Mobile Information Device Profile (JSR-37)

JCP Specification

Java 2 Platform, Micro Edition, 1.0a

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

1.0a, December 15, 2000

Copyright Notice

Contents

List of Tables

List of Figures

Preface

This document, *Mobile Information Device Profile (JSR-37) Specification*, defines the *Mobile Information Device Profile* (MIDP) for Java^{TM} 2 Platform, Micro Edition (J2ME^{TM}).

Revision History

TABLE P-1 Revision History

Date	Version	Description
Dec. 15, 2000	1.0a	Added ammended copyright. Added HTML version of Javadocs to download package. No changes to specification.
Sept. 1, 2000	1.0	Incorporated suggestions, changes, and defects from final public review.
July 14, 2000	0.95	Proposed final specification. Incorporates comments from public review. Sound API added.
May 5, 2000	0.9	First release for Public Draft. Changes incorporated from MIDPEG meeting (April 13–14).
April 13, 2000	0.8	Interim draft for MIDPEG internal review.
Mar. 21, 2000	0.7	Minor revisions for first release of Participant Draft.

TABLE P-1 Revision History

Date	Version	Description
Mar. 10, 2000	0.6	Preparing for Participant Draft. Addressed all comments from third meeting of the MIDPEG. Fixed minor formatting errors. Re-ordered chapters to move new APIs toward back of book. All changes marked with change bars.
Feb. 25, 2000	0.5	Final draft for MIDPEG meeting (Mar 2–3, 2000). Comments from MIDPEG incorporated. All revisions marked with chambers.
Feb. 18, 2000	0.4	Revisions now marked with change bars. Chapter 3, “Architecture” fleshed out. “**`javax.microedition.rms`**” on page 85 added.
Feb. 11, 2000	0.3	Revisions and addition of following material: Added Chapter 7, “Persistent Storage”
Feb. 4, 2000	0.2	Revisions and addition of the following material: Chapter 8, “Applications” Chapter 6, “Networking” Chapter 4, “System Functions” Chapter 5, “Timers”
Jan. 28, 2000	0.1	Initial release: Chapter 1, “Introduction and Background” **Chapter 2, “Requirements and Scope**” Chapter 3, “Architecture”

Who Should Use This Specification

The audience for this document is the Java Community Process (JCP) expert group defining this profile, implementors of the MIDP, and application developers targeting the MIDP.

A *profile* of J2ME defines device-type-specific sets of APIs for a particular vertical market or industry. Profiles are more exactly defined in the related publication, *Configurations and Profiles Architecture Specification*, Sun Microsystems, Inc.

How This Specification Is Organized

The topics in this specification are organized according to the following chapters:

Chapter 1, "Introduction and Background," provides a context for the MID Profile and defines key terms used in this specification.

Chapter 2, "Requirements and Scope," defines the scope of the specification and lists the requirements.

Chapter 3, "Architecture," defines the high-level architecture of the MIDP.

Chapter 4, "System Functions," defines how the MIDP extends or modifies APIs from the CLDC.

Chapter 5, "Timers," defines the MIDP APIs for calendar and time functions.

Chapter 6, "Networking," defines the networking APIs of the MIDP.

Chapter 7, "Persistent Storage," defines the storage APIs for the MIDP.

Chapter 8, "Applications," defines the concept of a MIDP application and provides an overview to the associated APIs.

Chapter 9, "User Interface," defines the graphical user interface APIs for the MIDP.

Appendix A, "Implementation Notes," discusses implementation issues for OEMs and developers.

Related Literature

The Java™ Language Specification by James Gosling, Bill Joy, and Guy L. Steele. Addison-Wesley, 1996, ISBN 0-201-63451-1

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

Connected, Limited Device Configuration (JSR-30), Sun Microsystems, Inc.

CHAPTER 1

Introduction and Background

1.1 Introduction

This document, produced as a result of Java Specification Request (JSR) 37, defines the Mobile Information Device Profile (MIDP) for the Java 2 Platform, Micro Edition (J2ME™). The goal of this specification is to define the architecture and the associated APIs required to enable an open, third-party, application development environment for mobile information devices, or MIDs.

The MIDP is designed to operate on top of the Connected Limited Device Configuration (CLDC) which is described in *Connected, Limited Device Configuration (JSR-30)*, Sun Microsystems, Inc.

1.2 Background

This specification was produced by the Mobile Information Profile Expert Group (MIDPEG). The following companies, listed in alphabetical order, are members of the MIDPEG:

- America Online
- DDI
- Ericsson
- Espial Group, Inc.
- Fujitsu
- Hitachi
- J-Phone

- Matsushita
- Mitsubishi
- Motorola, Inc.[1]
- NEC
- Nokia[2]
- NTT DoCoMo
- Palm
- Research In Motion
- Samsung
- Sharp
- Siemens
- Sony
- Sun Microsystems, Inc.[3]
- Symbian
- Telcordia Technologies, Inc.

1. Overall specification lead.
2. User-interface API leader.
3. Networking, persistent-storage, system, internationalization/localization, and timer API leader.

1.3 Document Conventions

1.3.1 Definitions

This document uses definitions based upon those specified in RFC 2119 (See http://www.ietf.org)

TABLE 1-1 Specification Terms

Term	Definition
MUST	The associated definition is an absolute requirement of this specification.
MUST NOT	The definition is an absolute prohibition of this specification.
SHOULD	Indicates a recommended practice. There may exist valid reasons in particular circumstances to ignore this recommendation, but the full implications must be understood and carefully weighed before choosing a different course.
SHOULD NOT	Indicates a non-recommended practice. There may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
MAY	Indicates that an item is truly optional.

1.3.2 Formatting Conventions

This specification uses the following formatting conventions.

TABLE 1-2 Formatting Conventions

Convention	Description
`Courier New`	Used in all Java code including keywords, data types, constants, method names, variables, class names, and interface names.
Italic	Used for emphasis and to signify the first use of a term.

CHAPTER **2**

Requirements and Scope

2.1 Requirements

The requirements listed in this chapter are additional requirements above those found in *Connected, Limited Device Configuration (JSR-30)*, Sun Microsystems, Inc.

At a high level, the MIDP specification assumes that the MID is limited in its processing power, memory, connectivity, and display size.

2.1.1 Hardware

As mentioned before, the main goal of the MIDPEG is to establish an open, third-party application development environment for MIDs. To achieve this goal, the MIDPEG has defined a MID to be a device that SHOULD have the following minimum characteristics:[1]

- 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 keyboard,"[2] "two-handed keyboard,"[3] or touch screen
- Memory:

1. Memory requirements cited here are for MIDP components only. CLDC and other system software memory requirements are beyond the scope of this specification and therefore not included.
2. A "one-handed" keyboard is a term used to describe an ITU-T phone keypad.
3. A "two-handed" keyboard is a term used to describe a QWERTY keyboard.

  - 128 kilobytes of non-volatile memory[4] for the MIDP components
  - 8 kilobytes of non-volatile memory for application-created persistent data
  - 32 kilobytes of volatile memory[5] for the Java runtime (e.g., the Java heap)
- Networking:
  - Two-way, wireless, possibly intermittent, with limited bandwidth

Examples of MIDs include, but are not restricted to, cellular phones, two-way pagers, and wireless-enabled personal digital assistants (PDAs).

2.1.2 Software

For devices with the aforementioned hardware characteristics, there is still a broad range of possible system software capabilities. Unlike the consumer desktop computer model where there are large, dominant system software architectures, the MID space is characterized by a wide variety of system software. For example, some MIDs may have a full-featured operating system that supports multi-processing[6] and hierarchical filesystems, while other MIDs may have small, thread-based operating systems with no notion of a filesystem. Faced with such variety, the MIDP makes minimal assumptions about the MID's system software. These requirements are as follows:

- A minimal *kernel* to manage the underlying hardware (i.e., handling of interrupts, exceptions, and minimal scheduling). This kernel must provide at least one schedulable entity to run the Java Virtual Machine (JVM). The kernel does not need to support separate address spaces (or *processes*) or make any guarantees about either real-time scheduling or latency behavior.
- A mechanism to read and write from non-volatile memory to support the APIs discussed in Chapter 7, "Persistent Storage."
- Read and write access to the device's wireless networking to support the APIs discussed in Chapter 6, "Networking."
- A mechanism to provide a time base for use in timestamping the records written to persistent storage (see Chapter 7, "Persistent Storage") and to provide the basis of the APIs in Chapter 5, "Timers."
- A minimal capability to write to a bit-mapped graphics display.
- A mechanism to capture user input from one (or more) of the three input mechanisms previously discussed (see "Hardware" on page 21).

4. *Non-volatile* means that the memory is expected to retain its contents between the user turning the devices "off" and "on". For the purposes of this specification, it is assumed that non-volatile memory is usually accessed in read mode, and that special setup may be required to write to it. Examples of non-volatile memory include ROM, flash, and battery-backed SDRAM. This specification does not define which memory technology a device must have, nor does it define the behavior of such memory in a power-loss scenario.

5. *Volatile* means that the memory that is not expected to retain its contents between the user turning the device "off" and "on". For the purpose of this specification, it is assumed that volatile memory accesses are evenly split between reads and writes, and no special setup is required to access it. The most common type of volatile memory is DRAM.

6. The ability to run multiple, concurrent processes, each with a separate and distinct memory map.

- A mechanism for managing the application life-cycle of the device. More information on application management can be found in “Implementation Notes” on page 65.

2.2 Scope

MIDs span a potentially wide set of capabilities. Rather than try to address all such capabilities, the MIDPEG agreed to limit the set of APIs specified, addressing only those APIs that were considered absolute requirements to achieve broad portability. These APIs are:

- Application (i.e., defining the semantics of a MIDP application and how it is controlled)
- User interface, or UI (includes display and input)
- Persistent storage
- Networking
- Timers

These APIs are discussed later in this document.

By the same reasoning, some areas of functionality were considered to be outside the scope of the MIDP. These areas include:

- **System-level APIs:** The emphasis on the MIDP APIs is, again, on enabling *application* programmers, rather than enabling *system* programming. Thus, low-level APIs that specify a system interface to a MID’s power management or voice CODECs are beyond the scope of this specification.
- **Application delivery and management:** While it is assumed that a MIDP-compliant device will support dynamic application downloading, the diversity of the worldwide wireless infrastructure makes it impractical to specify how the application download occurs. For example, in a low-bandwidth wireless network, it may not be practical for applications to be delivered to the device over the wireless link. Instead, such a device may opt to enable application downloading via a serial link or other physical links. What is assumed, however, is that an application running on a MID can access the network through specified network APIs. How applications are actually stored or installed on a MID is also beyond the scope of the specification. For a MID that has a full-featured, hierarchical filesystem, storage and installation is easy to accomplish. On the other hand, for devices that do not have a filesystem, application storage is much more problematic.
- **Low-level security**:[7] The MIDP specifies no additional low-level security features other than those provided by the CLDC.

7. *Low-level*, or *VM-level*, security ensures that an ill-formed or maliciously-encoded Java class file does not crash the MID’s Java Virtual Machine.

- **Application-level security**:[8] The MIDP's application model is described in Chapter 8, "Applications." Other than the semantics implied by the MIDP application model, the MIDP specifies no additional application-level security features other than those provided by the CLDC.
- **End-to-end security**:[9] Given the broad diversity of wireless infrastructure in the world, the MIDPEG found it impossible to architect a single end-to-end security mechanism.

8. *Application-level* security defines which APIs that the application can access.

9. *End-to-end* security establishes a model that guarantees that a transaction initiated on a MID is protected (i.e., encrypted, etc.) along the entire path from MID to/from the entity providing the services for that transaction.

CHAPTER 3

Architecture

3.1 Overview

This chapter addresses issues that both implementers and developers will encounter when implementing and developing MIDP. While not comprehensive, this chapter does reflect the most important issues raised during deliberations of the MIDPEG.

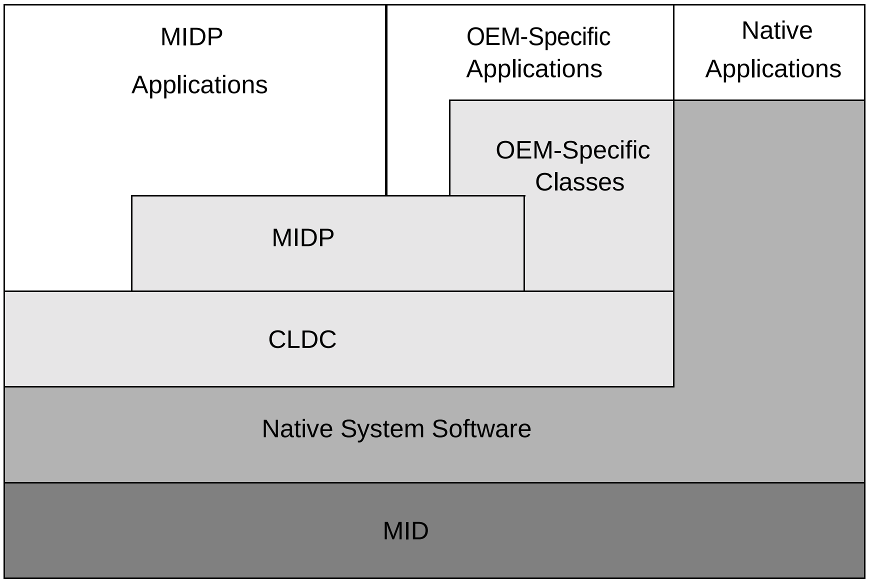
3.2 Architecture

As stated before, the goal of the MIDPEG is to create an open, third-party application development environment for MIDs. In a perfect world, this specification would only have to address functionality defined by the MIDP specification. In reality, most devices that implement the MIDP specification will be, at least initially, devices that exist on the market today. Figure 3-1 shows a high-level view of how the MIDP fits into a device. Note that not all devices that implement the MIDP specification will have all the elements shown in this figure, nor will every device necessarily layer its software as depicted in this figure.

In Figure 3-1, the lowest-level block (MID) represents the Mobile Information Device hardware. On top of this hardware is the native system software. This layer includes the operating system and libraries used by the device.

Starting at the next level, from left to right, is the next layer of software, the CLDC. This block represents the K-Virtual Machine (KVM) and associated libraries defined by the CLDC specification. This block provides the underlying Java functionality upon which higher-level Java APIs may be built.

FIGURE 3-1 High-Level Architecture View



Two categories of APIs are shown on top of the CLDC:

- **MIDP APIs:** The set of APIs defined in this document.
- **OEM-specific APIs:** Given the broad diversity of devices in the MIDP space, it is not possible to fully address all OEM requirements. These classes may be provided by an OEM to access certain functionality specific to a given device. These applications may not be portable to other MIDs.

Note that in the figure, the CLDC is shown as the basis of the MIDP and OEM-specific APIs. This does not imply that these APIs cannot have native functionality (i.e., methods declared as *native*). Rather, the intent of the figure is to show that any native methods on a MID are actually part of the KVM, which maps the Java-level APIs to the underlying native implementation.

The top-most blocks in Figure 3-1 represent the *application types* possible on a MID. A short description of each application type is shown in Table 3-1.

TABLE 3-1 **MID Application Types**

Application Type	Description
MIDP	A MIDP application, or *MIDlet*, is one that uses only the APIs defined by the MIDP and CLDC specifications. This type of application is the focus of the MIDP specification and is expected to be the most common type of application on a MID.
OEM-Specific	An OEM-specific application depends on classes that are not part of the MIDP specification (i.e., the OEM-specific classes). These applications are not portable across MIDs.
Native	A native application is one that is not written in Java and is built on top of the MID's existing, native system software.

It is beyond the scope of this specification to address OEM-specific or native applications.

CHAPTER 4

System Functions

4.1 Overview

The MIDP is based on the Connected, Limited Device Configuration (CLDC). Some features of the CLDC are modified or extended by the MIDP.

4.2 System Properties

The MIDP defines the following additional property values that MUST be made available to the application using `java.lang.System.getProperty`:

TABLE 4-1 System Properties Defined by MIDP

System Property	Description
`microedition.locale`	The current locale of the device (null by default)
`microedition.profiles`	must contain at least “MIDP-1.0”

Property microedition.locale

The locale property MUST consist of the language, country code, and variant separated by “-”. For example, “fr-FR” or “en-US.”

The language codes MUST be the lower-case, two-letter codes as defined by ISO-639 (See http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt).

The country code MUST be the upper-case, two-letter codes as defined by ISO-3166 (See http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html).

4.3 Application Resource Files

Application resource files are accessed using `getResourceAsStream(String name)` in `java.lang.Class`. In the MIDP specification, `getResourceAsStream` is used to allow resource files to be retrieved from the MIDlet Suite's JAR file.

4.4 System.exit

The behavior of `java.lang.System.exit` MUST throw a `java.lang.SecurityException` when invoked by a MIDlet. The only way a MIDlet can indicate that it is complete is by calling `MIDlet.notifyDestroyed`.

4.5 Runtime.exit

The behavior of `java.lang.Runtime.exit` MUST throw a `java.lang.SecurityException` when invoked by a MIDlet. The only way a MIDlet can indicate that it is complete is by calling `MIDlet.notifyDestroyed`.

CHAPTER 5

Timers

5.1 Overview

The MIDP adds functions that allow the application to set and be notified of timers.

5.2 Timers

Applications that need to delay or schedule activities for a later time can use the `Timer` and `TimerTask` classes, including functions for:

- one-time execution
- repeated execution at regular intervals

Classes

- `java.util.Timer`
- `java.util.TimerTask`

Please refer to "**`java.util`**" on page 75 for details on these APIs.

CHAPTER **6**

Networking

6.1 Overview

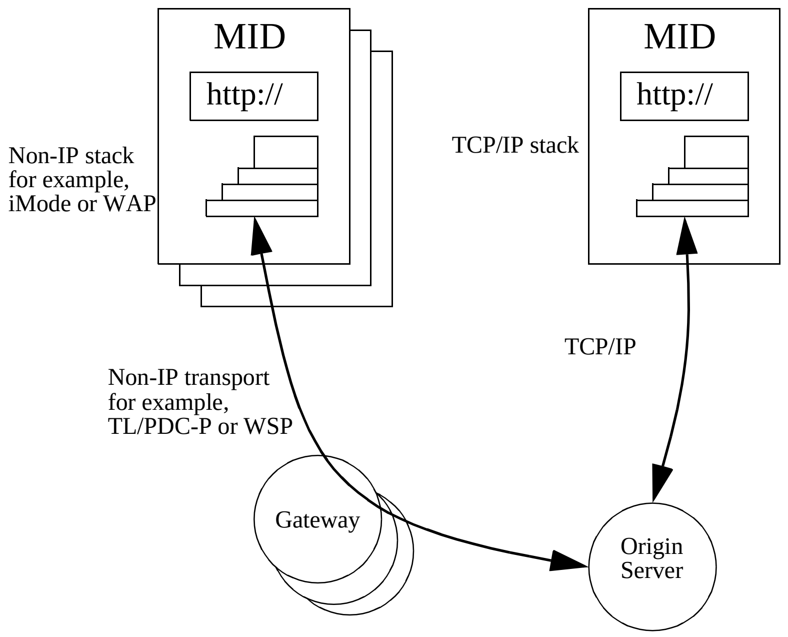
The MIDP extends the connectivity support provided by the Connected, Limited Device Configuration (CLDC) with specific functionality for the `GenericConnection` framework. The MIDP supports a subset of the HTTP protocol, which can be implemented using both IP protocols such as TCP/IP and non-IP protocols such as WAP and i-mode, utilizing a gateway to provide access to HTTP servers on the Internet.

The `GenericConnection` framework is used to support client-server and datagram networks. Using only the protocols specified by the MIDP will allow the application to be portable to all MIDs. MIDP implementations MUST provide support for accessing HTTP 1.1 servers and services.

There are wide variations in wireless networks. It is the joint responsibility of the device and the wireless network to provide the application service. It may require a *gateway* that can bridge between the wireless transports specific to the network and the wired Internet. The client application and the internet server MUST NOT need to be required to know either that non-IP networks are being used or the characteristics of those networks. While the client and server MAY both take advantage of such knowledge to optimize their transmissions, they MUST NOT be required to do so.

For example, a MID MAY have no in-device support for the Internet Protocol (IP). In this case, it would utilize a gateway (Figure 6-1) to access the Internet, and the gateway would be responsible for some services, such as DNS name resolution for Internet URLs. The device and network may define and implement security and network access policies that restrict access.

FIGURE 6-1 HTTP Network Connection



6.2 HttpConnection

The `GenericConnection` framework from the CLDC provides the base stream and content interfaces. The interface `HttpConnection` provides the additional functionality needed to set request headers, parse response headers, and perform other HTTP specific functions. Please refer to “**`javax.microedition.io`**” on page 131 for the details of the `javax.microedition.io.HttpConnection` API.

The interface MUST support:

- HTTP 1.1

Each device implementing the MIDP MUST support opening connections using the following URL schemes[1]:

- “http” as defined by RFC2616 *Hypertext Transfer Protocol -- HTTP/1.1*

1. RFC2396 *Uniform Resource Identifiers (URI): Generic Syntax.*

Each device implementing the MIDP MUST support the full specification of RFC2616 HEAD, GET and POST requests. The implementation MUST also support the absolute forms of URIs.

The implementation MUST pass all request headers supplied by the application and response headers as supplied by the network server. The ordering of request and response headers MAY be changed. While the headers may be transformed in transit, they MUST be reconstructed as equivalent headers on the device and server. Any transformations MUST be transparent to the application and origin server. The HTTP implementation does not automatically include any headers. The application itself is responsible for setting any request headers that it needs.

Connections may be implemented with any suitable protocol providing the ability to reliably transport the HTTP headers and data.[2]

6.2.1 HTTP Request Headers

The HTTP 1.1 specification provides a rich set of request and response headers that allow the application to negotiate the form, format, language, and other attributes of the content retrieved. In the MIDP, the application is responsible for selection and processing of request and response headers. Only the *User-Agent* header is described in detail. Any other header that is mutually agreed upon with the server may be used.

User-Agent Request Headers

For the MIDP, a simple *User-Agent* field may be used to identify the current device. As specified by RFC2616, the field contains blank separated features where the feature contains a name and optional version number.

The application is responsible for formatting and requesting that the *User-Agent* field be included in HTTP requests via the `setRequestProperty` method in the interface `javax.microedition.io.HttpConnection`. It can supply any application-specific features that are appropriate, in addition to any of the profile-specific request header values listed below.

Applications are not required to be loaded onto the device using HTTP. But if they are, then the *User-Agent* request header should be included in requests to load an application descriptor or application JAR file onto the device. This will allow the server to provide the most appropriate application for the device.

2. RFC2616 takes great care to not to mandate TCP streams as the only required transport mechanism.

The user-agent field SHOULD contain the following features as defined by system properties using `java.lang.System.getProperty`.

TABLE 6-1 System Properties Used for User-Agent Request Header

System Property	Description
`microedition.profiles`	The set of profiles supported by this device. For example, "MIDP-1.0."
`microedition.configuration`	The J2ME configuration supported by this device. For example, "CLDC-1.0."
`microedition.locale`	The name of the current locale on this device. For example, "en-US."

Example

```
User-Agent: Profile/MIDP-1.0 Configuration/CLDC-1.0
Content-Language: en-US
```

6.3 DatagramConnection

The MIDP specification does not mandate a datagram API/protocol for a MID. MIDs that implement a datagram API/protocol SHOULD use the `GenericConnection` framework (`DatagramConnection` interface) as defined by the CLDC specification.

CHAPTER 7

Persistent Storage

7.1 Overview

The MIDP 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.

7.2 Record Store

A record store consists of a collection of records that will remain persistent across multiple invocations of a MIDlet. The platform is responsible for making its best effort to maintain the integrity of the MIDlet's record stores throughout the normal use of the platform, including reboots, battery changes, etc.

Record stores are created in platform-dependent locations, which are not exposed to MIDlets. The naming space for record stores is controlled at the MIDlet suite granularity. MIDlets within a MIDlet suite are allowed to create multiple record stores, as long as they are each given different names. When a MIDlet suite is removed from a platform, all record stores associated with its MIDlets MUST also be removed. These APIs only allow the manipulation of the MIDlet suite's own record stores and do not provide any mechanism for record sharing between MIDlets in different MIDlet suites. MIDlets within a MIDlet suite can access one another's record stores directly.

Record store names are case sensitive and may consist of any combination of up to 32 Unicode characters. Record store names MUST be unique within the scope of a given MIDlet suite. In other words, MIDlets within a MIDlet suite are not allowed to create more than one record store with the same name; however, a MIDlet in 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 still distinct and separate.

No locking operations are provided in this API. Record store implementations ensure that all individual record store operations are atomic, synchronous, and serialized so that no corruption occurs with multiple accesses. However, if a MIDlet uses multiple threads to access a record store, it is the MIDlet's responsibility to coordinate this access, or unintended consequences may result. For example, if two threads in a MIDlet both call `RecordStore.setRecord()` concurrently on the same record, the record store will serialize these calls properly, and no database corruption will occur as a result. However, one of the writes will be subsequently overwritten by the other, which may cause problems within the MIDlet. Similarly, if a platform performs transparent synchronization of a record store or other access from below, it is the platform's responsibility to enforce exclusive access to the record store between the MIDlets and synchronization engine.

This record store API uses long integers for time/date stamps, in the format used by `System.currentTimeMillis()`. The record store is time stamped with the last time it was modified. The record store also maintains a *version*, which is an integer that is incremented for each operation that modifies the contents of the record store. These are useful for synchronization engines as well as applications.

7.3 Records

Records are arrays of bytes. Developers can use `DataInputStream` and `DataOutputStream` as well as `ByteArrayInputStream` and `ByteArrayOutputStream` to pack and unpack different data types into and out of the byte arrays.

Records are uniquely identified within a given record store by their `recordId`, which is an integer value. This `recordId` is used as the primary key for the records. The first record created in a record store will have `recordId` equal to 1, and each subsequent `recordId` will monotonically increase by one. For example, if two records are added to a record store, and the first has a `recordId` of 'n', the next will have a `recordId` of (n+1). MIDlets can create other indices by using the `RecordEnumeration` class.

CHAPTER **8**

Applications

8.1 Overview

The MIDP defines an application model to allow the limited resources of the device to be shared by multiple MIDP applications, or *MIDlets*. It defines what a MIDlet is, how it is packaged, what runtime environment is available to the MIDlet, and how it should be behave so that the device can manage its resources. The application model defines how multiple MIDlets forming a suite can be packaged together and share resources within the context of a single Java Virtual Machine. Sharing is feasible with the limited resources and security framework of the device since they are required to share class files and to be subject to a single set of policies and controls.

8.2 MIDP MIDlet Suite

A MIDP application MUST use only functionality specified by the MIDP specification as it is developed, tested, deployed, and run.

The elements of a MIDlet suite are:

- Runtime execution environment
- MIDlet suite packaging
- Application descriptor
- Application lifecycle

Each device is presumed to implement the functions required by its users to install, select, run, and remove MIDlets. The term *application management software* is used to refer collectively to these device specific functions (see Appendix A, “Implementation Notes”).

The application management software provides an environment in which the MIDlet is installed, started, stopped, and uninstalled. It is responsible for handling errors during the installation, execution, and removal of MIDlet suites and interacting with the user as needed. It provides to the MIDlet(s) the Java™ runtime environment required by the MIDP Specification.

One or more MIDlets MAY be packaged in a single JAR file. Each MIDlet consists of a class that extends the `MIDlet` class and other classes as may be needed by the MIDlet. The manifest in the JAR file identifies for each MIDlet the class implementing the MIDlet, its name, and icon. The MIDlet is the entity that is launched by the application management software. When a MIDlet suite is invoked, a Java Virtual Machine is needed on which the classes can be executed. A new instance of the MIDlet is created by the application management software and used to direct the MIDlet to start, pause, and destroy itself.

Sharing of data and other information between MIDlets is controlled by the individual APIs and their implementations. For example, the Record Management System API specifies the methods that should be used when the record stores associated with a MIDlet suite are shared among MIDlets.

