#)

API documentation

[MIDP](#)

application

[development environment](#)

[management](#)

[management software](#)

application launch

[event-driven](#)

[usage scenarios](#)

[application management software](#)

[application management software \(AMS\)](#)

[application manager](#)

[Application provided operations](#)

application push

[sample application](#)

[application startup](#)

application:descriptor

[MIDP:application:descriptor](#)

application:Java

[Java application](#)

[Arabic](#)

[Arabic-Indic digit input](#)

[automatic casing](#)

automatic launching

[MIDlet](#)

[AWT](#)

[Team LiB]

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[BACK](#) [2nd](#) [3rd](#)

[background](#)

[background color](#)

[backlight](#)

[backward navigation](#)

[battery life](#)

[business applications](#)

[button](#) [2nd](#)

[BUTTON](#)

[byte arrays](#)

[converting record data to and from](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[CANCEL](#) [2nd](#)

[Canvas](#) [2nd](#) [See also [canvas](#)] [3rd](#) [See also [canvas](#)] [4th](#) [See also [canvas](#)]
[adaptation to device user interface style](#)

[class:Canvas:](#) [See also [Canvas](#)] [2nd](#) [See also [Canvas](#)] [3rd](#) [See also [Canvas](#)]

[canvas](#)

events

[action](#)
[key](#)
[pointer](#)

[canvas:coordinate system](#)

[coordinate system](#)

[canvas:redrawing mechanism](#)

redrawing

[repainting](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#) [8th](#) [9th](#)

[canvas:visibility](#)

[visibility](#)

[CDC](#)

[Connected Device Configuration:](#) [See [CDC](#)]

[certificate](#)

[root CA](#)

[certificate expiration and revocation](#)

[Certificate processing for Over-The-Air \(OTA\) delivery](#)

[Choice](#)

[ChoiceGroup](#) [2nd](#)

[Item:types:ChoiceGroup](#)

[class](#)

file

[format](#)

loaders

[user-defined](#)

Screen

[SeeScreen](#)

types

[collection](#)

[data](#) [2nd](#)

[input/output](#)

[class:Calendar](#)

[class:Time](#)

[Calendar;Time](#)

[class:CLDC-specific](#)

[CLDC:classes:-specific](#)

[class:CLDC:derived from J2SE](#)

[CLDC:classes:derived from J2SE](#)

[class:Date](#)

[Date](#)

[class:file:format:supported](#)

[format:class file:supported](#)

[class:file:lookup order](#)

[file:class:lookup order](#)

[class:file:lookup order:restrictions](#)

[file:class:lookup order:restrictions](#)

[class:loading](#)

[Team LiB]

[Team LiB]

[[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

[data type classes](#)

datagrams

[Generic Connection framework example](#)

DataSource

[class](#)

[and protocol handling](#)

[DateField](#) [2nd](#)

[Item:types:DateField](#)

[DECIMAL](#)

[default Command](#) [2nd](#)

[device fonts](#)

[device properties](#)

[device vibrator](#)

[device-provided operations](#)

devices

[consumer](#)

[specialized nature of](#)

[target](#)

[characteristics](#)

[Display](#) [2nd](#) [3rd](#)

[Displayable](#) [2nd](#) [3rd](#) [4th](#) [5th](#)

[actually visible](#)

[automatically changing](#)

drawing

images

[mutable](#)

drawing:arcs

arcs:drawing and filling

[filling:arcs](#)

drawing:lines

[lines:drawing](#)

drawing:rectangles

filling:rectangles

[rectangles:drawing and filling](#)

drawing:rectangles:rounded

filling:rectangles:rounded

[rectangles:rounded:drawing and filling](#) [2nd](#)

drawing:text

[text:drawing](#) [2nd](#) [3rd](#) [4th](#)

[Team LiB]

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[editing modes](#)

[EMAILADDR](#)

[embedded systems](#)

encoding:character

[character encoding](#)

error handling:CLDC

[CLDC:error handling](#)

errors

[CLDC:virtual machine differences:errors](#)

events

[action](#)

[key](#)

[pointer](#)

example

[NetClientMIDlet](#)

example:timers

[timers:example](#)