8.3 MIDP Execution Environment

The MIDP defines the execution environment provided to MIDlets. The execution environment is shared by all MIDlets within a MIDlet suite, and any MIDlet can interact with other MIDlets packaged together. The application management software initiates the applications and makes the following available to the MIDlet:

- Classes and native code that implement the CLDC, including a Java Virtual Machine
- Classes and native code that implement the MIDP runtime
- All classes from a single JAR file for execution
- Non-class files from a single JAR file as resources
- Contents of the descriptor file

The CLDC and Java Virtual Machine provide multi-threading, locking and synchronization, the execution of byte codes, dispatching of methods, etc. A single VM is the scope of all policy, naming, and resource management. If a device supports multiple VMs, each may have its own scope, naming, and resource management policies. CLDC classes MUST NOT be superceded by MIDlet suite classes.

The MIDP provides the classes that implement the MIDP APIs. MIDP classes MUST NOT be superceded by MIDlet suite classes.

A single JAR file contains all of the MIDlet's classes. The MIDlet may load and invoke methods from any class in the JAR file, in the MIDP, or in the CLDC. All of the classes within these three scopes are shared in the execution environment of the MIDlets from the

JAR file. All states accessible via those classes are available to any Java class running on behalf of the MIDlet. There is a single space containing the objects of all MIDlets, MIDP, and CLDC in use by the MIDlet suite. The usual Java locking and synchronization primitives should be used when necessary to avoid concurrency problems. Each library will specify how it handles concurrency and how the MIDlet should use it to run safely in a multi-threaded environment.

The class files of the MIDlet are only available for execution and can neither be read as resources nor extracted for re-use. The implementation of the CLDC may store and interpret the contents of the JAR file in any manner suitable.

The files from the JAR file that are not Java class files are made available using methods on `java.lang.Class.getResourceAsStream` (see “Application Resource Files” on page 30). For example, the manifest would be available in this manner.

The contents of the MIDlet descriptor file, if it is present, are made available via the javax.microedition.midlet.`MIDlet.getAppProperty` method.

8.4 MIDlet Suite Packaging

One or more MIDlets are packaged in a single JAR file that includes:

- A manifest describing the contents
- Java classes for the MIDlet(s) and classes shared by the MIDlets
- Resource files used by the MIDlet(s)

The developer is responsible for creating and distributing the components of the JAR file as appropriate for the target user, device, network, locale, and jurisdiction. For example, for a particular locale, the resource files would be tailored to contain the strings and images needed for that locale.

The JAR manifest defines attributes that are used by the application management software to identify and install the MIDlet suite and as defaults for attributes not found in the application descriptor. The attributes are defined for use in both the manifest and the application descriptor.

The predefined attributes listed below allow the application management software to identify, retrieve, install, and invoke the MIDlet.

TABLE 8-1 MIDlet Attributes

Attribute Name	Attribute Description
MIDlet-Name	The name of the MIDlet suite that identifies the MIDlets to the user.
MIDlet-Version	The version number of the MIDlet suite. The format is major.minor.micro as described in the JDK Product Versioning Specification.[1] It can be used by the application management software for install and upgrade uses, as well as communication with the user.
MIDlet-Vendor	The organization that provides the MIDlet suite.
MIDlet-Icon	The name of a PNG file within the JAR file used to represent the MIDlet suite. It should be used when the Application Management Software displays an icon to identify the suite.
MIDlet-Description	The description of the MIDlet suite.
MIDlet-Info-URL	A URL for information further describing the MIDlet suite.
MIDlet-<n>	The name, icon, and class of the nth MIDlet in the JAR file separated by a comma. The lowest value of <n> MUST be 1 and consecutive ordinals MUST be used. 1. Name is used to identify this MIDlet to the user. 2. Icon is the name of an image (PNG) within the JAR for the icon of the nth MIDlet. 3. Class is the name of the class extending the `MIDlet` class for the nth MIDlet. The class MUST have a public no-args constructor.
MIDlet-Jar-URL	The URL from which the JAR file can be loaded.
MIDlet-Jar-Size	The number of bytes in the JAR file.
MIDlet-Data-Size	The minimum number of bytes of persistent data required by the MIDlet. The device may provide additional storage according to its own policy. The default is zero.
MicroEdition-Profile	The J2ME profile required, using the same format and value as the System property `microedition.profiles` (for example "MIDP-1.0").
MicroEdition-Configuration	The J2ME Configuration required using the same format and value as the System property `microedition.configuration` (for example "CLDC-1.0").

1. The Java™ Product Versioning Specification: http://java.sun.com/products/jdk/1.2/docs/guide/versioning/spec/VersioningSpecification.html

The version number has the format Major.Minor[.Micro] (X.X[.X]), where the .Micro portion MAY be omitted. (If the .Micro portion is not omitted, then it defaults to zero). In addition, each portion of the version number is allowed a maximum of two decimal digits (i.e., 0-99).

For example, 1.0.0 can be used to specify the first version of a MIDlet suite. For each portion of the version number, leading zeros are not significant. For example, 08 is equivalent to 8. Also, 1.0 is equivalent to 1.0.0. However, 1.1 is equivalent to 1.1.0, and not 1.0.1.

A missing MIDlet-Version tag is assumed to be 0.0.0, which means that any non-zero version number is considered as a newer version of the MIDlet suite.

8.4.1 JAR Manifest

The manifest provides information about the contents of the JAR file. JAR file formats and specifications are available at http://java.sun.com/products/jdk/1.2/docs/guide/jar/index.html. Refer to the JDK JAR and manifest documentation for the syntax and related details. Manifest attributes that start with "MIDlet-" and do not duplicate those in the application descriptor are passed to the MIDlet when requested.

The manifest MUST contain the following attributes:

- MIDlet-Name
- MIDlet-Version
- MIDlet-Vendor
- MIDlet-<n> for each MIDlet
- MicroEdition-Profile
- MicroEdition-Configuration

The manifest MAY contain the following:

- MIDlet-Description
- MIDlet-Icon
- MIDlet-Info-URL
- MIDlet-Data-Size

For example, a manifest for a hypothetical suite of card games would look like the following example:

```
MIDlet-Name: CardGames
MIDlet-Version: 1.1.9
MIDlet-Vendor: CardsRUS
MIDlet-1: Solitaire, /Solitare.png, com.cardsrus.org.Solitare
MIDlet-2: JacksWild, /JacksWild.png, com.cardsrus.org.JacksWild
MicroEdition-Profile: MIDP-1.0
MicroEdition-Configuration: CLDC-1.0
```

8.4.2 MIDlet Classes

All Java classes needed by the MIDlet are be placed in the JAR file using the standard structure, based on mapping the fully qualified class names to directory and file names. Each period is converted to a forward slash (/) and the `.class` extension is appended. For example, a class `com.sun.microedition.Test` would be placed in the JAR file with the name `com/sun/microedition/Test.class`.

8.5 Application Descriptor

The application descriptor is used by the application management software to manage the MIDlet and is used by the MIDlet itself for configuration specific attributes. Each JAR file may be accompanied by an application descriptor. The descriptor allows the application management software on the device to verify that the MIDlet is suited to the device before loading the full JAR file of the MIDlet suite. It also allows configuration-specific attributes (parameters) to be supplied to the MIDlet(s) without modifying the JAR file.

To allow the application management software to recognize a file as an application descriptor, a file extension and MIME type are defined.

- The file extension of an application descriptor file MUST be `jad`.
- The MIME type of an application descriptor file MUST be `text/vnd.sun.j2me.app-descriptor`.

Note – The extension and MIME type MUST be submitted to and approved by the Internet Assigned Numbers Authority (IANA).

A predefined set of attributes (Table 8-1) is specified to allow the application management software to identify, retrieve, and install the MIDlet(s). All attributes appearing in the descriptor file are made available to the MIDlet(s). The developer may use attributes not beginning with “MIDlet-” for application-specific purposes. All “MIDlet-” attributes are provided to the MIDlet(s). Attribute names are case-sensitive and MUST match exactly. The MIDlet retrieves attributes by name by calling the `MIDlet.getAppProperty` method.[1]

The application descriptor MUST contain the following attributes:

- MIDlet-Name
- MIDlet-Version
- MIDlet-Vendor
- MIDlet-Jar-URL
- MIDlet-Jar-Size

The application descriptor MAY contain:

- MIDlet-Description
- MIDlet-Icon
- MIDlet-Info-URL
- MIDlet-Data-Size
- MIDlet specific attributes that do not begin with “MIDlet-”

The mandatory attributes MIDlet-Name, MIDlet-Version, and MIDlet-Vendor MUST be duplicated in the descriptor and manifest files. If they are not identical, then the JAR MUST NOT be installed. While duplication of other attributes is not required, their values MAY differ even though both the manifest and descriptor files contain the same attribute. In this case, the value from the descriptor file will override the value from the manifest file.

Generally speaking, the format of the application descriptor is a sequence of lines consisting of an attribute name followed by a colon, the value of the attribute, and a carriage return. White space is ignored before and after the value. The order of the attributes is arbitrary.

The application descriptor MAY be encoded for transport or storage and MUST be decoded to Unicode before parsing, using the rules below. For example, an ISO8859-1 encoded file would need to be read through the equivalent of `java.io.InputStreamReader` with the appropriate encoding. Descriptors retrieved via HTTP, if that is supported, should use the standard HTTP content negotiation mechanisms, such as the Content-Encoding header and the Content-Type charset parameter to decode the stream to Unicode.

1. See “**`public final String getAppProperty (String key)`**” on page 126.

BNF for Parsing Application Descriptors

```
appldesc: *attrline
attrline: attrname ":" WSP attrvalue WSP newline

attrname: 1*<any Unicode char except CTLs or separators>
attrvalue: *valuechar | valuechar *(valuechar | WSP) valuechar
valuechar: <any valid Unicode character, excluding CTLS and WSP>

newline: CR LF | LF
CR = <Unicode carriage return (0x000D)>
LF = <Unicode linefeed (0x000a)>

WSP: 1*( SP | HT )
SP = <Unicode space (0x0020)>
HT = <Unicode horizontal-tab (0x0009)>
CTL = <Unicode characters 0x0000 - 0x001F and 0x007F>
separators = "(" | ")" | "<" | ">" | "@"
       | "," | ";" | ":" | "'" | <">
       | "/" | "[" | "]" | "?" | "="
       | "{" | "}" | SP | HT
```

For example, an application descriptor for a hypothetical suite of card games would look look like the following example:

```
MIDlet-Name: CardGames
MIDlet-Version: 1.1.9
MIDlet-Vendor: CardsRUS
MIDlet-Jar-URL: http://www.cardsrus.com/games/cardgames.jar
MIDlet-Jar-Size: 7378
MIDlet-Data-Size: 256
```

8.6 Application Lifecycle

Each MIDlet MUST extend the `MIDlet` class. The `MIDlet` class allows for the orderly starting, stopping, and cleanup of the MIDlet. The MIDlet can request the arguments from the application descriptor to communicate with the application management software. A MIDlet suite MUST NOT have a `public static void main()` method. If it exists, it MUST be ignored by the application management software. The application management software provides the initial class needed by the CLDC to start a MIDlet.

When a MIDlet suite is installed on a device, its classes, resource files, arguments, and persistent storage are kept *on the device* and ready for use. The MIDlet(s) are available to the user via the device's application management software.

When the MIDlet is run, an instance of the MIDlet's primary class is created using its public no-argument constructor, and the methods of the `MIDlet` are called to sequence the MIDlet through its various states. The MIDlet can either request changes in state or notify the application management software of state changes via the `MIDlet` methods. When the MIDlet is finished or terminated by the application management software, it is destroyed, and the resources it used can be reclaimed, including any objects it created and its classes. The MIDlet MUST NOT call `System.exit`, which will throw a `SecurityException` when called by a MIDlet. For a complete description of the classes and state changes, see "**`javax.microedition.midlet`**" on page 119.

The normal states of Java classes are not affected by these classes as they are loaded. Referring to any class will cause it to be loaded, and the normal static initialization will occur.

TABLE 8-2 **Classes in the `javax.microedition.midlet` Package**

Class in `javax.microedition.midlet`	**Description**
`MIDlet`	Extended by a MIDlet to allow the application management software to start, stop, and destroy it.
`MIDletStateChangeException`	Thrown when the application cannot make the change requested.

CHAPTER 9

User Interface

9.1 Overview

The main criteria for the MIDP have been drafted with mobile information devices in mind (i.e., high-end mobile phones and pagers). These devices differ from desktop systems in many ways, especially how the user interacts with them. The following UI-related requirements are important when designing the user interface API:

- 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 phone-type devices will be operated with one hand.
- The form factors and UI concepts of the device differ between devices, especially from desktop systems. For example, the display sizes are smaller, and the input devices do not always include pointing devices.
- The applications run on MIDs should have UIs that are compatible to the native applications so that the user finds them easy to use.

Given the capabilities of devices that will implement the MIDP (see Chapter 2, “Requirements and Scope”) and the above requirements, the MIDPEG decided not to simply subset the existing Java UI, which is the Abstract Windowing Toolkit (AWT). Reasons for this decision include:

- Although AWT was designed for desktop computers and optimized to these devices, it also suffers from assumptions based on this heritage.
- 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. At this point, the event object becomes garbage and must be reclaimed by the system’s garbage collector. The limited CPU and memory subsystems of a MID typically cannot handle this behavior.

- AWT has a rich but desktop-based feature set. This feature set includes support for features not found on MIDs. For example, AWT has extensive support for window management (e.g., overlapping windows, window resize, etc.). MIDs have small displays which are not large enough to display multiple overlapping windows. The limited display size also makes resizing a window impractical. As such, the windowing and layout manager support within AWT is not required for MIDs.
- AWT assumes certain user interaction models. The component set of AWT was designed to work with a *pointer device* (e.g., a mouse or pen input). As mentioned earlier, this assumption is valid for only a small subset of MIDs since many of these devices have only a keypad for user input.

9.2 Structure of the MIDP UI API

The MIDP UI is logically composed of two APIs: the high-level and the low-level.

The *high-level* API is designed for business applications whose client parts run on MIDs. For these applications, portability across devices is important. To achieve this portability, the high-level API employs a high level of abstraction and provides very little control over look and feel. This abstraction is further manifested in the following ways:

- The actual drawing to the MID’s display is performed by the implementation. Applications do not define the visual appearance (e.g., shape, color, font, etc.) of the components.
- Navigation, scrolling, and other primitive interaction is encapsulated by the implementation, and the application is not aware of these interactions.
- Applications cannot access concrete input devices like specific individual keys.

In other words, when using the high-level API, it is assumed that the underlying implementation will do the necessary adaptation to the device’s hardware and native UI style.

The *low-level* API, on the other hand, provides very little abstraction. This API is designed for applications that need precise placement and control of graphic elements, as well as access to low-level input events. Some applications also need to access special, device-specific features. A typical example of such an application would be a game.

Using the low-level API, an application can:

- Have full control of what is drawn on the display.
- Listen for primitive events like key presses and releases.
- Access concrete keys and other input devices.

Applications that program to the low-level API are not guaranteed to be portable, since the low-level API provides the means to access details that are specific to a particular device. If the application does not use these features, the applications will be portable, and it is recommended that the applications stick to the platform-independent part of the low-level API whenever possible. This means that the applications should not directly assume any other keys than defined in class `Canvas`, and should not depend on any specific screen size. Rather, the application game-event mechanism should be used instead of referring to concrete keys, and the application should inquire on the size of the display and adjust accordingly.

9.2.1 Class Hierarchy

The central abstraction of the MIDP's UI is a screen. A screen is an object that encapsulates device-specific graphics rendering user input. Only one screen may be visible at a time, and the user can only traverse through the items on that screen. The screen takes care of all events that occur as the user navigates in the screen, with only higher-level events being passed on to the application.

The rationale behind the screen-based design is based on the different display and keypad solutions found in MIDP devices. These differences imply that the component layout, scrolling, and focus traversal will be implemented differently on various devices. If an application had to be aware of these issues, portability would be compromised. Simple screens also organize the user interface into manageable pieces, resulting in user interfaces that are easy to use and learn.

There are three categories of screens:

- Screens that encapsulate a complex user interface component (e.g., classes `List` or `TextBox`). The structure of these screens is predefined, and the application cannot add other components to these screens.
- Generic screens (i.e., class `Form`) that the application can populate with text, images, and simple sets of related UI components.
- Screens that are used in context of the low-level API (i.e., subclasses of class `Canvas`).

Each screen, except the low-level `Canvas`, can attach a `Ticker`.

The class `Display` acts as the display manager that is instantiated for each active MIDlet and provides methods to retrieve information about the device's display capabilities. A `Screen` is made visible by calling the `setCurrent()` method of `Display`.

9.2.2 Class Overview

It is anticipated that most applications will utilize screens with predefined structures like `List`, `TextBox`, and `Alert`. These classes are used in the following ways:

- `List` is used when the user should select from a predefined set of choices.
- `TextBox` is used when asking textual input.
- `Alert` is used to display temporary messages containing text and images.

A special class `Form` is defined for cases where screens with a predefined structure are not sufficient. For example, an application may have two `TextFields`, or a `TextField` and a simple `ChoiceGroup`. Although this class (`Form`) allows creation of arbitrary combinations of components, developers should keep the limited display size in mind and create only simple `Forms`.

`Form` is designed to contain a small number of closely related UI elements. These elements are the subclasses of `Item`: `ImageItem`, `StringItem`, `TextField`, `ChoiceGroup`, and `Gauge`. The classes `ImageItem` and `StringItem` are convenience classes that make certain operations with `Form` and `Alert` easier. If the components do not all fit on the screen, the implementation may either make the form scrollable or implement some components so that they can either popup in a new screen or expand when the user edits the element.

9.2.3 Interplay with Application Manager

The user interface, like any other resource in the API, is to be controlled according to the principle of MIDP application management. The UI expects the following conditions from the application management software:

- `getDisplay()` is callable from `startApp()` until `destroyApp()` is returned.
- The Display object is the same until `destroyApp()` is called.
- The Displayable object set by `setCurrent()` is not changed by the application manager.

The application manager assumes that the application behaves as follows with respect to the MIDlet events:

- **`startApp`** - The application may call `setCurrent()` for the first screen. The application manager makes Displayable really visible when `startApp()` returns. Note that `startApp()` can be called several times if `pauseApp()` is called in between. This means that initialization should not take place, and the application should not accidentally switch to another screen with `setCurrent()`.
- **`pauseApp`** - The application may pause its threads. Also, if starting with another screen when the application is re-activated, the new screen should be set with `setCurrent()`.
- **`destroyApp`** - The application may delete created objects.

9.3 Event Handling

User interaction causes events, and the implementation notifies the application of the events by making corresponding callbacks. There are four kinds of UI callbacks:

- Abstract commands that are part of the high-level API
- Low-level events that represent single key presses and releases (and pointer events, if a pointer is available)
- Calls to the `paint()` method of a `Canvas` class
- Calls to a `Runnable` object's `run()` method requested by a call to `callSerially()` of class `Display`

All UI callbacks are serialized, so they will never occur in parallel. (Timer events are not considered UI events, and so timer callbacks may run concurrently with UI event callbacks. However, timer callbacks on the same `TimerTask` object are serialized with each other.) Otherwise, the UI callbacks are called as soon as possible after the previous UI callback returns. The implementation also guarantees that the call to `run()` requested by a call to `callSerially()` is made after any pending repaint requests have been satisfied.

9.3.1 Abstract Commands

Since MIDP UI is highly abstract, it does not dictate any concrete user interaction technique like soft buttons or menus. Also, low-level user interactions such as traversal or scrolling are not visible to the application. MIDP applications define `Commands`, and the implementation may manifest these via either abstract buttons, menus, or whatever mechanisms are appropriate for that device.

Commands are installed to a `Displayable` (`Canvas` or `Screen`) with a method `addCommand` of class `Displayable`.

The native style of the device may assume that certain types of commands are placed on standard places. For example, the “go-back” operation may always be mapped to the right soft button. The `Command` class allows the application to communicate such a semantic meaning to the implementation so that these standard mappings can be effected.

The `Command` objects have three constructor parameters:

- **`Label`**: Shown to the user as a hint.
- **`CommandType`**: The meaning of the command. One often used hint would be BACK, which causes the application to go back to a previous state. Most phone designs have a standard policy on which button is used for this operation. The `commandType` hint allows the implementation to take advantage of that policy.
- **`Priority`**: Provided to the implementation for better mapping to device capabilities.

There is also a select operation that could be, for example, implemented with GO/Select or a similar button. This button does not need to have a label, and its meaning should always be obvious to the user. For example, if the user is presented with a set of mutually exclusive options, the select operation will obviously select one of those options.

While pressing the select button does not usually cause command notification to the application, `List` of type `IMPLICIT` is an exception. Then notification with an implicit command, `SELECT_COMMAND`, is notified.

9.3.2 High-Level API for Events

The handling of events in the high-level API is based on a listener model. `Screen`s and `Canvase`s may have listeners for commands (see “Abstract Commands” on page 53). An object willing to be a listener should implement an interface `CommandListener` that has one method:

```
    void commandAction(Command c, Displayable d);
```

The application gets these events if the `Screen` or `Canvas` has attached `Command`s and if there is a registered listener. A unicast-version of the listener model is adopted, so the `Screen` or `Canvas` can have one listener at a time.

There is also a listener interface for state changes of the `Item`s in a `Form`. The method

```
    void itemStateChanged(Item item);
```

defined in interface `ItemStateListener` is called when the value of an interactive `Gauge`, `ChoiceGroup`, or `TextField` changes. It is not expected that the listener will be called after every change. However, if the value of an Item has been changed, the listener will be called for the change sometime before it is called for another item or before a command is delivered to the Form's CommandListener. It is suggested that the change listener is called at least after focus (or equivalent) is lost from field. The listener should only be called if the field’s value has actually changed.

9.3.3 Low-Level API for Events

Low-level graphics and events have the following methods to handle low-level key events:

```
public void keyPressed(int keyCode);
public void keyReleased(int keyCode);
public void keyRepeated(int keyCode);
```

The last call, `keyRepeated`, is not necessarily available in all devices. The applications can check the availability of repeat actions by calling the following method of the `Canvas`:

```
public static boolean hasRepeatEvents();
```

The API requires that there be standard key codes for the ITU-T keypad (0–9, *, #), but no keypad layout is required by the API. Although an implementation may provide additional keys, applications relying on these keys are not portable.

In addition, the class `Canvas` has methods for handling abstract game events. An implementation maps all these key events to suitable keys on the device. For example, a device with four-way navigation and a select key in the middle could use those keys, but a simpler device may use certain keys on the numeric keypad (e.g., 2, 4, 5, 6, 8). These game events allow development of portable applications that use the low-level events. The API defines a set of abstract key-events: UP, DOWN, LEFT, RIGHT, FIRE, GAME_A, GAME_B, GAME_C, and GAME_D.

An application can get the mapping of the key events to abstract key events by calling:

```
public static int getGameAction(int keyCode);
```

If the logic of the application is based on the values returned by this method, the application is portable and run regardless of the keypad design.

It is also possible to map abstract event to keys with:

```
public static int getKeyCode(int gameAction);
```

where `gameAction` is logical UP, DOWN, LEFT, RIGHT, FIRE, etc.

It is assumed that the mapping between keys and abstract events does not change during the execution of the game.

The following is an example of an application that retrieves and stores concrete key identifiers during its initialization phase. The application then uses these stored values during execution.

```
class TetrisCanvas extends Canvas {
    int leftKey, rightKey, downKey, rotateKey;

    void init () {
        leftKey = getKeyCode(LEFT);
        rightKey = getKeyCode(RIGHT);
        downKey = getKeyCode(DOWN);
        rotateKey = getKeyCode(FIRE);
    }

    public void keyPressed(int keyCode) {
        if (keyCode == leftKey) {
             moveBlockLeft();
        } else if (keyCode = rightKey) {
          ...
        }
    }
}
```

Another possibility would be to interpret the keys at runtime:

```
public void keyPressed(int keyCode) {
    int action = getGameAction(keyCode);
    if (action == LEFT) {
        moveBlockLeft();
     } else if (action == RIGHT) {
        ...
     }
}
```

The low-level API also has support for pointer events, but since the following input mechanisms may not be present in all devices, the following callback methods may never be called in some devices:

```
public void pointerPressed(int x, int y);
public void pointerReleased(int x, int y);
public void pointerDragged(int x, int y)
```

The application may check whether the pointer is available by calling the following methods of class `Canvas`:

```
    public static boolean hasPointerEvents();
    public static boolean hasPointerMotionEvents();
```

9.3.4 Interplay of High-Level Commands and the Low-Level API

The class `Canvas`, which is used for low-level events and drawing, is a subclass of `Displayable`, and applications can attach `Commands` to it. This is useful for jumping to an options setup `Screen` in the middle of a game. Another example could be a map-based navigation application where keys are used for moving in the map but commands are used for higher-level actions.

Some devices may not have the means to interact with command when `Canvas` and the low-level event mechanism are in use. In that case, the implementation may provide a means to switch to a command mode and back with some hot key. In this case, the running `Canvas` will receive messages `showNotify()` and `hideNotify()`.

9.4 Graphics and Text in Low-Level API

9.4.1 The Redrawing Scheme

Repainting is done automatically for all `Screens`, but not for `Canvas`; therefore, developers utilizing the low-level API must understand its repainting scheme.

In the low-level API, repainting of `Canvas` is done asynchronously so that several repaint requests may be implemented within a single call as an optimization. This means that the application requests the repainting by calling the method `repaint()` of class `Canvas`. The actual drawing is done in the method `paint()` — which is provided by the subclass `Canvas` — and does not necessarily happen synchronously to `repaint()`. It may happen later, and several repaint requests may cause one single call to `paint()`. The application can flush the repaint requests by calling `serviceRepaints()`.

As an example, assume that an application moves a box of width `wid` and height `ht` from coordinates (`x1,y1`) to coordinates (`x2,y2`), where `x2>x1` and `y2>y1`:

```
// move coordinates of box
box.x = x2;
box.y = y2;

// ensure old region repainted (with background)
canvas.repaint(x1,y1, wid, ht);

// make new region repainted
canvas.repaint(x2,y2, wid, ht);

// make everything really repainted
canvas.serviceRepaints();
```

The last call causes the repaint thread to be scheduled. The repaint thread finds the two requests from the event queue and repaints the region that is a union of the repaint area:

```
graphics.clipRect(x1,y1, (x2-x1+wid), (y2-y1+ht));
canvas.paint(graphics);
```

In this imaginary part of an implementation, the call `canvas.paint()` causes the application-defined `paint()` method to be called.

9.4.2 Drawing Model

The only drawing operation provided is pixel replacement. The destination pixel value is replaced by the current pixel value specified in the graphics object being used for rendering. No facility for combining pixel values, such as raster-ops or alpha blending, is provided.

A 24-bit color model is provided with 8 bits each for the red, green, and blue components of a color. Not all devices support 24-bit color, so they will map colors requested by the application into colors available on the device. Facilities are provided in the `Display` class for obtaining device characteristics, such as whether color is available and how many distinct gray levels are available. This enables applications to adapt their behavior to a device without compromising device independence.

Graphics may be rendered either directly to the display or to an off-screen image buffer. The destination of rendered graphics depends on the origin of the graphics object. A graphics object for rendering to the display is passed to the `Canvas` object's `paint()` method. This is the only way to obtain a graphics object whose destination is the display. Furthermore, applications may draw by using this graphics object only for the duration of the `paint()` method.

A graphics object for rendering to an off-screen image buffer may be obtained by calling the `getGraphics()` method on the desired image. These graphics objects may be held indefinitely by the application, and requests may be issued on these graphics objects at any time.

Note that class `Graphics` has only the method `setColor()` for setting the color instead of separate *setBackground* and *setForeground* calls. This means that backgrounds of drawing areas have to be drawn with explicit `fillRect()` calls.

9.4.3 Coordinate System

The origin (0,0) of the available drawing area and images is in the upper-left corner of the display. The numeric values of the x-coordinates monotonically increase from left to right, and the numeric values of the y-coordinates monotonically increase from top to bottom. Applications may assume that horizontal and vertical distances in 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 implementation of the UI will do the required coordinate transformation. A facility is provided for translating the origin of the coordinate system. All coordinates are specified as integers.

The coordinate system represents locations between pixels, not the pixels themselves. Therefore, the first pixel in the upper left corner of the display lies in the square bounded by coordinates (0,0), (1,0), (0,1), (1,1).

An application may inquire about the available drawing area by calling the following methods of `Canvas`:

```
    public static final int getWidth();
    public static final int getHeight();
```

9.4.4 Font Support

An application may request one of the font attributes specified below. However, the underlying implementation may use a subset of what is specified. So it is up to the implementation to return a font that most closely resembles the requested font.

Each font in the system is implemented individually. A programmer will call the static `getFont()` method instead of instantiating new `Font` objects. This paradigm eliminates the garbage creation normally associated with the use of fonts.

The `Font` class provides calls that access font metrics. The following attributes may be used to request a font (from the `Font` class):

- **Size**: SMALL, MEDIUM, LARGE.
- **Face**: PROPORTIONAL, MONOSPACE, SYSTEM.
- **Style:** PLAIN, BOLD, ITALIC, UNDERLINED.

9.4.5 Drawing Text and Images

By default, the drawing of text is based on anchor points instead of the standard notion of baseline. *Anchor points* are used to minimize the amount of computation required when placing text. For example, in order to center a piece of text, an application needs to call `stringWidth()` or `charWidth()` to get the width and then perform a combination of subtraction and division to compute the proper location.

The method to draw text is defined as follows:

```
public void drawString(String text, int x, int y, int anchor);
```

This method draws text in current foreground and background colors, using the current font with its anchor point at (`x,y`). The definition of the anchor point should be one of the horizontal constants (LEFT, HCENTER, RIGHT), logically combined (OR-ed) with one of the vertical constants (TOP, BOTTOM). Vertical centering of the text is not included in the API since it is hard to specify, not considered useful, and burdensome to implement. The default anchor point is 0, which signifies that the upper-left vertex of the text’s bounding box is used.

The actual position of the bounding box of the text relative to the (`x,y`) location is determined by the anchor point. These reference points occur at named locations along the outer edge of the bounding box. Thus, the following calls have identical results:

```
drawString(str, x, y, TOP|LEFT);
drawString(str, x + f.stringWidth(str)/2, y, TOP|HCENTER);
drawString(str, x + f.stringWidth(str), y, TOP|RIGHT);
drawString(str, x, y + f.getBaselinePosition(), BASELINE|LEFT);
drawString(str, x + f.stringWidth(str)/2,
           y + f.getBaselinePosition(), BASELINE|HCENTER);


drawString(str, x + f.stringWidth(str),
             y + f.getBaselinePosition(), BASELINE|RIGHT);

drawString(str, x, y + f.fontHeight(), BOTTOM|LEFT);
drawString(str, x + f.stringWidth(str)/2,
           y + f.fontHeight(),BOTTOM|HCENTER);
drawString(str, x + f.stringWidth(str), y +
           f.fontHeight(), BOTTOM|RIGHT);
```

For text drawing, character and line spacing are included as part of the values returned in the `Font.stringWidth()` and `Font.getHeight()` calls. For example, given the following code:

```
// (1)
drawString(string1+string2, x, y, TOP|LEFT);

// (2)
drawString(string1, x, y, TOP|LEFT);
f.getFont();
drawString(string2, x + f.stringWidth(string1), y,
             TOP|LEFT, string2);
```

code fragments (1) and (2) should behave identically. This relies on `Font.stringWidth()` to include the character spacing. Similarly, reasonable vertical spacing should be achieved simply by adding the font height to the Y-position of subsequent lines. For example:

```
drawString(string1 x, y, TOP|LEFT);
drawString(string2, x, y + getFont().fontHeight(),
             TOP|LEFT);
```

should draw `string1` and `string2` on separate lines with line spacing embedded in the font design. The font is assumed to include reasonable line spacing.

The `stringWidth()` of the string and the `fontHeight()` of the font in which it is drawn define the size of the bounding box of a piece of text. As described above, this box includes line and character spacing. The implementation is required to put this space below and to the right of the pixels actually belonging to the characters drawn. Applications that position graphics closely with respect to text (for example, to paint a rectangle around a string of text) may assume that there is space below and to the right of a string and that there is no space above and to the left of the string.

9.5 A Note on Concurrency

The UI API has been designed to be thread-safe. The methods may be called from callbacks, `TimerTasks`, or threads created by the application. Also, the implementation generally does not hold any locks on objects visible to the application. This means that the application synchronizes its own behavior by locking any object. There is one exception to the rule: `serviceRepaints()` of class `Canvas`. This method immediately calls the method `paint()`, but possibly in the context of different threads. If `paint()` tries to synchronize on any object that was locked by the application when `serviceRepaints()` was called, the application will deadlock. The application programmer should not hold any locks when `serviceRepaints()` is called. Also, locking an object used by `paint()` to synchronize is always an error.

The UI API includes also a mechanism similar to other UI toolkits for synchronization with events. With the method `callSerially()` of class `Display`, the application can execute an operation serially with events. `CallSerially()` can be used for the same effect as `serviceRepaints()`. The following code illustrates this implementation:

```
class MyCanvas extends Canvas {
    void doStuff() {
        //<code fragment 1>
        serviceRepaints();

        //<code fragment 2>
    }
}
```

The following code is an alternative way of implementing the same functionality:

```
class MyClass extends Canvas implements Runnable {
    void doStuff() {
        //<code fragment 1>
        callSerially(this);
    }

    public void run() {
        // called only after all pending repaints served
        //<<code fragment 2>;
    }
}
```

9.6 Implementation Notes

The implementation of a List or ChoiceGroup may include keyboard shortcuts for focusing and selecting the choice elements, but the use of these shortcuts is not visible to the application program.

In some implementations the UI components — Screens and Items — will be based on native components. It is up to the implementation to free the used resources when the Java objects are not needed anymore. One possible implementation scenario is a hook in the garbage collector of KVM.

APPENDIX **A**

Implementation Notes

A.1 Overview

This chapter addresses some possible implementation issues of the MIDP.

A.2 Implementation

This section discusses concepts that are not technically part of the MIDP specification but are fundamental issues for implementers of the MIDP.

A.2.1 Application Management

Throughout this document, frequent reference is made to an abstract entity called the *application management software*. In the context of this document, this term describes the software that controls how MIDlets are installed, upgraded, and de-installed from the MID. A more appropriate name for this entity might be *MIDlet management software*. This section describes some of the functionality of the MIDlet management software.

Note – The intent of this section is to show an *example* of possible functionality, not to mandate or specify this functionality.

To describe the functionality of the MIDlet management software, the different classes of MIDlets must be defined.

A.2.1.1 Classes of MIDlets

Broadly speaking, there are at least two possible classes of MIDlets. These classes are differentiated by how the MIDlet is retrieved and installed. These classes, along with some of their associated characteristics, are shown in Table A-1.

TABLE A-1 Possible Classes of MIDlets

MIDlet Class	Characteristics
Permanent	Resides, at least in part, in non-volatile memory. Performs many functions. May be run repeatedly without downloading again.
System	Resides, at least in part, in non-volatile memory. May be small or large. Performs device-specific functionality.

The first class of MIDlets is the *permanent* class. One use case for this MIDlet class would be the user of a MID browsing a selection of available MIDlets (e.g., games, etc.). After selecting a MIDlet, the MIDlet management software retrieves and permanently "installs" the MIDlet (i.e., writes it to persistent storage). The user may run this MIDlet repeatedly without retrieving the MIDlet again.

The other class of MIDlets is the *system* class, which is a special case of the permanent class. A system-class MIDlet is created by the manufacturer of the MID and performs device-specific functionality. System-class MIDlets may have access to non-public functionality of the device. So, in a sense, system-class MIDlets operate in a more privileged mode than other MIDlet classes. System-class MIDlets may have special constraints placed on their retrieval and installation.

MIDs may or may not support all of the classes of MIDlets described in this section. Other than recognizing the limitations of the target device (in terms of memory, etc.), a developer targeting the MIDP need not be aware of MIDlet-class distinctions.

For any of the MIDlet classes discussed in Section A.2.1.1, "Classes of MIDlets," there is an implied set of operations that the MIDlet management software must be able to perform. These operations are listed in Table A-2.[1]

TABLE A-2 Typical MIDlet Management Software Operations

Operation	Description
Retrieval	Retrieves the MIDlet from some source. Possible steps include *medium-identification, negotiation,* and *retrieval.*
Installation	Installs the MIDlet on the MID. Possible steps include *verification and transformation.*
Launching	Invokes the MIDlet. Possible steps include *inspection* and *invocation.*
Version Management	Allows installed MIDlets to be upgraded to newer versions. Possible steps include *inspection* and *version management*.
Removal	Removes a previously installed MIDlet. Possible steps include *inspection* and *deletion.*

Before a MIDlet can be launched, it must be retrieved from some source, called the *MIDlet-source*. A MID may have multiple mediums from which to retrieve MIDlets. For example, a MID may support retrieval via a serial cable, an IRDA port, or a wireless network. In this case, the MIDlet management software must support a *medium-identification* step in which the retrieval medium can be selected. After selecting the retrieval medium, the MIDlet management software can initiate the *negotiation* step. In this step, the MID and the MIDlet-source exchange information about the MIDlet and the MID. This information can include the MID's capability (e.g., available volatile memory), the size of the MIDlet,[2] cost, etc. Once the MID and the MIDlet-source have agreed that the MIDlet should be installed on the device, the *retrieval* step begins. In this step, the MID reads the MIDlet "into" the device.

Once the MIDlet has been retrieved, the installation process may begin. An implementation of the MIDP may need to *verify* that the retrieved MIDlet does not violate the MID's security policies. For example, a MID might enforce some sort of "code signing" mechanism to validate that the retrieved MIDlet is from a trusted source. The next step in installation is the *transformation* from the public representation of the MIDlet into some device-specific, internal representation.[3] This transformation may be as simple as writing the public representation to persistent storage, or it may actually entail preparing the MIDlet to execute directly from non-volatile memory.

1. While this chapter refers to operations on MIDlets, these operations are, in fact, operations on JAR files as described in Chapter 8, "Applications."
2. See description of the MIDlet-descriptor in Chapter 8, "Applications."
3. See the *Connected, Limited Device Configuration (JSR-30)*, Sun Microsystems, Inc., for a definition of public representation.

After installation, the MIDlet can now be *launched*. Launching a MIDlet means that the user is presented with a selection of installed MIDlets that are gathered by the MID performing the *inspection* step. The user may then select one of the MIDlets for the MID to run (or *invoke*). Invocation is the point at which the MIDlet enters the KVM. From this point, the APIs described in Chapter 8, “Applications,” are used to control the MIDlet.

At some point after installation, a new version of a MIDlet may become available. To upgrade to this new version, the MIDlet management software must keep track of what MIDlets have been installed (*identification*) and their “version number” (*version management)*. Using this information, the older version of the MIDlet can be upgraded to the newer version.

A related concept is MIDlet *removal*. This differs only slightly from the previous step in that after performing *inspection*, the MIDlet management software *deletes* the installed “image” of the MIDlet, and possibly its related resources. This may include records it has written to persistent storage via the APIs defined in Chapter 7, “Persistent Storage.”

A.2.1.2 Installation, Upgrade, and Removal

The application management software handles the device-specific functions for installing, removing, and running MIDlets. The application management software is responsible for the integrity and security of MIDlets on the device. It also presents the application model for the device to the user and handles errors that occur during the life cycle of a MIDlet.

The application management software installs or upgrades a MIDlet by examining the application descriptor and the corresponding JAR file.

When the user requests the installation of an MIDlet suite via its application descriptor or by presenting the JAR file, the application management software checks if it is one of the currently installed MIDlets. JAR files are uniquely identified by the MIDlet-name and MIDlet-vendor attributes from the manifest. If these values match, then the MIDlet suite is the same as one of the installed MIDlet suites. If so, and the MIDlet-Version of the requested version is newer than the installed version, the application management software may confirm with the user for approval before downloading and installing the newer version of the MIDlet suite.

The application management software should ensure that if the MIDlet suite update fails for any reason, the older version is left intact on the MIDP device. When the update is successful, the older version of the MIDlet suite should be removed. As part of the updating process, the persistent storage of the MIDlet suite should be preserved for use by the updated application.

When a MIDlet suite is removed, all components, persistent storage, and resources consumed by the MIDlet suite SHOULD be removed from the device.

The MIDP implementation will not be responsible for upgrading the format of the data in the RMS permanent storage. If an updated MIDlet uses a different data format than the version it is replacing, it will be the responsibility of the MIDlet to upgrade the data.

Note – There is no secure unique identifier for a MIDlet suite that is fully reliable and cannot be altered. The device's application management software may take additional actions to confirm the integrity of MIDlet suite to be installed.

Package
java.lang

Description

MID Profile Language Classes included from Java 2 Standard Edition. In addition to the `java.lang` classes specified in the Connected Limited Device Configuration the Mobile Information Device Profile includes the following class from Java 2 Standard Edition.

- java.lang.IllegalStateException.java

`IllegalStateException`s are thrown when illegal transitions are requested, such as scheduling a `TimerTask` or in the containment of user interface components.

Class Summary	
Classes	
`IllegalStateException`	Signals that a method has been invoked at an illegal or inappropriate time.

java.lang
IllegalStateException

Syntax

```
public class IllegalStateException extends java.lang.RuntimeException
```

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--java.lang.RuntimeException
                    |
                    +--java.lang.IllegalStateException
```

Description

Signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an appropriate state for the requested operation.

Since: JDK1.1

Member Summary	
Constructors	`public IllegalStateException ()`
	`public IllegalStateException (String s)`

Inherited Member Summary

Methods inherited from class Throwable

`getMessage(), printStackTrace(), toString()`

Methods inherited from class Object

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)`

Constructors

IllegalStateException()

```
public IllegalStateException ()
```

Constructs an IllegalStateException with no detail message. A detail message is a String that describes this particular exception.

IllegalStateException(String)

```
public IllegalStateException (String s)
```

Constructs an IllegalStateException with the specified detail message. A detail message is a String that describes this particular exception.

Parameters:

`s` - the String that contains a detailed message

Package java.util

Description

MID Profile Utility Classes included from Java 2 Standard Edition. In addition to the `java.util` classes specified in the Connected Limited Device Configuration the Mobile Information Device Profile includes the following classes from Java 2 Standard Edition.

- java.util.Timer
- java.util.TimerTask

Timers provide facility for an application to schedule task for future execution in a background thread. TimerTasks may be scheduled using Timers for one-time execution, or for repeated execution at regular intervals.

Class Summary

Classes	
`Timer`	A facility for threads to schedule tasks for future execution in a background thread.
`TimerTask`	A task that can be scheduled for one-time or repeated execution by a `Timer`.

java.util
Timer

Syntax

```
public class Timer

java.lang.Object
  |
  +--java.util.Timer
```

Description

A facility for threads to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.

Corresponding to each `Timer` object is a single background thread that is used to execute all of the timer's tasks, sequentially. Timer tasks should complete quickly. If a timer task takes excessive time to complete, it "hogs" the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "bunch up" and execute in rapid succession when (and if) the offending task finally completes.

After the last live reference to a `Timer` object goes away *and* all outstanding tasks have completed execution, the timer's task execution thread terminates gracefully (and becomes subject to garbage collection). However, this can take arbitrarily long to occur. By default, the task execution thread does not run as a *daemon thread*, so it is capable of keeping an application from terminating. If a caller wants to terminate a timer's task execution thread rapidly, the caller should invoke the timer's `cancel` method.

If the timer's task execution thread terminates unexpectedly, for example, because its `stop` method is invoked, any further attempt to schedule a task on the timer will result in an `IllegalStateException`, as if the timer's `cancel` method had been invoked.

This class is thread-safe: multiple threads can share a single `Timer` object without the need for external synchronization.

This class does *not* offer real-time guarantees: it schedules tasks using the `Object.wait(long)` method.

Timers function only within a single VM and are cancelled when the VM exits. When the VM is started no timers exist, they are created only by application request.

Since: 1.3

See Also: `TimerTask`, `Object.wait(long)`

Member Summary		
Constructors		
		`public Timer ()`
Methods		
	`void`	`public void cancel ()`
	`void`	`public void schedule (TimerTask task, Date time)`
	`void`	`public void schedule (TimerTask task, Date firstTime, long period)`
	`void`	`public void schedule (TimerTask task, long delay)`
	`void`	`public void schedule (TimerTask task, long delay, long period)`

Member Summary

void	public void scheduleAtFixedRate (TimerTask task, Date firstTime, long period)
void	public void scheduleAtFixedRate (TimerTask task, long delay, long period)

Inherited Member Summary

Methods inherited from class Object

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

Timer()

```
public Timer ()
```

Creates a new timer. The associated thread does *not* run as a daemon thread, which may prevent an application from terminating.

See Also: `Thread`, `public void cancel ()`

Methods

cancel()

```
public void cancel ()
```

Terminates this timer, discarding any currently scheduled tasks. Does not interfere with a currently executing task (if it exists). Once a timer has been terminated, its execution thread terminates gracefully, and no more tasks may be scheduled on it.

Note that calling this method from within the run method of a timer task that was invoked by this timer absolutely guarantees that the ongoing task execution is the last task execution that will ever be performed by this timer.

This method may be called repeatedly; the second and subsequent calls have no effect.

schedule(TimerTask, Date)

```
public void schedule (TimerTask task, Date time)
```

Schedules the specified task for execution at the specified time. If the time is in the past, the task is scheduled for immediate execution.

Parameters:

`task` - task to be scheduled.

`time` - time at which task is to be executed.

Throws: `IllegalArgumentException` - if `time.getTime()` is negative.

`IllegalStateException` - if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated.

schedule(TimerTask, Date, long)

```
public void schedule (TimerTask task, Date firstTime, long period)
```

Schedules the specified task for repeated *fixed-delay execution*, beginning at the specified time. Subsequent executions take place at approximately regular intervals, separated by the specified period.

In fixed-delay execution, each execution is scheduled relative to the actual execution time of the previous execution. If an execution is delayed for any reason (such as garbage collection or other background activity), subsequent executions will be delayed as well. In the long run, the frequency of execution will generally be slightly lower than the reciprocal of the specified period (assuming the system clock underlying `Object.wait(long)` is accurate).

Fixed-delay execution is appropriate for recurring activities that require "smoothness." In other words, it is appropriate for activities where it is more important to keep the frequency accurate in the short run than in the long run. This includes most animation tasks, such as blinking a cursor at regular intervals. It also includes tasks wherein regular activity is performed in response to human input, such as automatically repeating a character as long as a key is held down.

Parameters:

`task` - task to be scheduled.

`firstTime` - First time at which task is to be executed.

`period` - time in milliseconds between successive task executions.

Throws: `IllegalArgumentException` - if `time.getTime()` is negative.

`IllegalStateException` - if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated.

schedule(TimerTask, long)

```
public void schedule (TimerTask task, long delay)
```

Schedules the specified task for execution after the specified delay.

Parameters:

`task` - task to be scheduled.

`delay` - delay in milliseconds before task is to be executed.

Throws: `IllegalArgumentException` - if `delay` is negative, or `delay + System.currentTimeMillis()` is negative.

`IllegalStateException` - if task was already scheduled or cancelled, or timer was cancelled.

schedule(TimerTask, long, long)

`public void schedule (TimerTask task, long delay, long period)`

Schedules the specified task for repeated *fixed-delay execution*, beginning after the specified delay. Subsequent executions take place at approximately regular intervals separated by the specified period.

In fixed-delay execution, each execution is scheduled relative to the actual execution time of the previous execution. If an execution is delayed for any reason (such as garbage collection or other background activity), subsequent executions will be delayed as well. In the long run, the frequency of execution will generally be slightly lower than the reciprocal of the specified period (assuming the system clock underlying `Object.wait(long)` is accurate).

Fixed-delay execution is appropriate for recurring activities that require "smoothness." In other words, it is appropriate for activities where it is more important to keep the frequency accurate in the short run than in the long run. This includes most animation tasks, such as blinking a cursor at regular intervals. It also includes tasks wherein regular activity is performed in response to human input, such as automatically repeating a character as long as a key is held down.

Parameters:

`task` - task to be scheduled.

`delay` - delay in milliseconds before task is to be executed.

`period` - time in milliseconds between successive task executions.

Throws: `IllegalArgumentException` - if `delay` is negative, or `delay + System.currentTimeMillis()` is negative.

`IllegalStateException` - if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated.

scheduleAtFixedRate(TimerTask, Date, long)

`public void scheduleAtFixedRate (TimerTask task, Date firstTime, long period)`

Schedules the specified task for repeated *fixed-rate execution*, beginning at the specified time. Subsequent executions take place at approximately regular intervals, separated by the specified period.

In fixed-rate execution, each execution is scheduled relative to the scheduled execution time of the initial execution. If an execution is delayed for any reason (such as garbage collection or other background activity), two or more executions will occur in rapid succession to "catch up." In the long run, the frequency of execution will be exactly the reciprocal of the specified period (assuming the system clock underlying `Object.wait(long)` is accurate).

Fixed-rate execution is appropriate for recurring activities that are sensitive to *absolute* time, such as ringing a chime every hour on the hour, or running scheduled maintenance every day at a particular time. It is also appropriate for for recurring activities where the total time to perform a fixed number of executions is important, such as a countdown timer that ticks once every second for ten seconds. Finally, fixed-rate execution is appropriate for scheduling multiple repeating timer tasks that must remain synchronized with respect to one another.

Parameters:

`task` - task to be scheduled.

`firstTime` - First time at which task is to be executed.

`period` - time in milliseconds between successive task executions.

Throws: `IllegalArgumentException` - if `time.getTime()` is negative.

`IllegalStateException` - if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated.

scheduleAtFixedRate(TimerTask, long, long)

```
public void scheduleAtFixedRate (TimerTask task, long delay, long period)
```

Schedules the specified task for repeated *fixed-rate execution*, beginning after the specified delay. Subsequent executions take place at approximately regular intervals, separated by the specified period.

In fixed-rate execution, each execution is scheduled relative to the scheduled execution time of the initial execution. If an execution is delayed for any reason (such as garbage collection or other background activity), two or more executions will occur in rapid succession to "catch up." In the long run, the frequency of execution will be exactly the reciprocal of the specified period (assuming the system clock underlying `Object.wait(long)` is accurate).

Fixed-rate execution is appropriate for recurring activities that are sensitive to *absolute* time, such as ringing a chime every hour on the hour, or running scheduled maintenance every day at a particular time. It is also appropriate for for recurring activities where the total time to perform a fixed number of executions is important, such as a countdown timer that ticks once every second for ten seconds. Finally, fixed-rate execution is appropriate for scheduling multiple repeating timer tasks that must remain synchronized with respect to one another.

Parameters:

`task` - task to be scheduled.

`delay` - delay in milliseconds before task is to be executed.

`period` - time in milliseconds between successive task executions.

Throws: `IllegalArgumentException` - if `delay` is negative, or `delay + System.currentTimeMillis()` is negative.

`IllegalStateException` - if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated.

java.util
TimerTask

Syntax

```
public abstract class TimerTask implements java.lang.Runnablet

java.lang.Object
  |
  +--java.util.TimerTask
```

All Implemented Interfaces: Runnable

Description

A task that can be scheduled for one-time or repeated execution by a `Timer`.

Since: 1.3

See Also: `Timer`

Member Summary

Constructors		
		`protected TimerTask ()`
Methods		
	`boolean`	`public boolean cancel ()`
	`void`	`public abstract void run ()`
	`long`	`public long scheduledExecutionTime ()`

Inherited Member Summary

Methods inherited from class Object

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

TimerTask()

```
protected TimerTask ()
```

Creates a new timer task.

Methods

cancel()

`public boolean cancel ()`

Cancels this timer task. If the task has been scheduled for one-time execution and has not yet run, or has not yet been scheduled, it will never run. If the task has been scheduled for repeated execution, it will never run again. (If the task is running when this call occurs, the task will run to completion, but will never run again.)

Note that calling this method from within the `run` method of a repeating timer task absolutely guarantees that the timer task will not run again.

This method may be called repeatedly; the second and subsequent calls have no effect.

Returns: true if this task is scheduled for one-time execution and has not yet run, or this task is scheduled for repeated execution. Returns false if the task was scheduled for one-time execution and has already run, or if the task was never scheduled, or if the task was already cancelled. (Loosely speaking, this method returns `true` if it prevents one or more scheduled executions from taking place.)

run()

`public abstract void run ()`

The action to be performed by this timer task.

Specified By: `Runnable.run()` in interface Runnable

scheduledExecutionTime()

`public long scheduledExecutionTime ()`

Returns the *scheduled* execution time of the most recent *actual* execution of this task. (If this method is invoked while task execution is in progress, the return value is the scheduled execution time of the ongoing task execution.)

This method is typically invoked from within a task's run method, to determine whether the current execution of the task is sufficiently timely to warrant performing the scheduled activity:

```
public void run() {
    if (System.currentTimeMillis() - scheduledExecutionTime() >=
        MAX_TARDINESS)
            return;  // Too late; skip this execution.
    // Perform the task
}
```

This method is typically *not* used in conjunction with *fixed-delay execution* repeating tasks, as their scheduled execution times are allowed to drift over time, and so are not terribly significant.

Returns: the time at which the most recent execution of this task was scheduled to occur, in the format returned by `Date.getTime()`. The return value is undefined if the task has yet to commence its first execution.

See Also: `Date.getTime()`

Package javax.microedition.rms

Description

The Mobile Information Device Profile provides a mechanism for MIDlets to persistently store data and later retrieve it. This persistent storage mechanism is modeled after a simple record oriented database and is called the Record Management System.

Example:

The example uses the Record Management System to store and retrieve high scores for a game. In the example, high scores are stored in separate records, and sorted when necessary using a RecordEnumeration.