[EXCLUSIVE](#)

[EXIT 2nd](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

failure:non-transient

[non-transient failure](#)

failure:transient

[transient failure](#)

file:application resource

application:resource files

[MIDP:application:resources files](#)

file:application-specific resource

[application: resource files, specific](#)

finalization

[CLDC:finalization](#)

[CLDC:virtual machine differences:finalization](#)

flashBacklight

Focus

[focus](#)

[focus traversal](#)

font

[Font 2nd](#)

[FONT_INPUT_TEXT](#)

[FONT_STATIC_TEXT](#)

[fonts](#)

[graphics:fonts](#)

[foreground 2nd](#)

[foreground color](#)

[Form 2nd 3rd](#)

[Screen>Default Para Font>:Form](#)

[Form layout](#)

format

[class file](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[Game API](#)

[Gauge](#) [2nd](#)

[Item:types:Gauge](#)

[Generic Connection framework](#) [2nd](#)

[basic interface types](#)

[example](#)

[communication ports](#)

[datagrams](#)

[general form](#)

[HTTP](#)

[sockets](#) [2nd](#)

[used by MIDP](#)

[GET method](#)

[request:method:GET](#)

[getBestImageHeight](#)

[getBestImageWidth](#)

[getCipherSuite method](#)

[getColor](#)

[getCurrent](#)

[getDisplay](#)

[getFont](#)

[getHeight](#) [2nd](#)

[getServerCertificate method](#)

[getWidth](#) [2nd](#)

[Graphics](#)

[class:Graphics:](#) [See also graphics] [2nd](#) [See also graphics] [3rd](#) [See also graphics]

[graphics:drawing model](#)

[drawing](#)

[gray levels](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

HEAD method

[request:method:HEAD](#)

[headers](#) [See also request headers]

[HELP](#) 2nd

[hideNotify](#) 2nd

Hiragana script

[host operating system](#) 2nd

[kernel:](#) [See [host operating system](#)]

HTTP

[Generic Connection framework example](#)

HTTPS

[certificate processing for](#)

[HttpsConnection](#)

[interface](#)

[HttpsConnection interface](#)

[hyperlink](#)

[HYPERLINK](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[ImageItem](#) [2nd](#)

[class:ImageItem](#)

images

[creating and using](#)

[mutable](#) [2nd](#)

[incoming phone call](#)

[INCREMENTAL_UPDATING](#)

[indefinite Gauge](#)

[informational notes](#)

[Initial input mode](#)

[initial value of current alignment](#)

[INITIAL_CAPS_SENTENCE](#)

[INITIAL_CAPS_WORD](#)

[input constraint](#)

[input events](#)

[input mechanisms](#) [2nd](#)

[input methods](#) [2nd](#)

[input mode](#)

[Input Modes](#)

[input/output classes](#)

[InputConnection interface](#)

[insertion point](#)

[intent of a Command](#)

interaction modes

MIDlet suite

[trusted security mode](#)

interface

[Connection](#)

[Controllable](#)

[InputConnection](#)

interface:ContentConnection

[ContentConnection interface](#)

interface:DatagramConnection

[DatagramConnection interface](#)

interface:HttpConnection

[HttpConnection interface](#)

interface:OutputConnection

[OutputConnection interface](#)

interface:RecordListener

[RecordListener interface](#)

interface:StreamConnection

[StreamConnection interface](#)

interface:StreamConnectionNotifier

[StreamConnectionNotifier interface](#)

internationalization

[CLDC:internationalization](#)

Internet X.509 Public Key Infrastructure

[specification](#)

[interoperability](#)

[isColor](#)

[isShown](#) [2nd](#)

[Item](#) [2nd](#)

[Team LiB]

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[J2ME](#) [See [Java 2 Micro Edition](#)]

[Japanese language](#)

[Java 2 Micro Edition](#)

[architecture](#)

[chapter](#)

[Java 2 Micro Edition:configuration](#)

[configuration](#)

[Java 2 Micro Edition:optional package](#)

[optional package](#)

[Java 2 Micro Edition:profile](#)

[profile](#)

[Java Application Manager](#)

[JAM:](#) [See [Java Application Manager](#)]

[Java Archive files](#)

[JAR:](#) [See [Java Archive files](#)]

[Java Community Process](#)

[JCP:](#) [See [Java Community Process](#)]

[Java Language Specification:compatibility with](#)

[compatibility:with Java Language Specification](#)

[Java libraries](#)

[Java virtual machine](#)

[Java Virtual Machine Specification:compatibility with](#)

[compatibility:with Java Virtual Machine Specification](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

K Virtual Machine

[KVM](#): [See [K Virtual Machine](#)]

[key events](#)

[keys](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[label](#)

[Latin characters](#)

[layout 2nd](#)

[layout directives](#)

[LAYOUT_2](#)

[lcdui events](#)

[left-to-right layout](#)

[lifecycle of the application](#)

[lines:styles](#)

[graphics:line styles](#)

[List](#)

[class>List:SeeList](#)

[exclusive choice](#)

[implicit](#)

[List and Ticker](#)

[mutliple choice](#)

[setSelectCommand](#)

[lists \[See List\]](#)

[localization](#)

[locked Item](#)

[look and feel](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

Manager

[Class](#)

[class](#)

[media flow controls](#)

Media Framework

[Managing](#)

[menu](#) [2nd](#) [3rd](#)

[MIDI format](#) [2nd](#)

[MIDI format \(SP-MIDI\)](#)

[MIDlet](#)

[attributes](#)

[exiting](#)

[states](#) [2nd](#)

[active](#)

[destroyed](#)

[paused](#)

[transitions](#)

[transitions:how to request](#)

suites

[definition](#)

[execution environment](#)

[packaging](#)

MIDlet Launch

[alarm-based](#)

[network-based](#)

MIDlet suite

[removal](#)

[signing](#)

[signing certificate](#)

[trusted security mode](#)

[interaction modes](#)

[permissions](#)

[protection domain](#) [2nd](#)

MIDlet Suite

[updates](#)

MIDlet suite security

[caching of results](#)

MIDlet suites

[security for](#)

[untrusted](#)

MIDlet:application lifecycle

[application:MIDlet lifecycle](#)

MIDP

[API documentation](#)

[chapter](#)

[expert group](#) [2nd](#)

[goals](#)

libraries

[high-level user interface](#) [2nd](#) [3rd](#)

other APIs

[chapter](#)

[system software](#)

[Team LiB]

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[native applications](#)

[native settings](#)

[native user interface style](#)

[native user interfaces](#)

[Navigation](#)

[navigation](#)

[NetClientMIDlet](#)

[HTTP:example](#)

[Networking](#)

[Secure](#)

[New for MIDP 2.0](#)

[OTA provisioning](#)

[newline characters](#)

[NON_PREDICTIVE](#)

[numAlphaLevels](#)

[number of colors](#)

[numColors](#)

[NUMERIC](#)

[numeric keypad](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

Objects

[Media Control](#)

[OK 2nd](#)

optimization:implementation

[implementation:optimization](#)

optimization

[implementation-level](#)

over-the-air provisioning

[user-initiated](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

parsing URLs

[URLs:parsing](#)

[PASSWORD](#)

[PCM wav format](#)

[pen input](#)

permissions

MIDlet suite

[trusted security mode](#)

[phone book application](#)

[PHONENUMBER](#)

[PLAIN](#)

Player

[Class](#)

player

creating

[from HTTP locator media](#)

[from JAR media](#)

Player

[interface](#)

[life cycle](#)

[Player Creation and Management](#)

Player object

[states](#)

PlayerListener

[Class](#)

Players

[Creating](#)

[pointing device](#)

[POPUP](#)

[portability 2nd](#)

POST method

[request:method:POST](#)

pre-verification:off-device

[off-device pre-verification](#)

[predictive input 2nd](#)

[predictive input systems](#)

[preferred sizes of images](#)

prelinking

[class:prelinking](#)

preloading

[class:preloading](#)