```
import javax.microedition.rms.*;
import java.io.DataOutputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ByteArrayInputStream;
import java.io.DataInputStream;
import java.io.EOFException;
/**
 * A class used for storing and showing game scores.
 */
public class RMSGameScores
    implements RecordFilter, RecordComparator
{
    /*
     * The RecordStore used for storing the game scores.
     */
    private RecordStore recordStore = null;
    /*
     * The player name to use when filtering.
     */
    public static String playerNameFilter = null;
    /*
     * Part of the RecordFilter interface.
     */
    public boolean matches(byte[] candidate)
throws IllegalArgumentException
    {
// If no filter set, nothing can match it.
if (this.playerNameFilter == null) {
    return false;
}
ByteArrayInputStream bais = new ByteArrayInputStream(candidate);
DataInputStream inputStream = new DataInputStream(bais);
String name = null;
try {
    int score = inputStream.readInt();
    name = inputStream.readUTF();
}
catch (EOFException eofe) {
    System.out.println(eofe);
    eofe.printStackTrace();
}
catch (IOException eofe) {
    System.out.println(eofe);
    eofe.printStackTrace();
}
return (this.playerNameFilter.equals(name));
    }
    /*
     * Part of the RecordComparator interface.
     */
    public int compare(byte[] rec1, byte[] rec2)
    {
// Construct DataInputStreams for extracting the scores from
// the records.
ByteArrayInputStream bais1 = new ByteArrayInputStream(rec1);
DataInputStream inputStream1 = new DataInputStream(bais1);
ByteArrayInputStream bais2 = new ByteArrayInputStream(rec2);
DataInputStream inputStream2 = new DataInputStream(bais2);
int score1 = 0;
int score2 = 0;
try {
    // Extract the scores.
    score1 = inputStream1.readInt();
    score2 = inputStream2.readInt();
}
catch (EOFException eofe) {
    System.out.println(eofe);
```

```
        eofe.printStackTrace();
    }
    catch (IOException eofe) {
        System.out.println(eofe);
        eofe.printStackTrace();
    }
    // Sort by score
    if (score1 < score2) {
        return RecordComparator.PRECEDES;
    }
    else if (score1 > score2) {
        return RecordComparator.FOLLOWS;
    }
    else {
        return RecordComparator.EQUIVALENT;
    }
        }
        /**
         * The constructor opens the underlying record store,
         * creating it if necessary.
         */
        public RMSGameScores()
        {
    //
    // Create a new record store for this example
    //
    try {
        recordStore = RecordStore.openRecordStore("scores", true);
    }
    catch (RecordStoreException rse) {
        System.out.println(rse);
        rse.printStackTrace();
    }
        }
        /**
         * Add a new score to the storage.
         *
         * @param score the score to store.
         * @param playerName the name of the play achieving this score.
         */
        public void addScore(int score, String playerName)
        {
    //
    // Each score is stored in a separate record, formatted with
    // the score, followed by the player name.
    //
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream outputStream = new DataOutputStream(baos);
    try {
        // Push the score into a byte array.
        outputStream.writeInt(score);
        // Then push the player name.
        outputStream.writeUTF(playerName);
    }
    catch (IOException ioe) {
        System.out.println(ioe);
        ioe.printStackTrace();
    }
    // Extract the byte array
    byte[] b = baos.toByteArray();
    // Add it to the record store
    try {
        recordStore.addRecord(b, 0, b.length);
    }
    catch (RecordStoreException rse) {
        System.out.println(rse);
        rse.printStackTrace();
    }
```

```
        }
        /**
         * A helper method for the printScores methods.
         */
        private void printScoresHelper(RecordEnumeration re)
        {
    try {
        while(re.hasNextElement()) {
    int id = re.nextRecordIndex();
    ByteArrayInputStream bais = new ByteArrayInputStream(recordStore.getRecord(id));
    DataInputStream inputStream = new DataInputStream(bais);
    try {
        int score = inputStream.readInt();
        String playerName = inputStream.readUTF();
        System.out.println(playerName + " = " + score);
    }
    catch (EOFException eofe) {
        System.out.println(eofe);
        eofe.printStackTrace();
    }
        }
    }
    catch (RecordStoreException rse) {
        System.out.println(rse);
        rse.printStackTrace();
    }
    catch (IOException ioe) {
        System.out.println(ioe);
        ioe.printStackTrace();
    }
        }
        /**
         * This method prints all of the scores sorted by game score.
         */
        public void printScores()
        {
    try {
        // Enumerate the records using the comparator implemented
        // above to sort by game score.
        RecordEnumeration re = recordStore.enumerateRecords(null, this,
    true);
        printScoresHelper(re);
    }
    catch (RecordStoreException rse) {
        System.out.println(rse);
        rse.printStackTrace();
    }
        }
        /**
         * This method prints all of the scores for a given player,
         * sorted by game score.
         */
        public void printScores(String playerName)
        {
    try {
        // Enumerate the records using the comparator and filter
        // implemented above to sort by game score.
        RecordEnumeration re = recordStore.enumerateRecords(this, this,
    true);
        printScoresHelper(re);
    }
    catch (RecordStoreException rse) {
        System.out.println(rse);
        rse.printStackTrace();
    }
        }
        public static void main(String[] args)
        {
```

```
RMSGameScores rmsgs = new RMSGameScores();
rmsgs.addScore(100, "Alice");
rmsgs.addScore(120, "Bill");
rmsgs.addScore(80, "Candice");
rmsgs.addScore(40, "Dean");
rmsgs.addScore(200, "Ethel");
rmsgs.addScore(110, "Farnsworth");
rmsgs.addScore(220, "Farnsworth");
System.out.println("All scores");
rmsgs.printScores();
System.out.println("Farnsworth's scores");
RMSGameScores.playerNameFilter = "Farnsworth";
rmsgs.printScores("Farnsworth");
    }
}
```

Class Summary

Interfaces

`RecordComparator`	An interface defining a comparator which compares two records (in an implementation-defined manner) to see if they match or what their relative sort order is.
`RecordEnumeration`	A class representing a bidirectional record store Record enumerator.
`RecordFilter`	An interface defining a filter which examines a record to see if it matches (based on an application-defined criteria).
`RecordListener`	A listener interface for receiving Record Changed/Added/Deleted events from a record store.

Classes

`InvalidRecordIDException`	Thrown to indicate an operation could not be completed because the record ID was invalid.
`RecordStore`	A class representing a record store.
`RecordStoreException`	Thrown to indicate a general exception occurred in a record store operation.
`RecordStoreFullException`	Thrown to indicate an operation could not be completed because the record store system storage is full.
`RecordStoreNotFoundException`	Thrown to indicate an operation could not be completed because the record store could not be found.
`RecordStoreNotOpenException`	Thrown to indicate that an operation was attempted on a closed record store.

javax.microedition.rms

InvalidRecordIDException

Syntax

```
public class InvalidRecordIDException extends RecordStoreException
```

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--RecordStoreException
                    |
                    +--javax.microedition.rms.InvalidRecordIDException
```

Description

Thrown to indicate an operation could not be completed because the record ID was invalid.

Member Summary

Constructors

`public InvalidRecordIDException ()`
`public InvalidRecordIDException (String message)`

Inherited Member Summary

Methods inherited from class `Throwable`

`getMessage(), printStackTrace(), toString()`

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)`

Constructors

InvalidRecordIDException()

```
public InvalidRecordIDException ()
```

Constructs a new `InvalidRecordIDException` with no detail message.

InvalidRecordIDException(String)

```
public InvalidRecordIDException (String message)
```

Constructs a new `InvalidRecordIDException` with the specified detail message.

Parameters:
`message` - the detail message.

javax.microedition.rms
RecordComparator

Syntax

```
public interface RecordComparator
```

Description

An interface defining a comparator which compares two records (in an implementation-defined manner) to see if they match or what their relative sort order is. The application implements this interface to compare two candidate records. The return value must indicate the ordering of the two records. The compare method is called by RecordEnumeration to sort and return records in an application specified order. For example:

```
RecordComparator c = new AddressRecordComparator(); //
 class implements RecordComparator
if (c.compare(recordStore.getRecord(rec1), recordStore.getRecord(rec2)) == RecordCompara
tor.PRECEDES)
return rec1;
```

Member Summary		
Fields		
	`int`	`public static final int EQUIVALENT`
	`int`	`public static final int FOLLOWS`
	`int`	`public static final int PRECEDES`
Methods		
	`int`	`public int compare (byte[] rec1, byte[] rec2)`

Fields

EQUIVALENT

```
public static final int EQUIVALENT
```

EQUIVALENT means that in terms of search or sort order, the two records are the same. This does not necessarily mean that the two records are identical.

The value of EQUIVALENT is 0.

FOLLOWS

```
public static final int FOLLOWS
```

FOLLOWS means that the left (first parameter) record *follows* the right (second parameter) record in terms of search or sort order.

The value of FOLLOWS is 1.

PRECEDES

```
public static final int PRECEDES
```

PRECEDES means that the left (first parameter) record *precedes* the right (second parameter) record in terms of search or sort order.

The value of PRECEDES is -1.

Methods

compare(byte[], byte[])

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

Returns `RecordComparator.PRECEDES` if rec1 precedes rec2 in sort order, or `RecordComparator.FOLLOWS` if rec1 follows rec2 in sort order, or `RecordComparator.EQUIVALENT` if rec1 and rec2 are equivalent in terms of sort order.

Parameters:

`rec1` - The first record to use for comparison. Within this method, the application must treat this parameter as read-only.

`rec2` - The second record to use for comparison. Within this method, the application must treat this parameter as read-only.

Returns: `RecordComparator.PRECEDES` if rec1 precedes rec2 in sort order, or `RecordComparator.FOLLOWS` if rec1 follows rec2 in sort order, or `RecordComparator.EQUIVALENT` if rec1 and rec2 are equivalent in terms of sort order.

javax.microedition.rms

RecordEnumeration

Syntax

```
public interface RecordEnumeration
```

Description

A class representing a bidirectional record store Record enumerator. The RecordEnumeration logically maintains a sequence of the recordId's of the records in a record store. The enumerator will iterate over all (or a subset, if an optional record filter has been supplied) of the records in an order determined by an optional record comparator.

By using an optional `RecordFilter`, a subset of the records can be chosen that match the supplied filter. This can be used for providing search capabilities.

By using an optional `RecordComparator`, the enumerator can index through the records in an order determined by the comparator. This can be used for providing sorting capabilities.

If, while indexing through the enumeration, some records are deleted from the record store, the recordId's returned by the enumeration may no longer represent valid records. To avoid this problem, the RecordEnumeration can optionally become a listener of the RecordStore and react to record additions and deletions by recreating its internal index. Use special care when using this option however, in that every record addition, change and deletion will cause the index to be rebuilt, which may have serious performance impacts.

The first call to `nextRecord()` returns the record data from the first record in the sequence. Subsequent calls to `nextRecord()` return the next consecutive record's data. To return the record data from the previous consecutive from any given point in the enumeration, call `previousRecord()`. On the other hand, if after creation, the first call is to `previousRecord()`, the record data of the last element of the enumeration will be returned. Each subsequent call to `previousRecord()` will step backwards through the sequence.

Final note, to do record store searches, create a RecordEnumeration with no RecordComparator, and an appropriate RecordFilter with the desired search criterion.

Member Summary

Methods

`void`	`public void destroy ()`
`boolean`	`public boolean hasNextElement ()`
`boolean`	`public boolean hasPreviousElement ()`
`boolean`	`public boolean isKeptUpdated ()`
`void`	`public void keepUpdated (boolean keepUpdated)`
`byte[]`	`public byte[] nextRecord ()`
`int`	`public int nextRecordId ()`
`int`	`public int numRecords ()`
`byte[]`	`public byte[] previousRecord ()`
`int`	`public int previousRecordId ()`
`void`	`public void rebuild ()`
`void`	`public void reset ()`

Methods

destroy()

`public void destroy ()`

Frees internal resources used by this RecordEnumeration. MIDlets should call this method when they are done using a RecordEnumeration. If a MIDlet tries to use a RecordEnumeration after this method has been called, it will throw a `IllegalStateException`.

hasNextElement()

`public boolean hasNextElement ()`

Returns true if more elements exist in the *next* direction.

Returns: true if more elements exist in the *next* direction.

hasPreviousElement()

`public boolean hasPreviousElement ()`

Returns true if more elements exist in the *previous* direction.

Returns: true if more elements exist in the *previous* direction.

isKeptUpdated()

`public boolean isKeptUpdated ()`

Returns true if the enumeration keeps its enumeration current with any changes in the records.

Returns: true if the enumeration keeps its enumeration current with any changes in the records.

keepUpdated(boolean)

`public void keepUpdated (boolean keepUpdated)`

Used to set whether the enumeration will be keep its internal index up to date with the record store record additions/deletions/changes. Note that this should be used carefully due to the potential performance problems associated with maintaining the enumeration with every change.

Parameters:

`keepUpdated` - If true, the enumerator will keep its enumeration current with any changes in the records of the record store. Use with caution as there are possible performance consequences. If false the enumeration will not be kept current and may return recordIds for records that have been deleted or miss records that are added later. It may also return records out of order that have been modified after the enumeration was built. Note that any changes to records in the record store are accurately reflected when the record is later retrieved, either directly or through the enumeration. The thing that is risked by setting this parameter false is the filtering and sorting order of the enumeration when records are modified, added, or deleted.

See Also: `public void rebuild ()`

nextRecord()

```
public byte[] nextRecord ()
```

Returns a copy of the *next* record in this enumeration, where *next* is defined by the comparator and/or filter supplied in the constructor of this enumerator. The byte array returned is a copy of the record. Any changes made to this array will NOT be reflected in the record store. After calling this method, the enumeration is advanced to the next available record.

Returns: the next record in this enumeration.

Throws: `InvalidRecordIDException` - when no more records are available. Subsequent calls to this method will continue to throw this exception until `reset()` has been called to reset the enumeration.

`RecordStoreNotOpenException` - if the record store is not open.

`RecordStoreException` - if a general record store exception occurs.

nextRecordId()

```
public int nextRecordId ()
```

Returns the recordId of the *next* record in this enumeration, where *next* is defined by the comparator and/or filter supplied in the constructor of this enumerator. After calling this method, the enumeration is advanced to the next available record.

Returns: the recordId of the next record in this enumeration.

Throws: `InvalidRecordIDException` - when no more records are available. Subsequent calls to this method will continue to throw this exception until `reset()` has been called to reset the enumeration.

numRecords()

```
public int numRecords ()
```

Returns the number of records available in this enumeration's set. That is, the number of records that have matched the filter criterion. Note that this forces the RecordEnumeration to fully build the enumeration by applying the filter to all records, which may take a non-trivial amount of time if there are a lot of records in the record store.

Returns: the number of records available in this enumeration's set. That is, the number of records that have matched the filter criterion.

previousRecord()

```
public byte[] previousRecord ()
```

Returns a copy of the *previous* record in this enumeration, where *previous* is defined by the comparator and/or filter supplied in the constructor of this enumerator. The byte array returned is a copy of the record. Any changes made to this array will NOT be reflected in the record store. After calling this method, the enumeration is advanced to the next (previous) available record.

Returns: the previous record in this enumeration.

Throws: `InvalidRecordIDException` - when no more records are available. Subsequent calls to this method will continue to throw this exception until `reset()` has been called to reset the enumeration.

`RecordStoreNotOpenException` - if the record store is not open.

`RecordStoreException` - if a general record store exception occurs.

previousRecordId()

```
public int previousRecordId ()
```

Returns the recordId of the *previous* record in this enumeration, where *previous* is defined by the comparator and/or filter supplied in the constructor of this enumerator. After calling this method, the enumeration is advanced to the next (previous) available record.

Returns: the recordId of the previous record in this enumeration.

Throws: `InvalidRecordIDException` - when no more records are available. Subsequent calls to this method will continue to throw this exception until `reset()` has been called to reset the enumeration.

rebuild()

```
public void rebuild ()
```

Request that the enumeration be updated to reflect the current record set. Useful for when a MIDlet makes a number of changes to the record store, and then wants an existing RecordEnumeration to enumerate the new changes.

See Also: `public void keepUpdated (boolean keepUpdated)`

reset()

```
public void reset ()
```

Returns the enumeration index to the same state as right after the enumeration was created.

javax.microedition.rms
RecordFilter

Syntax

```
public interface RecordFilter
```

Description

An interface defining a filter which examines a record to see if it matches (based on an application-defined criteria). The application implements the match() method to select records to be returned by the RecordEnumeration. Returns true if the candidate record is selected by the RecordFilter. This interface is used in the record store for searching or subsetting records. For example:

```
    RecordFilter f = new DateRecordFilter(); // class implements RecordFilter
    if (f.matches(recordStore.getRecord(theRecordID)) == true)
    DoSomethingUseful(theRecordID);
```

Member Summary	
Methods	
boolean	public boolean matches (byte[] candidate)

Methods

matches(byte[])

```
public boolean matches (byte[] candidate)
```

Returns true if the candidate matches the implemented criterion.

Parameters:

`candidate` - The record to consider. Within this method, the application must treat this parameter as read-only.

Returns: true if the candidate matches the implemented criterion.

javax.microedition.rms

RecordListener

Syntax

```
public interface RecordListener
```

Description

A listener interface for receiving Record Changed/Added/Deleted events from a record store.

Member Summary		
Methods		
	`void`	`public void recordAdded (RecordStore recordStore, int recordId)`
	`void`	`public void recordChanged (RecordStore recordStore, int recordId)`
	`void`	`public void recordDeleted (RecordStore recordStore, int recordId)`

Methods

recordAdded(RecordStore, int)

```
public void recordAdded (RecordStore recordStore, int recordId)
```

Called when a record has been added to a record store.

Parameters:

`recordStore` - the RecordStore in which the record is stored.

`recordId` - the recordId of the record that has been added.

recordChanged(RecordStore, int)

```
public void recordChanged (RecordStore recordStore, int recordId)
```

Called after a record in a record store has been changed. If the implementation of this method retrieves the record, it will receive the changed version.

Parameters:

`recordStore` - the RecordStore in which the record is stored.

`recordId` - the recordId of the record that has been changed.

recordDeleted(RecordStore, int)

```
public void recordDeleted (RecordStore recordStore, int recordId)
```

Called after a record has been deleted from a record store. If the implementation of this method tries to retrieve the record from the record store, an InvalidRecordIDException will be thrown.

Parameters:

`recordStore` - the RecordStore in which the record was stored.

`recordId` - the recordId of the record that has been deleted.

javax.microedition.rms

RecordStore

Syntax

```
public class RecordStore
```

```
java.lang.Object
  |
  +--javax.microedition.rms.RecordStore
```

Description

A class representing a record store. A record store consists of a collection of records which will remain persistent across multiple invocations of the MIDlet. The platform is responsible for making its best effort to maintain the integrity of the MIDlet's record stores throughout the normal use of the platform, including reboots, battery changes, etc.

Record stores are created in platform-dependent locations, which are not exposed to the MIDlets. The naming space for record stores is controlled at the MIDlet suite granularity. MIDlets within a MIDlet suite are allowed to create multiple record stores, as long as they are each given different names. When a MIDlet suite is removed from a platform all the record stores associated with its MIDlets will also be removed. These APIs only allow the manipulation of the MIDlet suite's own record stores, and does not provide any mechanism for record sharing between MIDlets in different MIDlet suites. MIDlets within a MIDlet suite can access each other's record stores directly.

Record store names are case sensitive and may consist of any combination of up to 32 Unicode characters. Record store names must be unique within the scope of a given MIDlet suite. In other words, a MIDlets within a MIDlet suite are is not allowed to create more than one record store with the same name, however a MIDlet in different one MIDlet suites are is allowed to each have a record store with the same name as a MIDlet in another MIDlet suite. In that case, the record stores are still distinct and separate.

No locking operations are provided in this API. Record store implementations ensure that all individual record store operations are atomic, synchronous, and serialized, so no corruption will occur with multiple accesses. However, if a MIDlet uses multiple threads to access a record store, it is the MIDlet's responsibility to coordinate this access or unintended consequences may result. Similarly, if a platform performs transparent synchronization of a record store, it is the platform's responsibility to enforce exclusive access to the record store between the MIDlet and synchronization engine.

Records are uniquely identified within a given record store by their recordId, which is an integer value. This recordId is used as the primary key for the records. The first record created in a record store will have recordId equal to one (1). Each subsequent record added to a RecordStore will be assigned a recordId one greater than the record added before it. That is, if two records are added to a record store, and the first has a recordId of 'n', the next will have a recordId of 'n + 1'. MIDlets can create other indices by using the `RecordEnumeration` class.

This record store uses long integers for time/date stamps, in the format used by System.currentTimeMillis(). The record store is time stamped with the last time it was modified. The record store also maintains a *version*, which is an integer that is incremented for each operation that modifies the contents of the RecordStore. These are useful for synchronization engines as well as other things.

Member Summary
Methods

Member Summary

`int`	`public int addRecord (byte[] data, int offset, int numBytes)`
`void`	`public void addRecordListener (RecordListener listener)`
`void`	`public void closeRecordStore ()`
`void`	`public void deleteRecord (int recordId)`
`void`	`public static void deleteRecordStore (String recordStoreName)`
`RecordEnumeration`	`public RecordEnumeration enumerateRecords (RecordFilter filter, RecordComparator comparator, boolean keepUpdated)`
`long`	`public long getLastModified ()`
`String`	`public String getName ()`
`int`	`public int getNextRecordID ()`
`int`	`public int getNumRecords ()`
`byte[]`	`public byte[] getRecord (int recordId)`
`int`	`public int getRecord (int recordId, byte[] buffer, int offset)`
`int`	`public int getRecordSize (int recordId)`
`int`	`public int getSize ()`
`int`	`public int getSizeAvailable ()`
`int`	`public int getVersion ()`
`String[]`	`public static String[] listRecordStores ()`
`RecordStore`	`public static RecordStore openRecordStore (String recordStoreName, boolean createIfNecessary)`
`void`	`public void removeRecordListener (RecordListener listener)`
`void`	`public void setRecord (int recordId, byte[] newData, int offset, int numBytes)`

Inherited Member Summary

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Methods

addRecord(byte[], int, int)

```
public int addRecord (byte[] data, int offset, int numBytes)
```

Adds a new record to the record store. The recordId for this new record is returned. This is a blocking atomic operation. The record is written to persistent storage before the method returns.

Parameters:

`data` - The data to be stored in this record. If the record is to have zero-length data (no data), this parameter may be null.

`offset` - The index into the data buffer of the first relevant byte for this record.

`numBytes` - The number of bytes of the data buffer to use for this record (may be zero).

Returns: the recordId for the new record.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

`RecordStoreException` - if a different record store-related exception occurred.

`RecordStoreFullException` - if the operation cannot be completed because the record store has no more room.

addRecordListener(RecordListener)

```
public void addRecordListener (RecordListener listener)
```

Adds the specified RecordListener. If the specified listener is already registered, it will not be added a second time. When a record store is closed, all listeners are removed.

Parameters:
`listener` - the RecordChangedListener.

closeRecordStore()

```
public void closeRecordStore ()
```

This method is called when the MIDlet requests to have the record store closed. Note that the record store will not actually be closed until closeRecordStore() is called as many times as openRecordStore() was called. In other words, the MIDlet needs to make a balanced number of close calls as open calls before the record store is closed.

When the record store is closed, all listeners are removed. If the MIDlet attempts to perform operations on the RecordStore object after it has been closed, the methods will throw a RecordStoreNotOpenException.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

`RecordStoreException` - if a different record store-related exception occurred.

deleteRecord(int)

```
public void deleteRecord (int recordId)
```

The record is deleted from the record store. The recordId for this record is NOT reused.

Parameters:
`recordId` - The ID of the record to delete.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

`InvalidRecordIDException` - if the recordId is invalid.

`RecordStoreException` - if a general record store exception occurs.

deleteRecordStore(String)

```
public static void deleteRecordStore (String recordStoreName)
```

Deletes the named record store. MIDlet suites are only allowed to operate on their own record stores, including deletions. If the record store is currently open by a MIDlet when this method is called, or if the named record store does not exist, a RecordStoreException will be thrown.

Parameters:
`recordStoreName` - The MIDlet suite unique record store to delete.

Throws: `RecordStoreException` - if a record store-related exception occurred.
`RecordStoreNotFoundException` - if the record store could not be found.

enumerateRecords(RecordFilter, RecordComparator, boolean)

```
public RecordEnumeration enumerateRecords (RecordFilter filter,
          RecordComparator comparator, boolean keepUpdated)
```

Returns an enumeration for traversing a set of records in the record store in an optionally specified order.

The filter, if non-null, will be used to determine what subset of the record store records will be used.

The comparator, if non-null, will be used to determine the order in which the records are returned.

If both the filter and comparator are null, the enumeration will traverse all records in the record store in an undefined order. This is the most efficient way to traverse all of the records in a record store.

The first call to `RecordEnumeration.nextRecord()` returns the record data from the first record in the sequence. Subsequent calls to `RecordEnumeration.nextRecord()` return the next consecutive record's data. To return the record data from the previous consecutive from any given point in the enumeration, call `previousRecord()`. On the other hand, if after creation the first call is to `previousRecord()`, the record data of the last element of the enumeration will be returned. Each subsequent call to `previousRecord()` will step backwards through the sequence.

Parameters:
`filter` - if non-null, will be used to determine what subset of the record store records will be used.

`comparator` - if non-null, will be used to determine the order in which the records are returned.

`keepUpdated` - If true, the enumerator will keep its enumeration current with any changes in the records of the record store. Use with caution as there are possible performance consequences. If false the enumeration will not be kept current and may return recordIds for records that have been deleted or miss records that are added later. It may also return records out of order that have been modified after the enumeration was built. Note that any changes to records in the record store are accurately reflected when the record is later retrieved, either directly or through the enumeration. The thing that is risked by setting this parameter false is the filtering and sorting order of the enumeration when records are modified, added, or deleted.

Returns: an enumeration for traversing a set of records in the record store in an optionally specified order.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

See Also: `public void rebuild ()`

getLastModified()

`public long getLastModified ()`

Returns the last time the record store was modified, in the format used by System.currentTimeMillis().

Returns: the last time the record store was modified, in the format used by System.currentTimeMillis().

Throws: `RecordStoreNotOpenException` - if the record store is not open.

getName()

`public String getName ()`

Returns the name of this RecordStore.

Returns: the name of this RecordStore.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

getNextRecordID()

`public int getNextRecordID ()`

Returns the recordId of the next record to be added to the record store. This can be useful for setting up pseudo-relational relationships. That is, if you have two or more record stores whose records need to refer to one another, you can predetermine the recordIds of the records that will be created in one record store, before populating the fields and allocating the record in another record store. Note that the recordId returned is only valid while the record store remains open and until a call to `addRecord()`.

Returns: the recordId of the next record to be added to the record store.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

`RecordStoreException` - if a different record store-related exception occurred.

getNumRecords()

`public int getNumRecords ()`

Returns the number of records currently in the record store.

Returns: the number of records currently in the record store.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

getRecord(int)

`public byte[] getRecord (int recordId)`

Returns a copy of the data stored in the given record.

Parameters:
`recordId` - The ID of the record to use in this operation.

Returns: the data stored in the given record. Note that if the record has no data, this method will return null.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

`InvalidRecordIDException` - if the recordId is invalid.

`RecordStoreException` - if a general record store exception occurs.

getRecord(int, byte[], int)

```
public int getRecord (int recordId, byte[] buffer, int offset)
```

Returns the data stored in the given record.

Parameters:

`recordId` - The ID of the record to use in this operation.

`buffer` - The byte array in which to copy the data.

`offset` - The index into the buffer in which to start copying.

Returns: the number of bytes copied into the buffer, starting at index `offset`.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

`InvalidRecordIDException` - if the recordId is invalid.

`RecordStoreException` - if a general record store exception occurs.

`ArrayIndexOutOfBoundsException` - if the record is larger than the buffer supplied.

getRecordSize(int)

```
public int getRecordSize (int recordId)
```

Returns the size (in bytes) of the MIDlet data available in the given record.

Parameters:

`recordId` - The ID of the record to use in this operation.

Returns: the size (in bytes) of the MIDlet data available in the given record.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

`InvalidRecordIDException` - if the recordId is invalid.

`RecordStoreException` - if a general record store exception occurs.

getSize()

```
public int getSize ()
```

Returns the amount of space, in bytes, that the record store occupies. The size returned includes any overhead associated with the implementation, such as the data structures used to hold the state of the record store, etc.

Returns: the size of the record store in bytes.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

getSizeAvailable()

```
public int getSizeAvailable ()
```

Returns the amount of additional room (in bytes) available for this record store to grow. Note that this is not necessarily the amount of extra MIDlet-level data which can be stored, as implementations may store additional data structures with each record to support integration with native applications, synchronization, etc.

Returns: the amount of additional room (in bytes) available for this record store to grow.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

getVersion()

```
public int getVersion ()
```

Each time a record store is modified (record added, modified, deleted), it's *version* is incremented. This can be used by MIDlets to quickly tell if anything has been modified. The initial version number is implementation dependent. The increment is a positive integer greater than 0. The version number only increases as the RecordStore is updated.

Returns: the current record store version.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

listRecordStores()

```
public static String[] listRecordStores ()
```

Returns an array of the names of record stores owned by the MIDlet suite. Note that if the MIDlet suite does not have any record stores, this function will return NULL. The order of RecordStore names returned is implementation dependent.

Returns: an array of the names of record stores owned by the MIDlet suite. Note that if the MIDlet suite does not have any record stores, this function will return NULL.

openRecordStore(String, boolean)

```
public static RecordStore openRecordStore (String recordStoreName,
            boolean createIfNecessary)
```

Open (and possibly create) a record store associated with the given MIDlet suite. If this method is called by a MIDlet when the record store is already open by a MIDlet in the MIDlet suite, this method returns a reference to the same RecordStore object.

Parameters:

`recordStoreName` - The MIDlet suite unique name, not to exceed 32 characters, of the record store.

`createIfNecessary` - If true, the record store will be created if necessary.

Returns: The `RecordStore` object for the record store.

Throws: `RecordStoreException` - if a record store-related exception occurred.

`RecordStoreNotFoundException` - if the record store could not be found.

`RecordStoreFullException` - if the operation cannot be completed because the record store is full.

removeRecordListener(RecordListener)

```
public void removeRecordListener (RecordListener listener)
```

Removes the specified RecordListener. If the specified listener is not registered, this method does nothing.

Parameters:

`listener` - the RecordChangedListener.

setRecord(int, byte[], int, int)

```
public void setRecord (int recordId, byte[] newData, int offset, int numBytes)
```

Sets the data in the given record to that passed in. After this method returns, a call to `getRecord(int recordId)` will return an array of numBytes size containing the data supplied here.

Parameters:

`recordId` - The ID of the record to use in this operation.

`newData` - The new data to store in the record.

`offset` - The index into the data buffer of the first relevant byte for this record.