[preverifier 2nd](#)

profile

[Java 2 Micro Edition:profile 2nd](#)

profile:definition

[Java 2 Micro Edition:profile:definition](#)

profile:MIDP

[Java 2 Micro Edition:profile:MIDP](#)

[progress animation](#)

[progress bar](#)

[progress indicator](#)

[property queries](#)

[Team LiB]

[Team LiB]

[[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

readers

writers

record

comparing

converting data

enumerating

filtering

listening

manipulating

stores

manipulating

names

Record Management System

RecordComparator interface

interface:RecordComparator

RecordEnumerator interface

interface:RecordEnumerator

RecordFilter interface

interface:RecordFilter

reminders

removeCommand

request

headers

Accept-Language

User-Agent

request:headers

HTTP:request headers

requirements

J2ME

response headers

headers:response

HTTP:response headers

RFC 2437

RFC 2459 2nd

right-to-left layout

RMS [See Record Management System]

RMSMIDlet.java example

example:RMS

ROMizing

class:ROMizing

RSA Encryption Version 2.0

specification

[Team LiB]

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[sandbox security model \(MIDP 1.0\)](#)

scope:CLDC

[CLDC:scope](#)

scope:MIDP

[MIDP:scope](#)

[Screen](#) 2nd

[SCREEN](#) 2nd

Screen

[List](#) [See [List](#)]

[Screen backlight](#)

[screen size](#)

screen:user interface model

[user interface:screen model](#)

[scrolling](#) 2nd 3rd

[Secure Networking](#)

[secure protocols](#)

SecureConnection

[example code](#)

[interface](#)

[security of](#)

[SecureConnection interface](#)

[security for MIDlet suites](#)

[security policy for GSM/UMTS devices](#)

[SecurityInfo object](#)

[select operation](#)

[SELECT_COMMAND](#)

[SENSITIVE](#)

[setCommandListener](#) 2nd

[setCurrent](#)

[setCurrent\(null\)](#)

[setCurrentItem](#)

[setInitialInputMode](#)

[setLayout](#)

[setTicker](#)

[setTitle](#)

[showNotify](#) 2nd

signing certificate

[MIDlet suite](#)

[size of the display](#)

[sizeChanged](#)

sockets

example

[Generic Connection framework](#)

[softkey](#) 2nd

sound and media support

[Controls](#)

[Manager](#)

[Player](#)

[SP-MIDI](#)

[Spacer](#) 2nd 3rd

[Spotless project](#)

[SSI](#)

[Team LiB]

[Team LiB]

[[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

[table](#)

[text editor](#)

[text input mode](#)

[TEXT_WRAP_OFF](#)

[TEXT_WRAP_ON](#)

[TextBox](#)

[Screen:TextBox](#)

[TextField_2nd](#)

[Item:types:TextField](#)

[textual input](#)

[thread groups](#)

 daemon groups

[CLDC:virtual machine differences:thread groups;CLDC:virtual machine differences:daemon groups](#)

[thread-safety](#)

[Ticker_2nd](#)

[timed Alert](#)

[timer](#)

timers

[one-shot](#)

[repeating](#)

[TimerTasks](#)

[in UI](#)

[TLS](#)

[tone generation](#)

[ToneControl](#)

[interface](#)

[touch screen](#)

[traversal_2nd](#)

[Traversal](#)

trust for MIDlet suites

 establishing

[using X.509 PKI](#)

[Team LiB]

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

UNEDITABLE

Unicode

Unicode character blocks

unlocked Item

URL

user interaction models 2nd

one-handed

touch screen

two-handed

user interface

high-level

low-level

user interface:touch input

touch input

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

verification: class file

[class:file:verification](#)

verification: runtime

[runtime:verification](#)

verifier: runtime

[runtime:verifier](#)

[vibrate](#)

[vibrator](#)

[virtual keyboards](#)

VolumeControl

[interface](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[WAP \(Wireless Application Protocol\)](#)

WAP Certificate Profile

[specification](#)

WAPCert

[specification](#)

[window management](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[X.509 certificate](#)

[X.509 Certificate Profile for Trusted MIDlet Suites](#)

X.509 PKI

using

[to establish trust for MIDlet suite](#)

[\[Team LiB \]](#)