`numBytes` - The number of bytes of the data buffer to use for this record.

Throws: `RecordStoreNotOpenException` - if the record store is not open.

`InvalidRecordIDException` - if the recordId is invalid.

`RecordStoreException` - if a general record store exception occurs.

`RecordStoreFullException` - if the operation cannot be completed because the record store has no more room.

javax.microedition.rms

RecordStoreException

Syntax

```
public class RecordStoreException extends java.lang.Exception
```

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--javax.microedition.rms.RecordStoreException
```

Direct Known Subclasses: `InvalidRecordIDException`, `RecordStoreFullException`, `RecordStoreNotFoundException`, `RecordStoreNotOpenException`

Description

Thrown to indicate a general exception occurred in a record store operation.

Member Summary	

Constructors	`public RecordStoreException ()`
	`public RecordStoreException (String message)`

Inherited Member Summary
Methods inherited from class `Throwable`
`getMessage(), printStackTrace(), toString()`
Methods inherited from class `Object`
`equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)`

Constructors

RecordStoreException()

```
public RecordStoreException ()
```

Constructs a new `RecordStoreException` with no detail message.

RecordStoreException(String)

```
public RecordStoreException (String message)
```

Constructs a new `RecordStoreException` with the specified detail message.

Parameters:
`message` - the detail message.

javax.microedition.rms

RecordStoreFullException

Syntax

```
public class RecordStoreFullException extends RecordStoreException
```

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--RecordStoreException
                    |
                    +--javax.microedition.rms.RecordStoreFullException
```

Description

Thrown to indicate an operation could not be completed because the record store system storage is full.

Member Summary

Constructors

`public RecordStoreFullException ()`
`public RecordStoreFullException (String message)`

Inherited Member Summary

Methods inherited from class `Throwable`

`getMessage(), printStackTrace(), toString()`

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)`

Constructors

RecordStoreFullException()

```
public RecordStoreFullException ()
```

Constructs a new `RecordStoreFullException` with no detail message.

RecordStoreFullException(String)

```
public RecordStoreFullException (String message)
```

Constructs a new `RecordStoreFullException` with the specified detail message.

Parameters:
`message` - the detail message.

javax.microedition.rms
RecordStoreNotFoundException

Syntax

```
public class RecordStoreNotFoundException extends RecordStoreException

java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--RecordStoreException
                    |
                    +--javax.microedition.rms.RecordStoreNotFoundException
```

Description

Thrown to indicate an operation could not be completed because the record store could not be found.

Member Summary

Constructors

`public RecordStoreNotFoundException ()`
`public RecordStoreNotFoundException (String message)`

Inherited Member Summary

Methods inherited from class `Throwable`

`getMessage(), printStackTrace(), toString()`

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)`

Constructors

RecordStoreNotFoundException()

```
public RecordStoreNotFoundException ()
```

Constructs a new `RecordStoreNotFoundException` with no detail message.

RecordStoreNotFoundException(String)

```
public RecordStoreNotFoundException (String message)
```

Constructs a new `RecordStoreNotFoundException` with the specified detail message.

Parameters:
`message` - the detail message.

javax.microedition.rms

RecordStoreNotOpenException

Syntax

```
public class RecordStoreNotOpenException extends RecordStoreException
```

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--RecordStoreException
                    |
                    +--javax.microedition.rms.RecordStoreNotOpenException
```

Description

Thrown to indicate that an operation was attempted on a closed record store.

Member Summary	
Constructors	
	`public RecordStoreNotOpenException ()`
	`public RecordStoreNotOpenException (String message)`

Inherited Member Summary

Methods inherited from class `Throwable`

`getMessage(), printStackTrace(), toString()`

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)`

Constructors

RecordStoreNotOpenException()

```
public RecordStoreNotOpenException ()
```

Constructs a new `RecordStoreNotOpenException` with no detail message.

RecordStoreNotOpenException(String)

```
public RecordStoreNotOpenException (String message)
```

Constructs a new `RecordStoreNotOpenException` with the specified detail message.

Parameters:
`message` - the detail message.

Package

javax.microedition.midlet

Description

The MIDlet package defines Mobile Information Device Profile applications and the interactions between the application and the environment in which the application runs. An application of the Mobile Information Device Profile is a `MIDlet`.

The `MIDlet` lifecycle defines the protocol between a `MIDlet` and its environment through the following:

A simple well-defined state machine

A concise definition of the MIDlet's states

APIs to signal changes between the states

MIDlet Lifecycle Definitions

The following definitions are used in the `MIDlet` lifecycle:

application management software - a part of the device's software operating environment that manages `MIDlets`. It directs the `MIDlet` through state changes.

`MIDlet` - a MIDP application on the device. The `MIDlet` can signal the application management software about whether is it wants to run or has completed. A `MIDlet` has no knowledge of other `MIDlet`s through the `MIDlet` API.

`MIDlet States` - the states a `MIDlet` can have are defined by the transitions allowable through the `MIDlet` interface. More specific application states are known only to the application.

MIDlet States

The `MIDlet` state machine is designed to ensure that the behavior of an application is consistent and as close as possible to what device manufactures and users expect, specifically:

The perceived startup latency of an application should be very short.

It should be possible to put an application into a state where it is not active.

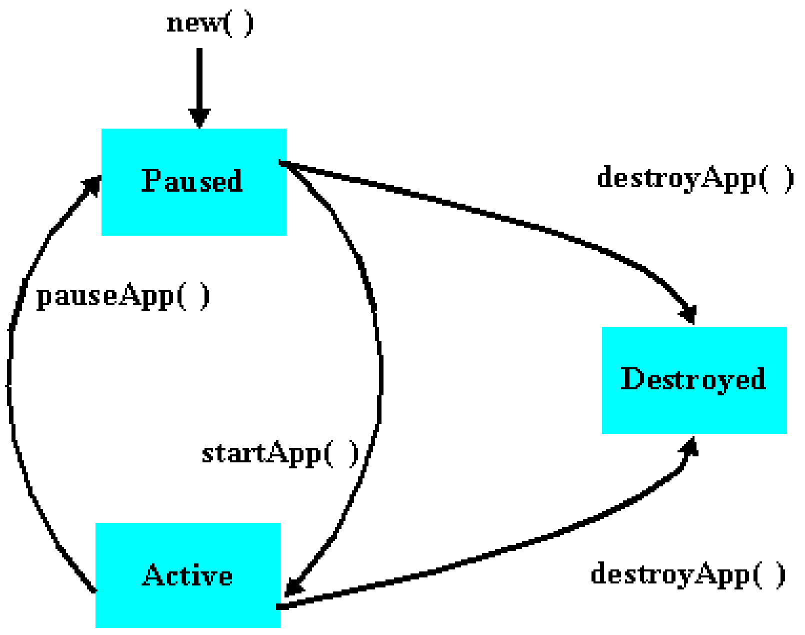
It should be possible to destroy an application at any time.

The valid states for `MIDlet`s are:

State Name	Description

Paused	The `MIDlet` is initialized and is quiescent. It should not be holding or using any shared resources. This state is entered: After the `MIDlet` has been created using `new`. The public no-argument constructor for the `MIDlet` is called and returns without throwing an exception. The application typically does little or no initialization in this step. If an exception occurs, the application immediately enters the Destroyed state and is discarded. From the Active state after the `MIDlet.pauseApp()` method returns successfully. From the Active state when the `MIDlet.notifyPaused()` method returns successfully to the `MIDlet`. From the Active state if `startApp` throws an `MIDletStateChangeException`.
Active	The `MIDlet` is functioning normally. This state is entered: Just prior to calling the `MIDlet.startApp()` method.
Destroyed	The `MIDlet` has released all of its resources and terminated. This state is entered: When the `MIDlet.destroyApp()` method returns except in the case when the `unconditional` argument is false and a `MIDletStateChangeException` is thrown. The `destroyApp()` method shall release all resources held and perform any necessary cleanup so it may be garbage collected. When the `MIDlet.notifyDestroyed()` method returns successfully to the application. The `MIDlet` must have performed the equivalent of the `MIDlet.destroyApp()` method before calling `MIDlet.notifyDestroyed()`. **Note:** This state is only entered once.

The states and transitions for a `MIDlet` are:



MIDlet Lifecycle Model

A typical sequence of `MIDlet` execution is:

Application Management Software	`MIDlet`
The application management software creates a new instance of a `MIDlet`.	The default (no argument) constructor for the `MIDlet` is called; it is in the Paused state.
The application management software has decided that it is an appropriate time for the `MIDlet` to run, so it calls the `MIDlet.startApp` method for it to enter the Active state.	The `MIDlet` acquires any resources it needs and begins to perform its service.
The application management software no longer needs the application be active, so it signals it to stop performing its service by calling the `MIDlet.pauseApp` method.	The `MIDlet` stops performing its service and might choose to release some resources it currently holds.
The application management software has determined that the `MIDlet` is no longer needed, or perhaps needs to make room for a higher priority application in memory, so it signals the `MIDlet` that it is a candidate to be destroyed by calling the `MIDlet.destroyApp` method.	If it has been designed to do so, the `MIDlet` saves state or user preferences and performs clean up.

MIDlet Interface

`pauseApp` - the `MIDlet` should release any temporary resources and become passive

`startApp` - the `MIDlet` should acquire any resources it needs and resume

`destroyApp` - the `MIDlet` should save any state and release all resources

`notifyDestroyed` - the `MIDlet` notifies the application management software that it has cleaned up and is done

`notifyPaused` - the `MIDlet` notifies the application management software that it has paused

`resumeRequest` - the `MIDlet` asks application management software to be started again

`getAppProperty` - gets a named property from the `MIDlet`

Application Implementation Notes

The application should take measures to avoid race conditions in the execution of the `MIDlet` methods. Each method may need to synchronize itself with the other methods avoid concurrency problems during state changes.

Example MIDlet Application

The example uses the MIDlet lifecycle to do a simple measurement of the speed of the Java Virtual Machine.

```
import javax.microedition.midlet.*;
/**
 * An example MIDlet runs a simple timing test
 * When it is started by the application management software it will
 * create a separate thread to do the test.
 * When it finishes it will notify the application management software
 * it is done.
 * Refer to the startApp, pauseApp, and destroyApp
 * methods so see how it handles each requested transition.
 */
public class MethodTimes extends MIDlet implements Runnable {
    // The state for the timing thread.
    Thread thread;
    /**
     * Start creates the thread to do the timing.
     * It should return immediately to keep the dispatcher
     * from hanging.
     */
    public void startApp() {
        thread = new Thread(this);
        thread.start();
    }
    /**
     * Pause signals the thread to stop by clearing the thread field.
     * If stopped before done with the iterations it will
     * be restarted from scratch later.
     */
    public void pauseApp() {
        thread = null;
    }
    /**
     * Destroy must cleanup everything.  The thread is signaled
     * to stop and no result is produced.
     */
    public void destroyApp(boolean unconditional) {
        thread = null;
    }
   /**
     * Run the timing test, measure how long it takes to
     * call a empty method 1000 times.
     * Terminate early if the current thread is no longer
     * the thread from the
     */
    public void run() {
        Thread curr = Thread.currentThread();  // Remember which thread is current
        long start = System.currentTimeMillis();
        for (int i = 0; i < 1000000 && thread == curr; i++) {
            empty();
        }
        long end = System.currentTimeMillis();
        // Check if timing was aborted, if so just exit
        // The rest of the application has already become quiescent.
        if (thread != curr) {
            return;
        }
        long millis = end - start;
        // Reporting the elapsed time is outside the scope of this example.
        // All done cleanup and quit
        destroyApp(true);
        notifyDestroyed();
    }
    /**
     * An Empty method.
     */
    void empty() {
    }
}
```

Class Summary

Classes

`MIDlet`	A `MIDLet` is a MID Profile application.
`MIDletStateChangeEx-ception`	Signals that a requested `MIDlet` state change failed.

javax.microedition.midlet

MIDlet

Syntax

```
public abstract class MIDlet

java.lang.Object
  |
  +--javax.microedition.midlet.MIDlet
```

Description

A `MIDLet` is a MID Profile application. The application must extend this class to allow the application management software to control the MIDlet and to be able to retrieve properties from the application descriptor and notify and request state changes. The methods of this class allow the application management software to create, start, pause, and destroy a MIDlet. A `MIDlet` is a set of classes designed to be run and controlled by the application management software via this interface. The states allow the application management software to manage the activities of multiple `MIDlets` within a runtime environment. It can select which `MIDlets` are active at a given time by starting and pausing them individually. The application management software maintains the state of the `MIDlet` and invokes methods on the `MIDlet` to change states. The `MIDlet` implements these methods to update its internal activities and resource usage as directed by the application management software. The `MIDlet` can initiate some state changes itself and notifies the application management software of those state changes by invoking the appropriate methods.

Note: The methods on this interface signal state changes. The state change is not considered complete until the state change method has returned. It is intended that these methods return quickly.

Member Summary

Constructors		
		`protected MIDlet ()`
Methods		
	`void`	`protected abstract void destroyApp (boolean unconditional)`
	`String`	`public final String getAppProperty (String key)`
	`void`	`public final void notifyDestroyed ()`
	`void`	`public final void notifyPaused ()`
	`void`	`protected abstract void pauseApp ()`
	`void`	`public final void resumeRequest ()`
	`void`	`protected abstract void startApp ()`

Inherited Member Summary

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

MIDlet()

`protected MIDlet ()`

Protected constructor for subclasses.

Methods

destroyApp(boolean)

`protected abstract void destroyApp (boolean unconditional)`

Signals the `MIDlet` to terminate and enter the *Destroyed* state. In the destroyed state the `MIDlet` must release all resources and save any persistent state. This method may be called from the *Paused* or *Active* states.

`MIDlet`s should perform any operations required before being terminated, such as releasing resources or saving preferences or state.

NOTE: The `MIDlet` can request that it not enter the *Destroyed* state by throwing an `MIDletState-ChangeException`. This is only a valid response if the `unconditional` flag is set to `false`. If it is `true` the `MIDlet` is assumed to be in the *Destroyed* state regardless of how this method terminates. If it is not an unconditional request, the `MIDlet` can signify that it wishes to stay in its current state by throwing the `MIDletStateChangeException`. This request may be honored and the `destroy()` method called again at a later time.

If a Runtime exception occurs during `destroyApp` then they are ignored and the MIDlet is put into the *Destroyed* state.

Parameters:
`unconditional` - If true when this method is called, the `MIDlet` must cleanup and release all resources. If false the `MIDlet` may throw `MIDletStateChangeException` to indicate it does not want to be destroyed at this time.

Throws: `<code>MIDletStateChangeException</code>` - is thrown if the `MIDlet` wishes to continue to execute (Not enter the *Destroyed* state). This exception is ignored if `unconditional` is equal to `true`.

`MIDletStateChangeException`

getAppProperty(String)

`public final String getAppProperty (String key)`

Provides a `MIDlet` with a mechanism to retrieve named properties from the application management software. The properties are retrieved from the combination of the application descriptor file and the manifest. If an attributes in the descriptor has the same name as an attribute in the manifest the value from the descriptor is used and the value from the manifest is ignored.

Parameters:
`key` - the name of the property

Returns: A string with the value of the property. `null` is returned if no value is available for the key.

Throws: `<code>NullPointerException</code>` - is thrown if key is `null`.

notifyDestroyed()

```
public final void notifyDestroyed ()
```

Used by an `MIDlet` to notify the application management software that it has entered into the *Destroyed* state. The application management software will not call the MIDlet's `destroyApp` method, and all resources held by the `MIDlet` will be considered eligible for reclamation. The `MIDlet` must have performed the same operations (clean up, releasing of resources etc.) it would have if the `MIDlet.destroyApp()` had been called.

notifyPaused()

```
public final void notifyPaused ()
```

Notifies the application management software that the MIDlet does not want to be active and has entered the *Paused* state. Invoking this method will have no effect if the `MIDlet` is destroyed, or if it has not yet been started.

It may be invoked by the `MIDlet` when it is in the *Active* state.

If a `MIDlet` calls `notifyPaused()`, in the future its `startApp()` method may be called make it active again, or its `destroyApp()` method may be called to request it to destroy itself.

pauseApp()

```
protected abstract void pauseApp ()
```

Signals the `MIDlet` to stop and enter the *Paused* state. In the *Paused* state the `MIDlet` must release shared resources and become quiescent. This method will only be called called when the `MIDlet` is in the *Active* state.

If a Runtime exception occurs during `pauseApp` the MIDlet will be destroyed immediately. Its `destroyApp` will be called allowing the MIDlet to cleanup.

resumeRequest()

```
public final void resumeRequest ()
```

Provides a `MIDlet` with a mechanism to indicate that it is interested in entering the *Active* state. Calls to this method can be used by the application management software to determine which applications to move to the *Active* state.

When the application management software decides to activate this application it will call the `startApp` method.

The application is generally in the *Paused* state when this is called. Even in the paused state the application may handle asynchronous events such as timers or callbacks.

startApp()

```
protected abstract void startApp ()
```

Signals the `MIDlet` that it has entered the *Active* state. In the *Active* state the `MIDlet` may hold resources. The method will only be called when the `MIDlet` is in the *Paused* state.

Two kinds of failures can prevent the service from starting, transient and non-transient. For transient failures the `MIDletStateChangeException` exception should be thrown. For non-transient failures the `notifyDestroyed` method should be called.

If a Runtime exception occurs during `startApp` the MIDlet will be destroyed immediately. Its `destroyApp` will be called allowing the MIDlet to cleanup.

Throws: `<code>MIDletStateChangeException</code>` - is thrown if the `MIDlet` cannot start now but might be able to start at a later time.

`MIDletStateChangeException`

javax.microedition.midlet

MIDletStateChangeException

Syntax

```
public class MIDletStateChangeException extends java.lang.Exception
```

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--javax.microedition.midlet.MIDletStateChangeException
```

Description

Signals that a requested `MIDlet` state change failed. This exception is thrown by the `MIDlet` in response to state change calls into the application via the `MIDlet` interface

See Also: `MIDlet`

Member Summary	
Constructors	
	`public MIDletStateChangeException ()`
	`public MIDletStateChangeException (String s)`

Inherited Member Summary

Methods inherited from class `Throwable`

`getMessage(), printStackTrace(), toString()`

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)`

Constructors

MIDletStateChangeException()

```
public MIDletStateChangeException ()
```

Constructs an exception with no specified detail message.

MIDletStateChangeException(String)

```
public MIDletStateChangeException (String s)
```

Constructs an exception with the specified detail message.

Parameters:
`s` - the detail message

Package
# javax.microedition.io

Description

MID Profile includes networking support based on the `GenericConnection` framework from the *Connected Limited Device Configuration*.

In addition to the `javax.microedition.io` classes specified in the *Connected Limited Device Configuration* the *Mobile Information Device Profile* includes the following interface for HTTP protocol access over the network.

- `javax.microedition.io.HttpConnection`

Class Summary

Interfaces

`HttpConnection`	This interface defines the necessary methods and constants for an HTTP connection.

javax.microedition.io

HttpConnection

Syntax

```
public interface HttpConnection extends javax.microedition.io.ContentConnection
```

All Superinterfaces: `Connection`, `ContentConnection`, `InputConnection`, `OutputConnection`, `StreamConnection`

Description

This interface defines the necessary methods and constants for an HTTP connection.

HTTP is a request-response protocol in which the parameters of request must be set before the request is sent. The connection exists in one of three states:

- Setup, in which the connection has not been made to the server.
- Connected, in which the connection has been made, request parameters have been sent and the response is expected.
- Closed, in which the connection has been closed and the methods will throw an IOException if called.

The following methods may be invoked only in the Setup state:

- `setRequestMethod`
- `setRequestProperty`

The transition from Setup to Connected is caused by any method that requires data to be sent to or received from the server.

The following methods cause the transition to the Connected state

- `openInputStream`
- `openOutputStream`
- `openDataInputStream`
- `openDataOutputStream`
- `getLength`
- `getType`
- `getEncoding`
- `getHeaderField`
- `getResponseCode`
- `getResponseMessage`
- `getHeaderFieldInt`
- `getHeaderFieldDate`
- `getExpiration`
- `getDate`
- `getLastModified`
- `getHeaderField`
- `getHeaderFieldKey`

The following methods may be invoked while the connection is open.

- `close`
- `getRequestMethod`
- `getRequestProperty`
- `getURL`
- `getProtocol`
- `getHost`
- `getFile`
- `getRef`
- `getPort`
- `getQuery`

Example using StreamConnection

Simple read of a url using StreamConnection. No HTTP specific behavior is needed or used.

Connector.open is used to open url and a StreamConnection is returned. From the StreamConnection the InputStream is opened. It is used to read every character until end of file (-1). If an exception is thrown the connection and stream are closed.

```
void getViaStreamConnection(String url) throws IOException {
    StreamConnection c = null;
    InputStream s = null;
    try {
        c = (StreamConnection)Connector.open(url);
        s = c.openInputStream();
        int ch;
        while ((ch = s.read()) != -1) {
            ...
        }
    } finally {
        if (s != null)
            s.close();
        if (c != null)
            c.close();
    }
}
```

Example using ContentConnection

Simple read of a url using ContentConnection. No HTTP specific behavior is needed or used.

Connector.open is used to open url and a ContentConnection is returned. The ContentConnection may be able to provide the length. If the length is available, it is used to read the data in bulk. From the ContentConnection the InputStream is opened. It is used to read every character until end of file (-1). If an exception is thrown the connection and stream are closed.

```
void getViaContentConnection(String url) throws IOException {
    ContentConnection c = null;
    InputStream is = null;
    try {
        c = (ContentConnection)Connector.open(url);
        int len = (int)c.getLength();
        if (len > 0) {
            is = c.openInputStream();
            byte[] data = new byte[len];
            int actual = is.read(data);
            ...
        } else {
            int ch;
            while ((ch = is.read()) != -1) {
                ...
            }
        }
    } finally {
        if (is != null)
            is.close();
        if (c != null)
            c.close();
    }
}
```

Example using HttpConnection

Read the HTTP headers and the data using HttpConnection.

Connector.open is used to open url and a HttpConnection is returned. The HTTP headers are read and processed. If the length is available, it is used to read the data in bulk. From the HttpConnection the InputStream is opened. It is used to read every character until end of file (-1). If an exception is thrown the connection and stream are closed.

```
void getViaHttpConnection(String url) throws IOException {
    HttpConnection c = null;
    InputStream is = null;
    try {
        c = (HttpConnection)Connector.open(url);

        // Getting the InputStream will open the connection
        // and read the HTTP headers. They are stored until
        // requested.
        is = c.openInputStream();

        // Get the ContentType
        String type = c.getType();

        // Get the length and process the data
        int len = (int)c.getLength();
        if (len > 0) {
            byte[] data = new byte[len];
            int actual = is.read(data);
            ...
        } else {
            int ch;
            while ((ch = is.read()) != -1) {
                ...
            }
        }
    } finally {
        if (is != null)
            is.close();
        if (c != null)
            c.close();
    }
}
```

Example using POST with HttpConnection

Post a request with some headers and content to the server and process the headers and content.

Connector.open is used to open url and a HttpConnection is returned. The request method is set to POST and request headers set. A simple command is written and flushed. The HTTP headers are read and processed. If the length is available, it is used to read the data in bulk. From the HttpConnection the InputStream is opened. It is used to read every character until end of file (-1). If an exception is thrown the connection and stream is closed.

```
void postViaHttpConnection(String url) throws IOException {
    HttpConnection c = null;
    InputStream is = null;
    OutputStream os = null;
    try {
        c = (HttpConnection)Connector.open(url);
        // Set the request method and headers
        c.setRequestMethod(HttpConnection.POST);
        c.setRequestProperty("If-Modified-Since",
            "29 Oct 1999 19:43:31 GMT");
        c.setRequestProperty("User-Agent",
            "Profile/MIDP-1.0 Configuration/CLDC-1.0");
        c.setRequestProperty("Content-Language", "en-US");
        // Getting the output stream may flush the headers
        os = c.openOutputStream();
        os.write("LIST games\n".getBytes());
        os.flush();                // Optional, openInputStream will flush
        // Opening the InputStream will open the connection
        // and read the HTTP headers. They are stored until
        // requested.
        is = c.openInputStream();
        // Get the ContentType
        String type = c.getType();
        processType(type);
        // Get the length and process the data
        int len = (int)c.getLength();
        if (len > 0) {
            byte[] data = new byte[len];
            int actual = is.read(data);
            process(data);
        } else {
            int ch;
            while ((ch = is.read()) != -1) {
                process((byte)ch);
            }
        }
    } finally {
        if (is != null)
            is.close();
        if (os != null)
            os.close();
        if (c != null)
            c.close();
    }
}
```

Simplified Stream Methods on Connector

Please note the following: The Connector class defines the following convenience methods for retrieving an input or output stream directly for a specified URL:

- `InputStream openDataInputStream(String url)`
- `DataInputStream openDataInputStream(String url)`
- `OutputStream openOutputStream(String url)`
- `DataOutputStream openDataOutputStream(String url)`

Please be aware that using these methods implies certain restrictions. You will not get a reference to the actual connection, but rather just references to the input or output stream of the connection. Not having a reference to the connection means that you will not be able to manipulate or query the connection directly. This in turn means that you will not be able to call any of the following methods:

- `getRequestMethod()`
- `setRequestMethod()`
- `getRequestProperty()`
- `setRequestProperty()`
- `getLength()`
- `getType()`
- `getEncoding()`
- `getHeaderField()`
- `getResponseCode()`
- `getResponseMessage()`
- `getHeaderFieldInt`
- `getHeaderFieldDate`
- `getExpiration`
- `getDate`
- `getLastModified`
- `getHeaderField`
- `getHeaderFieldKey`

Member Summary

Fields

`String`	`public static final String GET`
`String`	`public static final String HEAD`
`int`	`public static final int HTTP_ACCEPTED`
`int`	`public static final int HTTP_BAD_GATEWAY`
`int`	`public static final int HTTP_BAD_METHOD`
`int`	`public static final int HTTP_BAD_REQUEST`
`int`	`public static final int HTTP_CLIENT_TIMEOUT`
`int`	`public static final int HTTP_CONFLICT`
`int`	`public static final int HTTP_CREATED`
`int`	`public static final int HTTP_ENTITY_TOO_LARGE`
`int`	`public static final int HTTP_EXPECT_FAILED`
`int`	`public static final int HTTP_FORBIDDEN`
`int`	`public static final int HTTP_GATEWAY_TIMEOUT`
`int`	`public static final int HTTP_GONE`
`int`	`public static final int HTTP_INTERNAL_ERROR`
`int`	`public static final int HTTP_LENGTH_REQUIRED`
`int`	`public static final int HTTP_MOVED_PERM`
`int`	`public static final int HTTP_MOVED_TEMP`
`int`	`public static final int HTTP_MULT_CHOICE`
`int`	`public static final int HTTP_NO_CONTENT`
`int`	`public static final int HTTP_NOT_ACCEPTABLE`
`int`	`public static final int HTTP_NOT_AUTHORITATIVE`
`int`	`public static final int HTTP_NOT_FOUND`
`int`	`public static final int HTTP_NOT_IMPLEMENTED`
`int`	`public static final int HTTP_NOT_MODIFIED`

Member Summary

int	public static final int HTTP_OK
int	public static final int HTTP_PARTIAL
int	public static final int HTTP_PAYMENT_REQUIRED
int	public static final int HTTP_PRECON_FAILED
int	public static final int HTTP_PROXY_AUTH
int	public static final int HTTP_REQ_TOO_LONG
int	public static final int HTTP_RESET
int	public static final int HTTP_SEE_OTHER
int	public static final int HTTP_TEMP_REDIRECT
int	public static final int HTTP_UNAUTHORIZED
int	public static final int HTTP_UNAVAILABLE
int	public static final int HTTP_UNSUPPORTED_RANGE
int	public static final int HTTP_UNSUPPORTED_TYPE
int	public static final int HTTP_USE_PROXY
int	public static final int HTTP_VERSION
String	public static final String POST
Methods	
long	public long getDate ()
long	public long getExpiration ()
String	public String getFile ()
String	public String getHeaderField (int n)
String	public String getHeaderField (String name)
long	public long getHeaderFieldDate (String name, long def)
int	public int getHeaderFieldInt (String name, int def)
String	public String getHeaderFieldKey (int n)
String	public String getHost ()
long	public long getLastModified ()
int	public int getPort ()
String	public String getProtocol ()
String	public String getQuery ()
String	public String getRef ()
String	public String getRequestMethod ()
String	public String getRequestProperty (String key)
int	public int getResponseCode ()
String	public String getResponseMessage ()
String	public String getURL ()
void	public void setRequestMethod (String method)
void	public void setRequestProperty (String key, String value)

Inherited Member Summary

Methods inherited from interface `ContentConnection`

`getEncoding(), getLength(), getType()`

Methods inherited from interface `InputConnection`

`openDataInputStream(), openInputStream()`

Methods inherited from interface `Connection`

Inherited Member Summary
`close()`
Methods inherited from interface `OutputConnection`
`openDataOutputStream(), openOutputStream()`

Fields

GET

```
public static final String GET
```

HTTP Get method

HEAD

```
public static final String HEAD
```

HTTP Head method

HTTP_ACCEPTED

```
public static final int HTTP_ACCEPTED
```

202: The request has been accepted for processing, but the processing has not been completed.

HTTP_BAD_GATEWAY

```
public static final int HTTP_BAD_GATEWAY
```

502: The server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request.

HTTP_BAD_METHOD

```
public static final int HTTP_BAD_METHOD
```

405: The method specified in the Request-Line is not allowed for the resource identified by the Request-URI.

HTTP_BAD_REQUEST

```
public static final int HTTP_BAD_REQUEST
```

400: The request could not be understood by the server due to malformed syntax.

HTTP_CLIENT_TIMEOUT

```
public static final int HTTP_CLIENT_TIMEOUT
```

408: The client did not produce a request within the time that the server was prepared to wait. The client MAY repeat the request without modifications at any later time.

HTTP_CONFLICT

```
public static final int HTTP_CONFLICT
```

409: The request could not be completed due to a conflict with the current state of the resource.

HTTP_CREATED

```
public static final int HTTP_CREATED
```

201: The request has been fulfilled and resulted in a new resource being created.

HTTP_ENTITY_TOO_LARGE

```
public static final int HTTP_ENTITY_TOO_LARGE
```

413: The server is refusing to process a request because the request entity is larger than the server is willing or able to process.

HTTP_EXPECT_FAILED

```
public static final int HTTP_EXPECT_FAILED
```

417: The expectation given in an Expect request-header field could not be met by this server, or, if the server is a proxy, the server has unambiguous evidence that the request could not be met by the next-hop server.

HTTP_FORBIDDEN

```
public static final int HTTP_FORBIDDEN
```

403: The server understood the request, but is refusing to fulfill it. Authorization will not help and the request SHOULD NOT be repeated.

HTTP_GATEWAY_TIMEOUT

```
public static final int HTTP_GATEWAY_TIMEOUT
```

504: The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server specified by the URI or some other auxiliary server it needed to access in attempting to complete the request.

HTTP_GONE

```
public static final int HTTP_GONE
```

410: The requested resource is no longer available at the server and no forwarding address is known.

HTTP_INTERNAL_ERROR

```
public static final int HTTP_INTERNAL_ERROR
```

500: The server encountered an unexpected condition which prevented it from fulfilling the request.

HTTP_LENGTH_REQUIRED

```
public static final int HTTP_LENGTH_REQUIRED
```

411: The server refuses to accept the request without a defined Content- Length.

HTTP_MOVED_PERM

```
public static final int HTTP_MOVED_PERM
```

301: The requested resource has been assigned a new permanent URI and any future references to this resource SHOULD use one of the returned URIs.

HTTP_MOVED_TEMP

```
public static final int HTTP_MOVED_TEMP
```

302: The requested resource resides temporarily under a different URI.

HTTP_MULT_CHOICE

```
public static final int HTTP_MULT_CHOICE
```

300: The requested resource corresponds to any one of a set of representations, each with its own specific location, and agent- driven negotiation information is being provided so that the user (or user agent) can select a preferred representation and redirect its request to that location.

HTTP_NO_CONTENT

```
public static final int HTTP_NO_CONTENT
```

204: The server has fulfilled the request but does not need to return an entity-body, and might want to return updated meta-information.

HTTP_NOT_ACCEPTABLE

```
public static final int HTTP_NOT_ACCEPTABLE
```

406: The resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headers sent in the request.

HTTP_NOT_AUTHORITATIVE

```
public static final int HTTP_NOT_AUTHORITATIVE
```

203: The returned meta-information in the entity-header is not the definitive set as available from the origin server.

HTTP_NOT_FOUND

```
public static final int HTTP_NOT_FOUND
```

404: The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent.

HTTP_NOT_IMPLEMENTED

```
public static final int HTTP_NOT_IMPLEMENTED
```

501: The server does not support the functionality required to fulfill the request.

HTTP_NOT_MODIFIED

```
public static final int HTTP_NOT_MODIFIED
```

304: If the client has performed a conditional GET request and access is allowed, but the document has not been modified, the server SHOULD respond with this status code.

HTTP_OK

```
public static final int HTTP_OK
```

200: The request has succeeded.

HTTP_PARTIAL

```
public static final int HTTP_PARTIAL
```

206: The server has fulfilled the partial GET request for the resource.

HTTP_PAYMENT_REQUIRED

```
public static final int HTTP_PAYMENT_REQUIRED
```

402: This code is reserved for future use.

HTTP_PRECON_FAILED

```
public static final int HTTP_PRECON_FAILED
```

412: The precondition given in one or more of the request-header fields evaluated to false when it was tested on the server.

HTTP_PROXY_AUTH

```
public static final int HTTP_PROXY_AUTH
```

407: This code is similar to 401 (Unauthorized), but indicates that the client must first authenticate itself with the proxy.

HTTP_REQ_TOO_LONG

```
public static final int HTTP_REQ_TOO_LONG
```

414: The server is refusing to service the request because the Request-URI is longer than the server is willing to interpret.

HTTP_RESET

```
public static final int HTTP_RESET
```

205: The server has fulfilled the request and the user agent SHOULD reset the document view which caused the request to be sent.

HTTP_SEE_OTHER

```
public static final int HTTP_SEE_OTHER
```

303: The response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource.

HTTP_TEMP_REDIRECT

```
public static final int HTTP_TEMP_REDIRECT
```

307: The requested resource resides temporarily under a different URI.

HTTP_UNAUTHORIZED

```
public static final int HTTP_UNAUTHORIZED
```

401: The request requires user authentication. The response MUST include a WWW-Authenticate header field containing a challenge applicable to the requested resource.

HTTP_UNAVAILABLE

```
public static final int HTTP_UNAVAILABLE
```

503: The server is currently unable to handle the request due to a temporary overloading or maintenance of the server.

HTTP_UNSUPPORTED_RANGE

```
public static final int HTTP_UNSUPPORTED_RANGE
```

416: A server SHOULD return a response with this status code if a request included a Range request-header field , and none of the range-specifier values in this field overlap the current extent of the selected resource, and the request did not include an If-Range request-header field.

HTTP_UNSUPPORTED_TYPE

```
public static final int HTTP_UNSUPPORTED_TYPE
```

415: The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.

HTTP_USE_PROXY

```
public static final int HTTP_USE_PROXY
```

305: The requested resource MUST be accessed through the proxy given by the Location field.

HTTP_VERSION

```
public static final int HTTP_VERSION
```

505: The server does not support, or refuses to support, the HTTP protocol version that was used in the request message.

POST

```
public static final String POST
```

HTTP Post method

Methods

getDate()

```
public long getDate ()
```

Returns the value of the `date` header field.

Returns: the sending date of the resource that the URL references, or `0` if not known. The value returned is the number of milliseconds since January 1, 1970 GMT.

Throws: `IOException` - if an error occurred connecting to the server.

getExpiration()

`public long getExpiration ()`

Returns the value of the `expires` header field.

Returns: the expiration date of the resource that this URL references, or 0 if not known. The value is the number of milliseconds since January 1, 1970 GMT.

Throws: `IOException` - if an error occurred connecting to the server.

getFile()

`public String getFile ()`

Returns the file portion of the URL of this `HttpConnection`.

Returns: the file portion of the URL of this `HttpConnection`. `null` is returned if there is no file.

getHeaderField(int)

`public String getHeaderField (int n)`

Gets a header field value by index.

Parameters:
`n` - the index of the header field

Returns: the value of the nth header field or `null` if the array index is out of range. An empty String is returned if the field does not have a value.

Throws: `IOException` - if an error occurred connecting to the server.

getHeaderField(String)

`public String getHeaderField (String name)`

Returns the value of the named header field.

Parameters:
`name` - of a header field.

Returns: the value of the named header field, or `null` if there is no such field in the header.

Throws: `IOException` - if an error occurred connecting to the server.

getHeaderFieldDate(String, long)

`public long getHeaderFieldDate (String name, long def)`

Returns the value of the named field parsed as date. The result is the number of milliseconds since January 1, 1970 GMT represented by the named field.

This form of `getHeaderField` exists because some connection types (e.g., `http-ng`) have pre-parsed headers. Classes for that connection type can override this method and short-circuit the parsing.

Parameters:
`name` - the name of the header field.

`def` - a default value.

Returns: the value of the field, parsed as a date. The value of the `def` argument is returned if the field is missing or malformed.

Throws: `IOException` - if an error occurred connecting to the server.

---

getHeaderFieldInt(String, int)

```
public int getHeaderFieldInt (String name, int def)
```

Returns the value of the named field parsed as a number.

This form of `getHeaderField` exists because some connection types (e.g., `http-ng`) have pre-parsed headers. Classes for that connection type can override this method and short-circuit the parsing.

Parameters:
`name` - the name of the header field.

`def` - the default value.

Returns: the value of the named field, parsed as an integer. The `def` value is returned if the field is missing or malformed.

Throws: `IOException` - if an error occurred connecting to the server.

---

getHeaderFieldKey(int)

```
public String getHeaderFieldKey (int n)
```

Gets a header field key by index.

Parameters:
`n` - the index of the header field

Returns: the key of the nth header field or `null` if the array index is out of range.

Throws: `IOException` - if an error occurred connecting to the server.

---

getHost()

```
public String getHost ()
```

Returns the host information of the URL of this `HttpConnection`. e.g. host name or IPv4 address

Returns: the host information of the URL of this `HttpConnection`. `null` is returned if there is no host.

---

getLastModified()

```
public long getLastModified ()
```

Returns the value of the `last-modified` header field. The result is the number of milliseconds since January 1, 1970 GMT.

Returns: the date the resource referenced by this `HttpConnection` was last modified, or 0 if not known.

Throws: `IOException` - if an error occurred connecting to the server.

getPort()

`public int getPort ()`

Returns the network port number of the URL for this `HttpConnection`.

Returns: the network port number of the URL for this `HttpConnection`. The default HTTP port number (80) is returned if there was no port number in the string passed to `Connector.open`.

getProtocol()

`public String getProtocol ()`

Returns the protocol name of the URL of this `HttpConnection`. e.g., http or https

Returns: the protocol of the URL of this `HttpConnection`.

getQuery()

`public String getQuery ()`

Returns the query portion of the URL of this `HttpConnection`. RFC2396 defines the query component as the text after the last question-mark (?) character in the URL.

Returns: the query portion of the URL of this `HttpConnection`. `null` is returned if there is no value.

getRef()

`public String getRef ()`

Returns the ref portion of the URL of this `HttpConnection`. RFC2396 specifies the optional fragment identifier as the the text after the crosshatch (#) character in the URL. This information may be used by the user agent as additional reference information after the resource is successfully retrieved. The format and interpretation of the fragment identifier is dependent on the media type[RFC2046] of the retrieved information.

Returns: the ref portion of the URL of this `HttpConnection`. `null` is returned if there is no value.

getRequestMethod()

`public String getRequestMethod ()`

Get the current request method. e.g. HEAD, GET, POST The default value is GET.

Returns: the HTTP request method

getRequestProperty(String)

`public String getRequestProperty (String key)`

Returns the value of the named general request property for this connection.

Parameters:
`key` - the keyword by which the request property is known (e.g., "accept").

Returns: the value of the named general request property for this connection. If there is no key with the specified name then `null` is returned.

getResponseCode()

```
public int getResponseCode ()
```

Returns the HTTP response status code. It parses responses like:

```
HTTP/1.0 200 OK
HTTP/1.0 401 Unauthorized
```

and extracts the ints 200 and 401 respectively. from the response (i.e., the response is not valid HTTP).

Returns: the HTTP Status-Code or -1 if no status code can be discerned.

Throws: `IOException` - if an error occurred connecting to the server.

getResponseMessage()

```
public String getResponseMessage ()
```

Gets the HTTP response message, if any, returned along with the response code from a server. From responses like:

```
HTTP/1.0 200 OK
HTTP/1.0 404 Not Found
```

Extracts the Strings "OK" and "Not Found" respectively. Returns null if none could be discerned from the responses (the result was not valid HTTP).

Returns: the HTTP response message, or `null`

Throws: `IOException` - if an error occurred connecting to the server.

getURL()

```
public String getURL ()
```

Return a string representation of the URL for this connection.

Returns: the string representation of the URL for this connection.

setRequestMethod(String)

```
public void setRequestMethod (String method)
```

Set the method for the URL request, one of:

- GET
- POST
- HEAD

are legal, subject to protocol restrictions. The default method is GET.

Parameters:
`method` - the HTTP method

Throws: `IOException` - if the method cannot be reset or if the requested method isn't valid for HTTP.

setRequestProperty(String, String)

```
public void setRequestProperty (String key, String value)
```

Sets the general request property. If a property with the key already exists, overwrite its value with the new value.

NOTE: HTTP requires all request properties which can legally have multiple instances with the same key to use a comma-separated list syntax which enables multiple properties to be appended into a single property.

Parameters:
`key` - the keyword by which the request is known (e.g., "`accept`").

`value` - the value associated with it.

Throws: `IOException` - is thrown if the connection is in the connected state.

Package

javax.microedition.lcdui

Description

The UI API provides a set of features for implementation of user interfaces for MIDP applications.

For more information see Chapter 9 of MIDP specification.

Screen-based approach

The central abstraction of the MIDP's UI is that of a screen. A screen is an object that encapsulates device-specific graphics rendering user input. Only one screen may be visible at the time, and the user can only traverse through the items on that screen. The screen takes care of all events that occur as the user navigates in the screen, with only higher-level events being passed on to the application.

The application can switch the screens by calling `public void setCurrent (Displayable nextDisplayable)` .

It is recommended that the screens are simple and contain as few UI components as reasonable.

Two-layer approach

The MIDP UI is logically composed of two APIs: the high-level API and the low-level API.

The *high-level API* is designed for business applications whose client parts run on MIDs. For these applications, portability across devices is important. In order to achieve this portability, the high-level API employs a high level of abstraction and provides very little control over look and feel. This abstraction is further manifested in the following three ways:

The actual drawing to the MID's display is performed by the implementation. Applications do not define the visual appearance (e.g. shape, color, font, etc.) of the components. Navigation, scrolling, and other primitive interaction is encapsulated by the implementation, and the application is not aware of these interactions. Applications can not access concrete input devices like specific individual keys.

In other words, when using the high-level API, it is assumed that the underlying implementation will do the necessary adaptation to device's hardware and native UI style.

The screens implementing the high-level API are the subclasses of `Screen` .

The *low-level API*, on the other hand, provides quite little abstraction. This API is designed for applications that need precise placement and control of graphic elements and access to low-level input events. Some applications also need to access special, device-specific features. A typical example of such an application would be a game. Using the low-level API, an application can:

Have full control of what is drawn on the display. Listen for primitive events like key presses and releases. Access concrete keys and other input devices

Classes `Canvas` and `Graphics` implement the low-level API.

Applications that program to the low-level API are not guaranteed to be portable, since the low-level API provides means to access details that are specific to a particular device. If the application does not use these features, the applications will portable and it is recommended that the applications stick to the platform-independent part of the low-level API when ever possible. This means that the applications should not directly assume any other keys than defined in class Canvas, and should not blindly trust on any specific screen size. Rather, the application game-event mechanism should be used instead of referring to concrete keys, and application should ask and adjust to the size of the display.

Class Summary

Interfaces	
`Choice`	Choice defines an API for a user interface components implementing selection from predefined number of choices.
`CommandListener`	This interface is used by applications which need to receive high-level events from the implementation.
`ItemStateListener`	This interface is used by applications which need to receive events that indicate changes in the internal state of the interactive items within a `Form` screen.
Classes	
`Alert`	An alert is a screen that shows data to the user and waits for a certain period of time before proceeding to the next screen.
`AlertType`	The AlertType provides an indication of the nature of alerts.
`Canvas`	The Canvas class is a base class for writing applications that need to handle low-level events and to issue graphics calls for drawing to the display.
`ChoiceGroup`	A ChoiceGroup is a group of selectable elements intended to be placed within a `Form` .
`Command`	The Command class is a construct that encapsulates the semantic information of an action.
`DateField`	A DateField is an editable component for presenting date and time (calendar) information that may be placed into a Form.
`Display`	Display represents the manager of the display and input devices of the system.
`Displayable`	An object that has the capability of being placed on the display.
`Font`	The Font class represents fonts and font metrics.
`Form`	A Form is a Screen that contains an arbitrary mixture of items: images, read-only text fields, editable text fields, editable date fields, gauges, and choice groups.
`Gauge`	The Gauge class implements a bar graph display of a value intended for use in a form.
`Graphics`	Provides simple 2D geometric rendering capability.
`Image`	The Image class is used to hold graphical image data.
`ImageItem`	A class that provides layout control when Image objects are added to a `Form` or to an `Alert` .
`Item`	A superclass for components that can be added to a `Form` and `Alert` .
`List`	The List class is a Screen containing list of choices.
`Screen`	The common superclass of all high-level user interface classes.
`StringItem`	An item that can contain a string.
`TextBox`	The TextBox class is a Screen that allows the user to enter and edit text.
`TextField`	A TextField is an editable text component that may be placed into a `Form` .
`Ticker`	Implements a "ticker-tape," a piece of text that runs continuously across the display.

javax.microedition.lcdui

Alert

Syntax

`public class Alert extends Screen`

```
Displayable
  |
  +--Screen
        |
        +--javax.microedition.lcdui.Alert
```

Description

An alert is a screen that shows data to the user and waits for a certain period of time before proceeding to the next screen. An alert is an ordinary screen that can contain text (String) and image, and which handles events like other screens.

The intended use of Alert is to inform the user about errors and other exceptional conditions.

The application can set the alert time to be infinity with `setTimeout(Alert.FOREVER)` in which case the Alert is considered to be *modal* and the implementation provide a feature that allows the user to "dismiss" the alert, whereupon the next screen is displayed as if the timeout had expired immediately.

If an application specifies an alert to be of a timed variety *and* gives it too much content such that it must scroll, then it automatically becomes a modal alert.

An alert may have an `AlertType` associated with it to provide an indication of the nature of the alert. The implementation may use this type to play an appropriate sound when the Alert is presented to the user. See `public boolean playSound (Display display)`.

Alerts do not accept application-defined commands.

If the Alert is visible on the display when changes to its contents are requested by the application, the changes take place automatically. That is, applications need not take any special action to refresh a Alert's display after its contents have been modified.

See Also: `AlertType`

Member Summary

Fields		
	`int`	`public static final int FOREVER`
Constructors		
		`public Alert (java.lang.String title)`
		`public Alert (java.lang.String title, java.lang.String alertText, Image alertImage, AlertType alertType)`
Methods		
	`void`	`public void addCommand (Command cmd)`
	`int`	`public int getDefaultTimeout ()`
	`Image`	`public Image getImage ()`
	`String`	`public java.lang.String getString ()`
	`int`	`public int getTimeout ()`
	`AlertType`	`public AlertType getType ()`

Member Summary

void	public void setCommandListener (CommandListener l)
void	public void setImage (Image img)
void	public void setString (java.lang.String str)
void	public void setTimeout (int time)
void	public void setType (AlertType type)

Inherited Member Summary

Methods inherited from class `Screen`

`public Ticker getTicker ()`, `public java.lang.String getTitle ()`, `public void setTicker (Ticker ticker)`, `public void setTitle (java.lang.String s)`

Methods inherited from class `Displayable`

`public boolean isShown ()`, `public void removeCommand (Command cmd)`

Fields

FOREVER

```
public static final int FOREVER
```

FOREVER indicates that an Alert is kept visible until the user dismisses it. It is used as a value for the parameter to `public void setTimeout (int time)` to indicate that the alert is modal. Instead of waiting for a specified period of time, a modal Alert will wait for the user to take some explicit action, such as pressing a button, before proceeding to the next screen.

Value -2 is assigned to FOREVER.

Constructors

Alert(String)

```
public Alert (java.lang.String title)
```

Constructs a new, empty Alert object with the given title. If null is passed, the Alert will have no title. Calling this constructor is equivalent to calling

```
Alert(title, null, null, null)
```

Parameters:
`title` - the title string, or null

See Also: `public Alert (java.lang.String title, java.lang.String alertText, Image alertImage, AlertType alertType)`

Alert(String, String, Image, AlertType)

```
public Alert (java.lang.String title, java.lang.String alertText, Image alertImage,
            AlertType alertType)
```

Constructs a new Alert object with the given title, content string and image, and alert type. The layout of the contents is implementation dependent. The timeout value of this new alert is the same value that is returned by getDefaultTimeout(). If an image is provided it must be immutable. The handling and behavior of specific AlertTypes is described in `AlertType` . Null is allowed as the value of the alertType parameter and indicates that the Alert is not to have a specific alert type.

Parameters:

`title` - the title string, or null if there is no title

`alertText` - the string contents, or null if there is no string

`alertImage` - the image contents, or null if there is no image

`alertType` - the type of the Alert, or null if the Alert has no specific type

Throws: `IllegalArgumentException` - if the image is mutable

Methods

addCommand(Command)

```
public void addCommand (Command cmd)
```

Commands are not allowed on Alerts, so this method will always throw IllegalStateException whenever it is called.

Overrides: `public void addCommand (Command cmd)` in class `Displayable`

Parameters:

`cmd` - the Command

Throws: `IllegalStateException` - always

getDefaultTimeout()

```
public int getDefaultTimeout ()
```

Gets the default time for showing an Alert. This is either a positive value, which indicates a time in milliseconds, or the special value FOREVER, which indicates that Alerts are modal by default. The value returned will vary across implementations and is presumably tailored to be suitable for each.

Returns: default timeout in milliseconds, or FOREVER

getImage()

```
public Image getImage ()
```

Gets the Image used in the Alert.

Returns: the Alert's image, or null if there is no image

getString()

`public java.lang.String getString ()`

Gets the text string used in the Alert.

Returns: the Alert's text string, or null if there is no text

getTimeout()

`public int getTimeout ()`

Gets the time this Alert will be shown. This is either a positive value, which indicates a time in milliseconds, or the special value FOREVER, which indicates that this Alert is modal.

Returns: timeout in milliseconds, or FOREVER

getType()

`public AlertType getType ()`

Gets the type of the Alert.

Returns: a reference to an instance of AlertType, or null if the Alert has no specific type

setCommandListener(CommandListener)

`public void setCommandListener (CommandListener l)`

Listeners are not allowed on Alerts, so this method will always throw IllegalStateException whenever it is called.

Overrides: `public void setCommandListener (CommandListener l)` in class `Displayable`

Parameters:
`l` - the Listener

Throws: `IllegalStateException` - always

setImage(Image)

`public void setImage (Image img)`

Sets the Image used in the Alert.

Parameters:
`img` - the Alert's image, or null if there is no image

Throws: `IllegalArgumentException` - if img is mutable

setString(String)

`public void setString (java.lang.String str)`

Sets the text string used in the Alert.

Parameters:
`str` - the Alert's text string, or null if there is no text

---

setTimeout(int)

```
public void setTimeout (int time)
```

Set the time for which the Alert is to be shown. This must either be a positive time value in milliseconds, or the special value FOREVER.

Parameters:
`time` - timeout in milliseconds, or FOREVER

Throws: `IllegalArgumentException` - if time is not positive and is not FOREVER

---

setType(AlertType)

```
public void setType (AlertType type)
```

Sets the type of the Alert. The handling and behavior of specific AlertTypes is described in `AlertType` .

Parameters:
`type` - an AlertType, or `null` if the Alert has no specific type

javax.microedition.lcdui
AlertType

Syntax

```
public class AlertType
```

javax.microedition.lcdui.AlertType

Description

The AlertType provides an indication of the nature of alerts. Alerts are used by an application to present various kinds of information to the user. An AlertType may be used to directly signal the user without changing the current Displayable. The `playSound` method can be used to spontaneously generate a sound to alert the user. For example, a game using a Canvas can use `playSound` to indicate success or progress. The predefined types are `INFO`, `WARNING`, `ERROR`, `ALARM`, and `CONFIRMATION`.

See Also: `Alert`

Member Summary		
Fields		
	AlertType	public static final AlertType ALARM
	AlertType	public static final AlertType CONFIRMATION
	AlertType	public static final AlertType ERROR
	AlertType	public static final AlertType INFO
	AlertType	public static final AlertType WARNING
Constructors		
		protected AlertType ()
Methods		
	boolean	public boolean playSound (Display display)

Fields

ALARM

```
public static final AlertType ALARM
```

An ALARM AlertType is a hint to alert the user to an event for which the user has previously requested to be notified. For example, the message might say, "Staff meeting in five minutes."

CONFIRMATION

```
public static final AlertType CONFIRMATION
```

A CONFIRMATION AlertType is a hint to confirm user actions. For example, "Saved!" might be shown to indicate that a Save operation has completed.

ERROR

`public static final AlertType ERROR`

An ERROR AlertType is a hint to alert the user to an erroneous operation. For example, an error alert might show the message, "There is not enough room to install the application."

INFO

`public static final AlertType INFO`

An INFO AlertType typically provides non-threatening information to the user. For example, a simple splash screen might be an INFO AlertType.

WARNING

`public static final AlertType WARNING`

A WARNING AlertType is a hint to warn the user of a potentially dangerous operation. For example, the warning message may contain the message, "Warning: this operation will erase your data."

Constructors

AlertType()

`protected AlertType ()`

Protected constructor for subclasses.

Methods

playSound(Display)

`public boolean playSound (Display display)`

Alert the user by playing the sound for this AlertType. The AlertType instance is used as a hint by the device to generate an appropriate sound. Instances other than those predefined above may be ignored. The actual sound made by the device, if any, is determined by the device. The device may ignore the request, use the same sound for several AlertTypes or use any other means suitable to alert the user.

Parameters:
`display` - to which the AlertType's sound should be played.

Returns: `true` if the user was alerted, `false` otherwise.

Throws: `NullPointerException` - if `display` is `null`

javax.microedition.lcdui

Canvas

Syntax

`public abstract class Canvas extends Displayable`

```
Displayable
  |
  +--javax.microedition.lcdui.Canvas
```

Description

The Canvas class is a base class for writing applications that need to handle low-level events and to issue graphics calls for drawing to the display. Game applications will likely make heavy use of the Canvas class. From an application development perspective, the Canvas class is interchangeable with standard Screen classes, so an application may mix and match Canvas with high-level screens as needed. For example, a List screen may be used to select the track for a racing game, and a Canvas subclass would implement the actual game.

The Canvas provides the developer with methods to handle game actions, key events, and pointer events (if supported by the device). Methods are also provided to identify the device's capabilities and keyboard mapping. The key events are reported with respect to *key codes*, which are directly bound to concrete keys on the device, use of which may hinder portability. Portable applications should use game actions instead of key codes.

Like other subclasses of Displayable, the Canvas class allows the application to register a listener for commands. Unlike other Displayables, however, the Canvas class requires applications to subclass it in order to use it. The paint() method is declared `abstract`, and so the application *must* provide an implementation in its subclass. Other event-reporting methods are not declared `abstract,` and their default implementations are empty (that is, they do nothing). This allows the application to override only the methods that report events in which the application has interest.

This is in contrast to the `Screen` classes, which allow the application to define listeners and to register them with instances of the Screen classes. This style is not used for the Canvas class, because several new listener interfaces would need to be created, one for each kind of event that might be delivered. An alternative would be to have fewer listener interfaces, but this would require listeners to filter out events in which they had no interest.

Key Events

Applications receive keystroke events in which the individual keys are named within a space of *key codes*. Every key for which events are reported to MIDP applications is assigned a key code. The key code values are unique for each hardware key unless two keys are obvious synonyms for each other. MIDP defines the following key codes: `public static final int KEY_NUM0`, `public static final int KEY_NUM1`, `public static final int KEY_NUM2`, `public static final int KEY_NUM3`, `public static final int KEY_NUM4`, `public static final int KEY_NUM5`, `public static final int KEY_NUM6`, `public static final int KEY_NUM7`, `public static final int KEY_NUM8`, `public static final int KEY_NUM9`, `public static final int KEY_STAR`, and `public static final int KEY_POUND`. (These key codes correspond to keys on a ITU-T standard telephone keypad.) Other keys may be present on the keyboard, and they will generally have key codes distinct from those list above. In order to guarantee portability, applications should use only the standard key codes.

The standard key codes' values are equal to the Unicode encoding for the character that represents the key. If the device includes any other keys that have an obvious correspondence to a Unicode character, their key values should equal the Unicode encoding for that character. For keys that have no corresponding Unicode character, the implementation must use negative values. Zero is defined to be an invalid key code. It is thus possible for an application to convert a keyCode into a Unicode character using the following code:

```
if (keyCode > 0) {
    char ch = (char)keyCode;
    // ...
}
```

This technique is useful only in certain limited cases. In particular, it is not sufficient for full textual input, because it does not handle upper and lower case, keyboard shift states, and characters that require more than one keystroke to enter. For textual input, applications should always use `TextBox` or `TextField` objects.

It is sometimes useful to find the *name* of a key in order to display a message about this key. In this case the application may use the `public java.lang.String getKeyName (int keyCode)` method to find a key's name.

Game Actions

Portable applications that need arrow key events and gaming-related events should use *game actions* in preference to key codes and key names. MIDP defines the following game actions: `public static final int UP`, `public static final int DOWN`, `public static final int LEFT`, `public static final int RIGHT`, `public static final int FIRE`, `public static final int GAME_A`, `public static final int GAME_B`, `public static final int GAME_C`, and `public static final int GAME_D`.

Each key code may be mapped to at most one game action. However, a game action may be associated with more than one key code. The application can translate a key code into a game action using the `public int getGameAction (int keyCode)` method, and it can translate a key code into a game action using the `public int getKeyCode (int gameAction)` method. The implementation is not allowed to change the mapping of game actions and key codes during execution of the application.

Portable applications that are interested in using game actions should translate every key event into a game action by calling the `public int getGameAction (int keyCode)` method and then testing the result. For example, on some devices the game actions UP, DOWN, LEFT and RIGHT may be mapped to 4-way navigation arrow keys. In this case, getKeyCode(UP) would return a device-dependent code for the up-arrow key. On other devices, a possible mapping would be on the number keys 2, 4, 6 and 8. In this case, getKeyCode(UP) would return KEY_NUM2. In both cases, the getGameAction() method would return the LEFT game action when the user presses the key that is a "natural left" on her device.

Commands

It is also possible for the user to issue `Command` when a canvas is current. Commands are mapped to keys and menus in a device-specific fashion. For some devices the keys used for commands may overlap with the keys that will deliver key code events to the canvas. If this is the case, the device will provide a means transparent to the application that enables the user to select a mode that determines whether these keys will deliver commands or key code events to the application. The set of key code events available to a canvas will not change depending upon the number of commands that are present on the canvas. Game developers should be aware that access to commands will vary greatly across devices, and that requiring the user to issue commands during game play may have a great impact on the ease with which the game can be played.

Event Delivery

The Canvas object defines several methods that are called by the implementation. These methods are primarily for the purpose of delivering events to the application, and so they are referred to as *event delivery* methods. The set of methods is:

- showNotify()
- hideNotify()
- keyPressed()
- keyRepeated()
- keyReleased()
- pointerPressed()
- pointerDragged()
- pointerReleased()
- paint()
- the CommandListener's commandAction() method

These methods are all called serially. That is, the implementation will never call an event delivery method before a prior call to *any* of the event delivery methods has returned. (But see the note below.) This property enables applications to be assured that processing of a previous user event will have completed before the next event is delivered.

Calls to the run() method of Runnable objects passed to Display.callSerially() will also be serialized along with calls to the event delivery methods. See `public void callSerially (javax.microedition.lcdui.Runnable r)` for further information.

Note: The serviceRepaints() method is an exception to this rule, as it blocks until paint() is called and returns. This will occur even if the application is in the midst of one of the event delivery methods and it calls serviceRepaints().

The key-related, pointer-related, paint(), and commandAction() methods will only be called while the Canvas is actually visible on the output device. These methods will therefore only be called on this Canvas object only after a call to showNotify() and before a call to hideNotify(). After hideNotify() has been called, none of the key, pointer, paint, and commandAction() methods will be called until after a subsequent call to showNotify() has returned. A call to a sun() method resulting from callSerially() may occur irrespective of calls to showNotify() and hideNotify().

The `protected void showNotify ()` method is called prior to the Canvas actually being made visible on the display, and the `protected void hideNotify ()` method is called after the Canvas has been removed from the display. The visibility state of a Canvas (or any other Displayable object) may be queried through the use of the `public boolean isShown ()` method. The change in visibility state of a Canvas may be caused by the application management software moving MIDlets between foreground and background states, or by the system obscuring the Canvas with system screens. Thus, the calls to showNotify() and hideNotify() are not under the control of the MIDlet and may occur fairly frequently. Application developers are encouraged to perform expensive setup and teardown tasks outside the showNotify() and hideNotify() methods in order to make them as lightweight as possible.

Member Summary

Fields

	`int`	`public static final int DOWN`
	`int`	`public static final int FIRE`
	`int`	`public static final int GAME_A`

Member Summary

	int	public static final int GAME_B
	int	public static final int GAME_C
	int	public static final int GAME_D
	int	public static final int KEY_NUM0
	int	public static final int KEY_NUM1
	int	public static final int KEY_NUM2
	int	public static final int KEY_NUM3
	int	public static final int KEY_NUM4
	int	public static final int KEY_NUM5
	int	public static final int KEY_NUM6
	int	public static final int KEY_NUM7
	int	public static final int KEY_NUM8
	int	public static final int KEY_NUM9
	int	public static final int KEY_POUND
	int	public static final int KEY_STAR
	int	public static final int LEFT
	int	public static final int RIGHT
	int	public static final int UP
Constructors		
		protected Canvas ()
Methods		
	int	public int getGameAction (int keyCode)
	int	public int getHeight ()
	int	public int getKeyCode (int gameAction)
	String	public java.lang.String getKeyName (int keyCode)
	int	public int getWidth ()
	boolean	public boolean hasPointerEvents ()
	boolean	public boolean hasPointerMotionEvents ()
	boolean	public boolean hasRepeatEvents ()
	void	protected void hideNotify ()
	boolean	public boolean isDoubleBuffered ()
	void	protected void keyPressed (int keyCode)
	void	protected void keyReleased (int keyCode)
	void	protected void keyRepeated (int keyCode)
	void	protected abstract void paint (Graphics g)
	void	protected void pointerDragged (int x, int y)
	void	protected void pointerPressed (int x, int y)
	void	protected void pointerReleased (int x, int y)
	void	public final void repaint ()
	void	public final void repaint (int x, int y, int width, int height)
	void	public final void serviceRepaints ()
	void	protected void showNotify ()

Inherited Member Summary

Methods inherited from class Displayable

Inherited Member Summary

`public void addCommand (Command cmd)`, `public boolean isShown ()`, `public void remove-Command (Command cmd)`, `public void setCommandListener (CommandListener l)`

Fields

DOWN

```
public static final int DOWN
```

Constant for the DOWN game action.

Constant value 6 is set to DOWN.

FIRE

```
public static final int FIRE
```

Constant for the FIRE game action.

Constant value 8 is set to FIRE.

GAME_A

```
public static final int GAME_A
```

Constant for the general purpose "A" game action.

Constant value 9 is set to GAME_A.

GAME_B

```
public static final int GAME_B
```

Constant for the general purpose "B" game action.

Constant value 10 is set to GAME_B.

GAME_C

```
public static final int GAME_C
```

Constant for the general purpose "C" game action.

Constant value 11 is set to GAME_C.

GAME_D

```
public static final int GAME_D
```

Constant for the general purpose "D" game action.

Constant value 12 is set to GAME_D.

KEY_NUM0

```
public static final int KEY_NUM0
```

keyCode for ITU-T key 0.

Constant value 48 is set to KEY_NUM0.

KEY_NUM1

```
public static final int KEY_NUM1
```

keyCode for ITU-T key 1.

Constant value 49 is set to KEY_NUM1.

KEY_NUM2

```
public static final int KEY_NUM2
```

keyCode for ITU-T key 2.

Constant value 50 is set to KEY_NUM2.

KEY_NUM3

```
public static final int KEY_NUM3
```

keyCode for ITU-T key 3.

Constant value 51 is set to KEY_NUM3.

KEY_NUM4

```
public static final int KEY_NUM4
```

keyCode for ITU-T key 4.

Constant value 52 is set to KEY_NUM4.

KEY_NUM5

```
public static final int KEY_NUM5
```

keyCode for ITU-T key 5.

Constant value 53 is set to KEY_NUM5.

KEY_NUM6

```
public static final int KEY_NUM6
```

keyCode for ITU-T key 6.

Constant value 54 is set to KEY_NUM6.

KEY_NUM7

```
public static final int KEY_NUM7
```

keyCode for ITU-T key 7.

Constant value 55 is set to KEY_NUM7.

KEY_NUM8

```
public static final int KEY_NUM8
```

keyCode for ITU-T key 8.

Constant value 56 is set to KEY_NUM8.

KEY_NUM9

```
public static final int KEY_NUM9
```

keyCode for ITU-T key 9.

Constant value 57 is set to KEY_NUM09.

KEY_POUND

```
public static final int KEY_POUND
```

keyCode for ITU-T key "pound" (#).

Constant value 35 is set to KEY_POUND.

KEY_STAR

```
public static final int KEY_STAR
```

keyCode for ITU-T key "star" (*).

Constant value 42 is set to KEY_STAR.

LEFT

```
public static final int LEFT
```

Constant for the LEFT game action.

Constant value 2 is set to LEFT.

RIGHT

```
public static final int RIGHT
```

Constant for the RIGHT game action.

Constant value 5 is set to RIGHT.

UP

```
public static final int UP
```

Constant for the UP game action.

Constant value 1 is set to UP.

Constructors

Canvas()

```
protected Canvas ()
```

Constructs a new Canvas object.

Methods

getGameAction(int)

```
public int getGameAction (int keyCode)
```

Gets the game action associated with the given key code of the device. Returns zero if no game action is associated with this key code. See above for further discussion of game actions.

The mapping between key codes and game actions will not change during the execution of the application.

Parameters:
`keyCode` - the key code

Returns: the game action corresponding to this key, or 0 if none

Throws: `IllegalArgumentException` - if keyCode is not a valid key code

getHeight()

```
public int getHeight ()
```

Gets height of the displayable area in pixels. The value is unchanged during the execution of the application and all Canvases will have the same value.

Returns: height of the displayable area

getKeyCode(int)

```
public int getKeyCode (int gameAction)
```

Gets a key code that corresponds to the specified game action on the device. The implementation is required to provide a mapping for every game action, so this method will always return a valid key code for every game action. See above for further discussion of game actions.

Note that a key code is associated with at most one game action, whereas a game action may be associated with several key codes. Then, supposing that `g` is a valid game action and `k` is a valid key code for a key associated with a game action, consider the following expressions:

```
    g == getGameAction(getKeyCode(g))     // (1)
    k == getKeyCode(getGameAction(k))     // (2)
```

Expression (1) is *always* true. However, expression (2) might be true but is *not necessarily* true.

The mapping between key codes and game actions will not change during the execution of the application.

Parameters:
`gameAction` - the game action

Returns: a key code corresponding to this game action

Throws: `IllegalArgumentException` - if gameAction is not a valid game action

getKeyName(int)

```
public java.lang.String getKeyName (int keyCode)
```

Gets an informative key string for a key. The string returned will resemble the text physically printed on the key. This string is suitable for displaying to the user. For example, on a device with function keys F1 through F4, calling this method on the keyCode for the F1 key will return the string "F1". A typical use for this string will be to compose help text such as "Press F1 to proceed."

This method will return a non-empty string for every valid key code.

There is no direct mapping from game actions to key names. To get the string name for a game action GAME_A, the application must call

```
    getKeyName(getKeyCode(GAME_A))
```

Parameters:
`keyCode` - the key code being requested

Returns: a string name for the key

Throws: `IllegalArgumentException` - if keyCode is not a valid key code

getWidth()

```
public int getWidth ()
```

Gets width of the displayable area in pixels. The value is unchanged during the execution of the application and all Canvases will have the same value.

Returns: width of the displayable area

hasPointerEvents()

`public boolean hasPointerEvents ()`

Checks if the platform supports pointer press and release events.

Returns: true if the device supports pointer events

hasPointerMotionEvents()

`public boolean hasPointerMotionEvents ()`

Checks if the platform supports pointer motion events (pointer dragged). Applications may use this method to determine if the platform is capable of supporting motion events.

Returns: true if the device supports pointer motion events

hasRepeatEvents()

`public boolean hasRepeatEvents ()`

Checks if the platform can generate repeat events when key is kept down.

Returns: true if the device supports repeat events

hideNotify()

`protected void hideNotify ()`

The implementation calls hideNotify() shortly after the Canvas has been removed from the display. Canvas subclasses may override this method in order to pause animations, revoke timers, etc. The default implementation of this method in class Canvas is empty.

isDoubleBuffered()

`public boolean isDoubleBuffered ()`

Checks if the Graphics is double buffered by the implementation.

Returns: true if double buffered, false otherwise.

keyPressed(int)

`protected void keyPressed (int keyCode)`

Called when a key is pressed.

The getGameAction() method can be called to determine what game action, if any, is mapped to the key. Class Canvas has an empty implementation of this method, and the subclass has to redefine it if it wants to listen this method.

Parameters:
`keyCode` - The key code of the key that was pressed.

keyReleased(int)

```
protected void keyReleased (int keyCode)
```

Called when a key is released. The getGameAction() method can be called to determine what game action, if any, is mapped to the key. Class Canvas has an empty implementation of this method, and the subclass has to redefine it if it wants to listen this method.

Parameters:
`keyCode` - The key code of the key that was released

keyRepeated(int)

```
protected void keyRepeated (int keyCode)
```

Called when a key is repeated (held down). The getGameAction() method can be called to determine what game action, if any, is mapped to the key. Class Canvas has an empty implementation of this method, and the subclass has to redefine it if it wants to listen this method.

Parameters:
`keyCode` - The key code of the key that was repeated

See Also: `public boolean hasRepeatEvents ()`

paint(Graphics)

```
protected abstract void paint (Graphics g)
```

Renders the Canvas. The application must implement this method in order to paint any graphics.

The Graphics object's clip region defines the area of the screen that is considered to be invalid. A correctly-written paint() routine must paint *every* pixel within this region. Applications *must not* assume that they know the underlying source of the paint() call and use this assumption to paint only a subset of the pixels within the clip region. The reason is that this particular paint() call may have resulted from multiple repaint() requests, some of which may have been generated from outside the application. An application that paints only what it thinks is necessary to be painted may display incorrectly if the screen contents had been invalidated by, for example, an incoming telephone call.

Operations on this graphics object after the paint() call returns are undefined. Thus, the application *must not* cache this Graphics object for later use or use by another thread. It must only be used within the scope of this method.

The implementation may postpone visible effects of graphics operations until the end of the paint method.

The contents of the Canvas are never saved if it is hidden and then is made visible again. Thus, shortly after showNotify() is called, paint() will always be called with a Graphics object whose clip region specifies the entire displayable area of the Canvas. Applications *must not* rely on any contents being preserved from a previous occasion when the Canvas was current. This call to paint() will not necessarily occur before any other key, pointer, or commandAction() methods are called on the Canvas. Applications whose repaint recomputation is expensive may create an offscreen Image, paint into it, and then draw this image on the Canvas when paint() is called.

The application code must never call paint(); it is called only by the implementation.

The Graphics object passed to the paint() method has the following properties:

- the destination is the actual display, or if double buffering is in effect, a back buffer for the display;

- the clip region includes at least one pixel within this Canvas;
- the current color is black;
- the font is the same as the font returned by `public static Font getDefaultFont ()`;
- the stroke style is `public static final int SOLID`;
- the origin of the coordinate system is located at the upper-left corner of the Canvas; and
- the Canvas is visible, that is, a call to isShown() will return `true`.

Parameters:

`g` - the Graphics object to be used for rendering the Canvas

pointerDragged(int, int)

`protected void pointerDragged (int x, int y)`

Called when the pointer is dragged. The `public boolean hasPointerMotionEvents ()` method may be called to determine if the device supports pointer events. Class Canvas has an empty implementation of this method, and the subclass has to redefine it if it wants to listen this method.

Parameters:

`x` - The horizontal location where the pointer was dragged (relative to the Canvas)

`y` - The vertical location where the pointer was dragged (relative to the Canvas)

pointerPressed(int, int)

`protected void pointerPressed (int x, int y)`

Called when the pointer is pressed. The `public boolean hasPointerEvents ()` method may be called to determine if the device supports pointer events. Class Canvas has an empty implementation of this method, and the subclass has to redefine it if it wants to listen this method.

Parameters:

`x` - The horizontal location where the pointer was pressed (relative to the Canvas)

`y` - The vertical location where the pointer was pressed (relative to the Canvas)

pointerReleased(int, int)

`protected void pointerReleased (int x, int y)`

Called when the pointer is released. The `public boolean hasPointerEvents ()` method may be called to determine if the device supports pointer events. Class Canvas has an empty implementation of this method, and the subclass has to redefine it if it wants to listen this method.

Parameters:

`x` - The horizontal location where the pointer was released (relative to the Canvas)

`y` - The vertical location where the pointer was released (relative to the Canvas)

repaint()

`public final void repaint ()`

Requests a repaint for the entire Canvas. The effect is identical to

```
repaint(0, 0, getWidth(), getHeight());
```

repaint(int, int, int, int)

`public final void repaint (int x, int y, int width, int height)`

Requests a repaint for the specified region of the Screen. Calling this method may result in subsequent call to paint(), where the passed Graphics object's clip region will include at least the specified region.

If the canvas is not visible, or if width and height are zero or less, or if the rectangle does not specify a visible region of the display, this call has no effect.

The call to paint() occurs independently of the call to repaint(). That is, repaint() will not block waiting for paint() to finish. The paint() method will either be called after the caller of repaint() returns to the implementation (if the caller is a callback) or on another thread entirely.

To synchronize with its paint() routine, applications can use either `public void callSerially (javax.microedition.lcdui.Runnable r)` or `public final void serviceRepaints ()` , or they can code explicit synchronization into their paint() routine.

The origin of the coordinate system is above and to the left of the pixel in the upper left corner of the displayable area of the Canvas. The X-coordinate is positive right and the Y-coordinate is positive downwards.

Parameters:

`x` - the x coordinate of the rectangle to be repainted

`y` - the y coordinate of the rectangle to be repainted

`width` - the width of the rectangle to be repainted

`height` - the height of the rectangle to be repainted

See Also: `public void callSerially (javax.microedition.lcdui.Runnable r)`, `public final void serviceRepaints ()`

serviceRepaints()

`public final void serviceRepaints ()`

Forces any pending repaint requests to be serviced immediately. This method blocks until the pending requests have been serviced. If there are no pending repaints, or if this canvas is not visible on the display, this call does nothing and returns immediately.

WARNING: This method blocks until the call to the application's paint() method returns. The application has no control over which thread calls paint(); it may vary from implementation to implementation. If the caller of serviceRepaints() holds a lock that the paint() method acquires, this may result in deadlock. Therefore, callers of serviceRepaints() *must not* hold any locks that might be acquired within the paint() method.

The `public void callSerially (javax.microedition.lcdui.Runnable r)` method provides a facility where an application can be called back after painting has completed, avoiding the danger of deadlock.

See Also: `public void callSerially (javax.microedition.lcdui.Runnable r)`

showNotify()

`protected void showNotify ()`

The implementation calls showNotify() immediately prior to this Canvas being made visible on the display. Canvas subclasses may override this method to perform tasks before being shown, such as setting up animations, starting timers, etc. The default implementation of this method in class Canvas is empty.

javax.microedition.lcdui

Choice

Syntax

```
public interface Choice
```

All Known Implementing Classes: `List`, `ChoiceGroup`

Description

Choice defines an API for a user interface components implementing selection from predefined number of choices. Such UI components are `List` and `ChoiceGroup` . The contents of the Choice are represented with strings and optional images.

Each element of a Choice is composed of a text string and an optional image. The application may provide `null` for the image if the element does not have an image part. If the application provides an image, the implementation may choose to ignore the image if it exceeds the capacity of the device to display it. If the implementation displays the image, it will be displayed adjacent to the text string and the pair will be treated as a unit.

Images within any particular Choice object should all be of the same size, because the implementation is allowed to allocate the same amount of vertical space for every element.

If an element is too long to be displayed, the implementation will provide the user with means to see the whole element. If this is done by wrapping an element to multiple lines, the second and subsequent lines show a clear indication to the user that they are part of the same element and are not a new element.

After a Choice object has been created, elements may be inserted, appended, and deleted, and each element's string part and image part may be get and set. Elements within a Choice object are referred to by their indexes, which are consecutive integers in the range from zero to size()-1, with zero referring to the first element and size()-1 to the last element.

There are three types of Choices: implicit-choice (valid only for `List`), exclusive-choice, and multiple-choice.

The exclusive-choice presents a series of elements and interacts with the user. That is, when the user selects an element, that element is shown to be selected using a distinct visual representation. Exactly one element must be selected at any given time. If at any time a situation would result where there are elements in the exclusive-choice but none is selected, the implementation will choose an element and select it. This situation can arise when an element is added to an empty Choice, when the selected element is deleted from the Choice, or when a Choice is created and populated with elements by a constructor. In these cases, the choice of which element is selected is left to the implementation. Applications for which the selected element is significant should set the selection explicitly. There is no way for the user to unselect an element within an Exclusive Choice.

The implicit choice is an exclusive choice where the focused element is implicitly selected when a command is initiated.

A multiple-choice presents a series of elements and allows the user to select any number of elements in any combination. As with exclusive-choice, the multiple-choice interacts with the user in object-operation mode. The visual appearance of a multiple-choice will likely have a visual representation distinct from the exclusive-choice that shows the selected state of each element as well as indicating to the user that multiple elements may be selected.

The selected state of an element is a property of the element. This state stays with that element if other elements are inserted or deleted, causing elements to be shifted around. For example, suppose element *n* is selected, and a new element is inserted at index zero. The selected element would now have index *n+1*. A similar rule applies to deletion. Assuming *n* is greater than zero, deleting element zero would leave element *n-1* selected. Setting the contents of an element leaves its selected state unchanged. When a new element is inserted or appended, it is always unselected (except in the special case of adding an element to an empty Exclusive Choice as mentioned above).

When a Choice is present on the display the user can interact with it indefinitely (for instance, traversing from element to element and possibly scrolling). These traversing and scrolling operations do not cause application-visible events. The system notifies the application either when some application-defined `Command` is fired, or when selection state of `ChoiceGroup` is changed. When command is fired a high-level event is delivered to the listener of the Screen. The event delivery is done with `public void commandAction (Command c, Displayable d)`. In the case of `ChoiceGroup` the `public void itemStateChanged (Item item)` is called when the user changes the selection state of the ChoiceGroup. At this time the application can query the Choice for information about the currently selected element(s).

Member Summary

Fields		
	`int`	`public static final int EXCLUSIVE`
	`int`	`public static final int IMPLICIT`
	`int`	`public static final int MULTIPLE`
Methods		
	`int`	`public int append (java.lang.String stringPart, Image imagePart)`
	`void`	`public void delete (int elementNum)`
	`Image`	`public Image getImage (int elementNum)`
	`int`	`public int getSelectedFlags (boolean[] selectedArray_return)`
	`int`	`public int getSelectedIndex ()`
	`String`	`public java.lang.String getString (int elementNum)`
	`void`	`public void insert (int elementNum, java.lang.String stringPart, Image imagePart)`
	`boolean`	`public boolean isSelected (int elementNum)`
	`void`	`public void set (int elementNum, java.lang.String stringPart, Image imagePart)`
	`void`	`public void setSelectedFlags (boolean[] selectedArray)`
	`void`	`public void setSelectedIndex (int elementNum, boolean selected)`
	`int`	`public int size ()`

Fields

EXCLUSIVE

```
public static final int EXCLUSIVE
```

EXCLUSIVE is a choice having exactly one element selected at time.

Value 1 is assigned to EXCLUSIVE.

IMPLICIT

`public static final int IMPLICIT`

IMPLICIT is a choice in which the currently focused item is selected when a `Command` is initiated. (Note: IMPLICIT is not accepted by `ChoiceGroup`)

Value 3 is assigned to IMPLICIT.

MULTIPLE

`public static final int MULTIPLE`

MULTIPLE is a choice that can have arbitrary number of elements selected at a time.

Value 2 is assigned to MULTIPLE.

Methods

append(String, Image)

`public int append (java.lang.String stringPart, Image imagePart)`

Appends an element to the Choice. The added element will be the last element of the Choice. The size of the Choice grows by one.

Parameters:
`stringPart` - the string part of the element to be added

`imagePart` - the image part of the element to be added, or null if there is no image part

Returns: the assigned index of the element

Throws: `IllegalArgumentException` - if the image is mutable

`NullPointerException` - if stringPart is null

delete(int)

`public void delete (int elementNum)`

Deletes the element referenced by elementNum. The size of the Choice shrinks by one. It is legal to delete all elements from a Choice. The elementNum parameter must be within the range [0..size()-1], inclusive.

Parameters:
`elementNum` - the index of the element to be deleted

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

getImage(int)

`public Image getImage (int elementNum)`

Gets the Image part of the element referenced by elementNum. The elementNum parameter must be within the range [0..size()-1], inclusive.

Parameters:

`elementNum` - the index of the element to be queried

Returns: the image part of the element, or null if there is no image

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

See Also: `public java.lang.String getString (int elementNum)`

getSelectedFlags(boolean[])

```
public int getSelectedFlags (boolean[] selectedArray_return)
```

Queries the state of a Choice and returns the state of all elements in the boolean array selectedArray_return. NOTE: this is a result parameter. It must be at least as long as the size of the Choice as returned by size(). If the array is longer, the extra elements are set to false.

This call is valid for all types of Choices. For MULTIPLE, any number of elements may be selected and set to true in the result array. For EXCLUSIVE and IMPLICIT exactly one element will be selected (unless there are zero elements in the Choice).

Parameters:

`selectedArray_return` - array to contain the results

Returns: the number of selected elements in the Choice

Throws: `IllegalArgumentException` - if selectedArray_return is shorter than the size of the Choice.

`NullPointerException` - if selectedArray_return is null

getSelectedIndex()

```
public int getSelectedIndex ()
```

Returns the index number of an element in the Choice that is selected. For Choice types EXCLUSIVE and IMPLICIT there is at most one element selected, so this method is useful for determining the user's choice. Returns -1 if the Choice has no elements (and therefore has no selected elements).

For MULTIPLE, this always returns -1 because no single value can in general represent the state of such a Choice. To get the complete state of a MULTIPLE Choice, see `public int getSelectedFlags (boolean[] selectedArray_return)` .

Returns: index of selected element, or -1 if none

getString(int)

```
public java.lang.String getString (int elementNum)
```

Gets the String part of the element referenced by elementNum. The elementNum parameter must be within the range [0..size()-1], inclusive.

Parameters:
`elementNum` - the index of the element to be queried

Returns: the string part of the element

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

See Also: `public Image getImage (int elementNum)`

---

insert(int, String, Image)

```
public void insert (int elementNum, java.lang.String stringPart, Image imagePart)
```

Inserts an element into the Choice just prior to the element specified. The size of the Choice grows by one. The elementNum parameter must be within the range [0..size()], inclusive. The index of the last element is size()-1, and so there is actually no element whose index is size(). If this value is used for elementNum, the new element is inserted immediately after the last element. In this case, the effect is identical to `public int append (java.lang.String stringPart, Image imagePart)` .

Parameters:
`elementNum` - the index of the element where insertion is to occur

`stringPart` - the string part of the element to be inserted

`imagePart` - the image part of the element to be inserted, or null if there is no image part

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

`IllegalArgumentException` - if the image is mutable

`NullPointerException` - if stringPart is null

---

isSelected(int)

```
public boolean isSelected (int elementNum)
```

Gets a boolean value indicating whether this element is selected. The elementNum parameter must be within the range [0..size()-1], inclusive.

Parameters:
`elementNum` - the index of the element to be queried

Returns: selection state of the element

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

---

set(int, String, Image)

```
public void set (int elementNum, java.lang.String stringPart, Image imagePart)
```

Sets the element referenced by elementNum to the specified element, replacing the previous contents of the element. The elementNum parameter must be within the range [0..size()-1], inclusive.

Parameters:
`elementNum` - the index of the element to be set

`stringPart` - the string part of the new element

`imagePart` - the image part of the element, or null if there is no image part

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

`IllegalArgumentException` - if the image is mutable

`NullPointerException` - if stringPart is null

setSelectedFlags(boolean[])

```
public void setSelectedFlags (boolean[] selectedArray)
```

Attempts to set the selected state of every element in the Choice. The array must be at least as long as the size of the Choice. If the array is longer, the additional values are ignored.

For Choice objects of type MULTIPLE, this sets the selected state of every element in the Choice. An arbitrary number of elements may be selected.

For Choice objects of type EXCLUSIVE and IMPLICIT, exactly one array element must have the value true. If no element is true, the first element in the Choice will be selected. If two or more elements are true, the implementation will choose the first true element and select it.

Parameters:
`selectedArray` - an array in which the method collect the selection status

Throws: `IllegalArgumentException` - if selectedArray is shorter than the size of the Choice

`NullPointerException` - if selectedArray is null

setSelectedIndex(int, boolean)

```
public void setSelectedIndex (int elementNum, boolean selected)
```

For MULTIPLE, this simply sets an individual element's selected state.

For EXCLUSIVE, this can be used only to select any element, that is, the `selected` parameter must be `true` . When an element is selected, the previously selected element is deselected. If `selected` is `false` , this call is ignored. If element was already selected, the call has no effect.

For IMPLICIT, this can be used only to select any element, that is, the `selected` parameter must be `true` . When an element is selected, the previously selected element is deselected. If `selected` is `false` , this call is ignored. If element was already selected, the call has no effect.

The call to setSelectedIndex does not cause implicit activation of any Command.

For all list types, the elementNum parameter must be within the range [0..size()-1], inclusive.

Parameters:
`elementNum` - the index of the element, starting from zero

`selected` - the state of the element, where `true` means selected and `false` means not selected

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

size()

```
public int size ()
```

Gets the number of elements present.

Returns: the number of elements in the Choice

javax.microedition.lcdui

ChoiceGroup

Syntax

```
public class ChoiceGroup extends Item implements Choice
```

```
Item
  |
  +--javax.microedition.lcdui.ChoiceGroup
```

All Implemented Interfaces: Choice

Description

A ChoiceGroup is a group of selectable elements intended to be placed within a Form . The group may be created with a mode that requires a single choice to be made or that allows multiple choices. The implementation is responsible for providing the graphical representation of these modes and must provide visually different graphics for different modes. For example, it might use "radio buttons" for the single choice mode and "check boxes" for the multiple choice mode.

Note: most of the essential methods have been specified in the Choice interface.

Member Summary

Constructors		
		`public ChoiceGroup (java.lang.String label, int choiceType)`
		`public ChoiceGroup (java.lang.String label, int choiceType, java.lang.String[] stringElements, Image[] imageElements)`
Methods		
	`int`	`public int append (java.lang.String stringPart, Image imagePart)`
	`void`	`public void delete (int elementNum)`
	`Image`	`public Image getImage (int elementNum)`
	`int`	`public int getSelectedFlags (boolean[] selectedArray_return)`
	`int`	`public int getSelectedIndex ()`
	`String`	`public java.lang.String getString (int elementNum)`
	`void`	`public void insert (int elementNum, java.lang.String stringElement, Image imageElement)`
	`boolean`	`public boolean isSelected (int elementNum)`
	`void`	`public void set (int elementNum, java.lang.String stringPart, Image imagePart)`
	`void`	`public void setSelectedFlags (boolean[] selectedArray)`
	`void`	`public void setSelectedIndex (int elementNum, boolean selected)`
	`int`	`public int size ()`

Inherited Member Summary

Fields inherited from interface Choice

Inherited Member Summary

`public static final int EXCLUSIVE`, `public static final int IMPLICIT`, `public static final int MULTIPLE`

Methods inherited from class `Item`

`public java.lang.String getLabel ()`, `public void setLabel (java.lang.String label)`

Constructors

ChoiceGroup(String, int)

```
public ChoiceGroup (java.lang.String label, int choiceType)
```

Creates a new, empty ChoiceGroup, specifying its title and its type. The type must be one of EXCLUSIVE or MULTIPLE. The IMPLICIT choice type is not allowed within a ChoiceGroup.

Parameters:

`label` - the item's label (see `Item`)

`choiceType` - either EXCLUSIVE or MULTIPLE

Throws: `IllegalArgumentException` - if choice type is neither EXCLUSIVE nor MULTIPLE

See Also: `public static final int EXCLUSIVE`, `public static final int MULTIPLE`, `public static final int IMPLICIT`

ChoiceGroup(String, int, String[], Image[])

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

Creates a new ChoiceGroup, specifying its title, the type of the ChoiceGroup, and an array of Strings and Images to be used as its initial contents.

The type must be one of EXCLUSIVE or MULTIPLE. The IMPLICIT type is not allowed for Choice-Group.

The stringElements array must be non-null and every array element must also be non-null. The length of the stringElements array determines the number of elements in the ChoiceGroup. The imageElements array may be null to indicate that the ChoiceGroup elements have no images. If the imageElements array is non-null, it must be the same length as the stringElements array. Individual elements of the imageElements array may be null in order to indicate the absence of an image for the corresponding ChoiceGroup element. Any elements present in the imageElements array must refer to immutable images.

Parameters:

`label` - the item's label (see `Item`)

`choiceType` - EXCLUSIVE or MULTIPLE

`stringElements` - set of strings specifying the string parts of the ChoiceGroup elements

`imageElements` - set of images specifying the image parts of the ChoiceGroup elements

Throws: `NullPointerException` - if stringElements is null

`NullPointerException` - if the stringElements array contains any null elements

`IllegalArgumentException` - if the imageElements array is non-null and has a different length from the stringElements array

`IllegalArgumentException` - if choiceType is neither EXCLUSIVE nor MULTIPLE

`IllegalArgumentException` - if any image in the imageElements array is mutable

See Also: `public static final int EXCLUSIVE`, `public static final int MULTIPLE`, `public static final int IMPLICIT`

Methods

append(String, Image)

`public int append (java.lang.String stringPart, Image imagePart)`

Specified By: `public int append (java.lang.String stringPart, Image imagePart)` in interface `Choice`

Parameters:
`stringPart` - the string part of the element to be added

`imagePart` - the image part of the element to be added, or null if there is no image part

Returns: the assigned index of the element

Throws: `IllegalArgumentException` - if the image is mutable

`NullPointerException` - if stringPart is null

delete(int)

`public void delete (int elementNum)`

Specified By: `public void delete (int elementNum)` in interface `Choice`

Parameters:
`elementNum` - the index of the element to be deleted

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

getImage(int)

`public Image getImage (int elementNum)`

Specified By: `public Image getImage (int elementNum)` in interface `Choice`

Parameters:
`elementNum` - the number of the element to be queried

Returns: the image part of the element, or null if there is no image

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

See Also: `public java.lang.String getString (int elementNum)`

getSelectedFlags(boolean[])

```
public int getSelectedFlags (boolean[] selectedArray_return)
```

Queries the state of a ChoiceGroup and returns the state of all elements in the boolean array selectedArray_return. NOTE: this is a result parameter. It must be at least as long as the size of the ChoiceGroup as returned by size(). If the array is longer, the extra elements are set to false.

For ChoiceGroup objects of type MULTIPLE, any number of elements may be selected and set to true in the result array. For ChoiceGroup objects of type EXCLUSIVE, exactly one element will be selected, unless there are zero elements in the ChoiceGroup.

Specified By: `public int getSelectedFlags (boolean[] selectedArray_return)` in interface `Choice`

Parameters:
`selectedArray_return` - array to contain the results.

Returns: the number of selected elements in the ChoiceGroup

Throws: `IllegalArgumentException` - if selectedArray_return is shorter than the size of the ChoiceGroup.

`NullPointerException` - if selectedArray_return is null.

getSelectedIndex()

```
public int getSelectedIndex ()
```

Returns the index number of an element in the ChoiceGroup that is selected. For ChoiceGroup objects of type EXCLUSIVE there is at most one element selected, so this method is useful for determining the user's choice. Returns -1 if there are no elements in the ChoiceGroup.

For ChoiceGroup objects of type MULTIPLE, this always returns -1 because no single value can in general represent the state of such a ChoiceGroup. To get the complete state of a MULTIPLE Choice, see `public int getSelectedFlags (boolean[] selectedArray_return)` .

Specified By: `public int getSelectedIndex ()` in interface `Choice`

Returns: index of selected element, or -1 if none

getString(int)

```
public java.lang.String getString (int elementNum)
```

Specified By: `public java.lang.String getString (int elementNum)` in interface `Choice`

Parameters:
`elementNum` - the index of the element to be queried

Returns: the string part of the element

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

See Also: `public Image getImage (int elementNum)`

insert(int, String, Image)

`public void insert (int elementNum, java.lang.String stringElement, Image imageElement)`

Specified By: `public void insert (int elementNum, java.lang.String stringPart, Image imagePart)` in interface `Choice`

Parameters:

`elementNum` - the index of the element where insertion is to occur

`stringPart` - the string part of the element to be inserted

`imagePart` - the image part of the element to be inserted, or null if there is no image part

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

`IllegalArgumentException` - if the image is mutable

`NullPointerException` - if stringPart is null

isSelected(int)

`public boolean isSelected (int elementNum)`

Specified By: `public boolean isSelected (int elementNum)` in interface `Choice`

Parameters:

`elementNum` - the index of the element to be queried

Returns: selection state of the element

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

set(int, String, Image)

`public void set (int elementNum, java.lang.String stringPart, Image imagePart)`

Specified By: `public void set (int elementNum, java.lang.String stringPart, Image imagePart)` in interface `Choice`

Parameters:

`elementNum` - the index of the element to be set

`stringPart` - the string part of the new element

`imagePart` - the image part of the element, or null if there is no image part

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

`IllegalArgumentException` - if the image is mutable

`NullPointerException` - if stringPart is null

setSelectedFlags(boolean[])

```
public void setSelectedFlags (boolean[] selectedArray)
```

Attempts to set the selected state of every element in the ChoiceGroup. The array must be at least as long as the size of the ChoiceGroup. If the array is longer, the additional values are ignored.

For ChoiceGroup objects of type MULTIPLE, this sets the selected state of every element in the Choice. An arbitrary number of elements may be selected.

For ChoiceGroup objects of type EXCLUSIVE, exactly one array element must have the value `true`. If no element is true, the first element in the Choice will be selected. If two or more elements are true, the implementation will choose the first true element and select it.

Specified By: `public void setSelectedFlags (boolean[] selectedArray)` in interface `Choice`

Parameters:
`selectedArray` - an array in which the method collect the selection status

Throws: `IllegalArgumentException` - if selectedArray is shorter than the size of the ChoiceGroup.

`NullPointerException` - if the selectedArray is null.

setSelectedIndex(int, boolean)

```
public void setSelectedIndex (int elementNum, boolean selected)
```

For ChoiceGroup objects of type MULTIPLE, this simply sets an individual element's selected state.

For ChoiceGroup objects of type EXCLUSIVE, this can be used only to select an element. That is, the `selected` parameter must be `true` . When an element is selected, the previously selected element is deselected. If `selected` is `false` , this call is ignored.

For both list types, the elementNum parameter must be within the range [0..size()-1], inclusive.

Specified By: `public void setSelectedIndex (int elementNum, boolean selected)` in interface `Choice`

Parameters:
`elementNum` - the number of the element. Indexing of the elements is zero-based.

`selected` - the new state of the element true=selected, false=not selected.

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

size()

```
public int size ()
```

Specified By: `public int size ()` in interface `Choice`

Returns: the number of elements in the ChoiceGroup

javax.microedition.lcdui
Command

Syntax

`public class Command`

`javax.microedition.lcdui.Command`

Description

The Command class is a construct that encapsulates the semantic information of an action. The behavior that the command activates is not encapsulated in this object. This means that command contains only information about "command" not the actual action that happens when command is activated. The action is defined in a `CommandListener` associated with the Screen. Command objects are *presented* in the user interface and the way they are presented may depend on the semantic information contained within the command.

Commands may be implemented in any user interface construct that has semantics for activating a single action. This, for example, can be a soft button, item in a menu, or some other direct user interface construct. For example, a speech interface may present these commands as voice tags.

The mapping to concrete user interface constructs may also depend on the total number of the commands. For example, if an application asks for more abstract commands then can be mapped onto the available physical buttons on a device, then the device may use an alternate human interface such as a menu. For example, the abstract commands that cannot be mapped onto physical buttons are placed in a menu and the label "Menu" is mapped onto one of the programmable buttons.

A command contains three pieces of information: a *label*, a *type*, and a *priority*. The label is used for the visual representation of the command, whereas the type and the priority indicate the semantics of the command.

Label

Each command includes a label string. The label string is what the application requests to be shown to the user to represent this command. For example, this string may appear next to a soft button on the device or as an element in a menu. For command types other than SCREEN, this label may be overridden by a system-specific label that is more appropriate for this command on this device. The contents of the label string are otherwise not interpreted by the implementation.

Type

The application uses the command type to specify the intent of this command. For example, if the application specifies that the command is of type BACK, and if the device has a standard of placing the "back" operation on a certain soft-button, the implementation can follow the style of the device by using the semantic information as a guide. The defined types are `public static final int BACK`, `public static final int CANCEL`, `public static final int EXIT`, `public static final int HELP`, `public static final int ITEM`, `public static final int OK`, `public static final int SCREEN`, and `public static final int STOP`.

Priority

The application uses the priority value to describe the importance of this command relative to other commands on the same screen. Priority values are integers, where a lower number indicates greater importance. The actual values are chosen by the application. A priority value of one might indicate the most important command, priority values of two, three, four, and so on indicate commands of lesser importance.

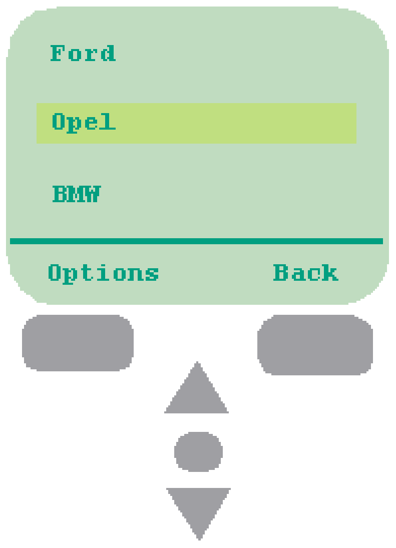
Typically, the implementation first chooses the placement of a command based on the type of command and then places similar commands based on a priority order. This could mean that the command with the highest priority is placed so that user can trigger it directly and that commands with lower priority are placed on a menu.

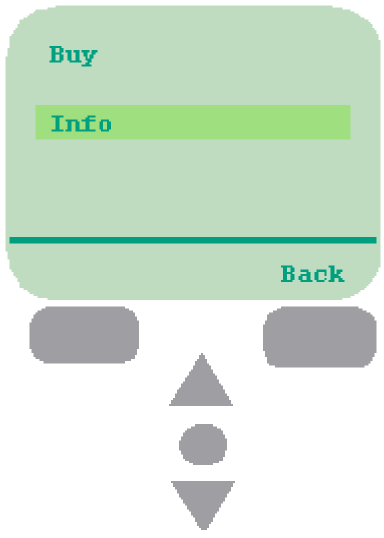
It is not an error for there to be commands on the same screen with the same priorities and types. If this occurs, the implementation will choose the order in which they are presented.

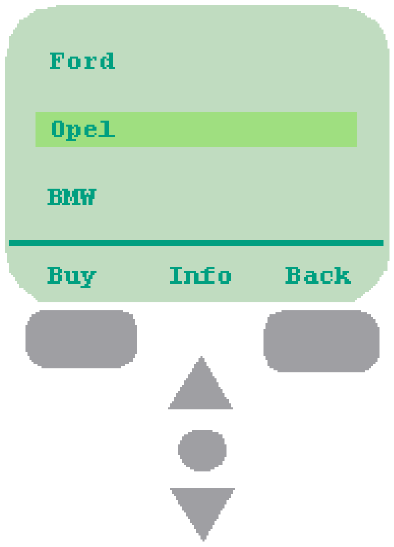
For example, if the application has the following set of commands:

```
        new Command("Buy", Command.SCREEN, 1);
        new Command("Info", Command.SCREEN, 1);
        new Command("Back", Command.BACK, 1);
```

An implementation with two soft buttons may map the BACK command to the right soft button and create an "Options" menu on the left soft button to contain the other commands.







The application is always responsible for providing the means for the user to progress through different screens. An application may set up a screen that has no commands. This is allowed by the API but is generally not useful; if this occurs the user would have no means to move to another screen. Such program would simply considered to be in error. A typical device should provide a means for the user to direct the application manager to kill the erroneous application.

Member Summary

Fields		
	`int`	`public static final int BACK`
	`int`	`public static final int CANCEL`
	`int`	`public static final int EXIT`
	`int`	`public static final int HELP`
	`int`	`public static final int ITEM`
	`int`	`public static final int OK`
	`int`	`public static final int SCREEN`
	`int`	`public static final int STOP`
Constructors		
		`public Command (java.lang.String label, int commandType, int priority)`
Methods		
	`int`	`public int getCommandType ()`
	`String`	`public java.lang.String getLabel ()`
	`int`	`public int getPriority ()`

Fields

BACK

`public static final int BACK`

A navigation command that returns the user to the logically previous screen. The jump to the previous screen is not done automatically by the implementation but by the `public void commandAction (Command c, Displayable d)` provided by the application. Note that the application defines the actual action since the strictly previous screen may not be logically correct.

Value 2 is assigned to BACK.

See Also: `public static final int CANCEL`, `public static final int STOP`

CANCEL

`public static final int CANCEL`

A command that is a standard negative answer to a dialog implemented by current screen. Nothing is cancelled automatically by the implementation; cancellation is implemented by the `public void commandAction (Command c, Displayable d)` provided by the application.

With this command type, the application hints to the implementation that the user wants to dismiss the current screen without taking any action on anything that has been entered into the current screen, and usually that the user wants to return to the prior screen. In many cases CANCEL is interchangeable with BACK, but BACK is mainly used for navigation as in a browser-oriented applications.

Value 3 is assigned to CANCEL.

See Also: `public static final int BACK`, `public static final int STOP`

EXIT

`public static final int EXIT`

A command used for exiting from the application. When the user invokes this command, the implementation does not exit automatically. The application's `public void commandAction (Command c, Displayable d)` will be called, and it should exit the application if it is appropriate to do so.

Value 7 is assigned to EXIT.

HELP

`public static final int HELP`

This command specifies a request for on-line help. No help information is shown automatically by the implementation. The `public void commandAction (Command c, Displayable d)` provided by the application is responsible for showing the help information.

Value 5 is assigned to HELP.

ITEM

`public static final int ITEM`

With this command type the application can hint to the implementation that the command is specific to a particular item on the screen. For example, an implementation of List can use this information for creating context sensitive menus.

Value 8 is assigned to ITEM.

OK

`public static final int OK`

A command that is a standard positive answer to a dialog implemented by current screen. Nothing is done automatically by the implementation; any action taken is implemented by the `public void commandAction (Command c, Displayable d)` provided by the application.

With this command type the application hints to the implementation that the user will use this command to ask the application to confirm the data that has been entered in the current screen and to proceed to the next logical screen.

CANCEL is often used together with OK.

Value 4 is assigned to OK.

See Also: `public static final int CANCEL`

SCREEN

`public static final int SCREEN`

Specifies an application-defined command that pertains to the current screen. Examples could be "Load" and "Save".

Value 1 is assigned to SCREEN.

STOP

`public static final int STOP`

A command that will stop some currently running process, operation, etc. Nothing is stopped automatically by the implementation. The cessation must be performed by the `public void commandAction (Command c, Displayable d)` provided by the application.

With this command type the application hints to the implementation that the user will use this command to stop any currently running process visible to the user on the current screen. Examples of running processes might include downloading or sending of data. Use of the STOP command does not necessarily imply a switch to another screen.

Value 6 is assigned to STOP.

See Also: `public static final int BACK`, `public static final int CANCEL`

Constructors

Command(String, int, int)

`public Command (java.lang.String label, int commandType, int priority)`

Creates a new command object with the given label, type, and priority.

Parameters:

`label` - the label string

`commandType` - the command's type, one of `public static final int BACK` , `public static final int CANCEL` , `public static final int EXIT` , `public static final int HELP` , `public static final int ITEM` , `public static final int OK` , `public static final int SCREEN` , or `public static final int STOP`

`priority` - the command's priority value

Throws: `IllegalArgumentException` - if the commandType is an invalid type

`NullPointerException` - if label is null

Methods

getCommandType()

`public int getCommandType ()`

Gets the type of the command.

Returns: type of the Command

getLabel()

`public java.lang.String getLabel ()`

Gets the label of the command.

Returns: label of the Command

getPriority()

`public int getPriority ()`

Gets the priority of the command.

Returns: priority of the Command

javax.microedition.lcdui

CommandListener

Syntax

```
public interface CommandListener
```

Description

This interface is used by applications which need to receive high-level events from the implementation. An application will provide an implementation of a Listener (typically by using a nested class or an inner class) and will then provide an instance of it on a Screen in order to receive high-level events on that screen.

The specification does not require the platform to create several threads for the event delivery. Thus, if a Listener method does not return or the return is not delayed, the system may be blocked. So, there is the following note to application developers:

- *the Listener method should return immediately.*

See Also: `public void setCommandListener (CommandListener l)`

Member Summary	
Methods	
void	`public void commandAction (Command c, Displayable d)`

Methods

commandAction(Command, Displayable)

```
public void commandAction (Command c, Displayable d)
```

Indicates that a command event has occurred on Displayable d.

Note for application developer: the method should return immediately.

Parameters:

`c` - a Command object identifying the command. This is either one of the applications have been added to Displayable with `public void addCommand (Command cmd)` or is the implicit `public static final Command SELECT_COMMAND` of List.

`d` - the Displayable on which this event has occurred

javax.microedition.lcdui

DateField

Syntax

```
public class DateField extends Item

Item
  |
  +--javax.microedition.lcdui.DateField
```

Description

A DateField is an editable component for presenting date and time (calendar) information that may be placed into a Form. Value for this field can be initially set or left unset. If value is not set then the UI for the field shows this clearly. The field value for "not initialized state" is not valid value and `getDate()` for this state returns null.

Instance of a DateField can be configured to accept date or time information or both of them. This input mode configuration is done by DATE, TIME or DATE_TIME static fields of this class. DATE input mode allows to set only date information and TIME only time information (hours, minutes). DATE_TIME allows to set both clock time and date values.

In TIME input mode the date components of Date object must be set to the "zero epoch" value of January 1, 1970.

Calendar calculations in this field are based on default locale and defined time zone. Because of the calculations and different input modes date object may not contain same millisecond value when set to this field and get back from this field.

Member Summary

Fields		
	`int`	`public static final int DATE`
	`int`	`public static final int DATE_TIME`
	`int`	`public static final int TIME`
Constructors		
		`public DateField (java.lang.String label, int mode)`
		`public DateField (java.lang.String label, int mode, java.util.TimeZone timeZone)`
Methods		
	`Date`	`public java.util.Date getDate ()`
	`int`	`public int getInputMode ()`
	`void`	`public void setDate (java.util.Date date)`
	`void`	`public void setInputMode (int mode)`

Inherited Member Summary

Methods inherited from class `Item`

`public java.lang.String getLabel ()`, `public void setLabel (java.lang.String label)`

Fields

DATE

```
public static final int DATE
```

Input mode for date information (day, month, year). With this mode this DateField presents and allows only to modify date value. The time information of date object is ignored.

Value 1 is assigned to DATE.

DATE_TIME

```
public static final int DATE_TIME
```

Input mode for date (day, month, year) and time (minutes, hours) information. With this mode this Date-Field presents and allows to modify both time and date information.

Value 3 is assigned to DATE_TIME.

TIME

```
public static final int TIME
```

Input mode for time information (hours and minutes). With this mode this DateField presents and allows only to modify time. The date components should be set to the "zero epoch" value of January 1, 1970 and should not be accessed.

Value 2 is assigned to TIME.

Constructors

DateField(String, int)

```
public DateField (java.lang.String label, int mode)
```

Creates a DateField object with the specified label and mode. This call is identical to DateField(label, mode, null).

Parameters:
`label` - item label

`mode` - the input mode, one of DATE, TIME or DATE_TIME

Throws: `IllegalArgumentException` - if the input mode's value is invalid

DateField(String, int, TimeZone)

```
public DateField (java.lang.String label, int mode, java.util.TimeZone timeZone)
```

Creates a date field in which calendar calculations are based on specific TimeZone object and the default calendaring system for the current locale. The value of the DateField is initially in the "uninitialized" state. If timeZone is null, the system's default time zone is used.

Parameters:

`label` - item label

`mode` - the input mode, one of DATE, TIME or DATE_TIME

`timeZone` - a specific time zone, or null for the default time zone

Throws: `IllegalArgumentException` - if the input mode's value is invalid

Methods

getDate()

```
public java.util.Date getDate ()
```

Returns date value of this field. Returned value is null if field value is not initialized. The date object is constructed according the rules of locale specific calendaring system and defined time zone. In TIME mode field the date components are set to the "zero epoch" value of January 1, 1970. If a date object that presents time beyond one day from this "zero epoch" then this field is in "not initialized" state and this method returns null. In DATE mode field the time component of the calendar is set to zero when constructing the date object.

Returns: date object representing time or date depending on input mode

getInputMode()

```
public int getInputMode ()
```

Gets input mode for this date field. Valid input modes are DATE, TIME and DATE_TIME.

Returns: input mode of this field

setDate(Date)

```
public void setDate (java.util.Date date)
```

Sets a new value for this field. Null can be passed to set the field state to "not initialized" state. The input mode of this field defines what components of passed Date object is used.

In TIME input mode the date components must be set to the "zero epoch" value of January 1, 1970. If a date object that presents time beyond one day then this field is in "not initialized" state. In TIME input mode the date component of Date object is ignored and time component is used to precision of minutes.

In DATE input mode the time component of Date object is ignored.

In DATE_TIME input mode the date and time component of Date are used but only to precision of minutes.

Parameters:
`date` - new value for this field

setInputMode(int)

```
public void setInputMode (int mode)
```

Set input mode for this date field. Valid input modes are DATE, TIME and DATE_TIME.

Parameters:
`mode` - the input mode, must be one of DATE, TIME or DATE_TIME

Throws: `IllegalArgumentException` - if an invalid value is specified

javax.microedition.lcdui

Display

Syntax

`public class Display`

`javax.microedition.lcdui.Display`

Description

Display represents the manager of the display and input devices of the system. It includes methods for retrieving properties of the device and for requesting that objects be displayed on the device. Other methods that deal with device attributes are primarily used with `Canvas` objects and are thus defined there instead of here.

There is exactly one instance of Display per `MIDlet` and the application can get a reference to that instance by calling the `public static Display getDisplay (javax.microedition.midlet m)` method. The application may call the getDisplay() method from the beginning of the startApp() call until the destroyApp() call returns. The Display object returned by all calls to getDisplay() will remain the same during this time.

A typical application will perform the following actions in response to calls to its MIDlet methods:

- **startApp -** the application is moving from the paused state to the active state. Initialization of objects needed while the application is active should be done. The application may call `public void setCurrent (Displayable nextDisplayable)` for the first screen if that has not already been done. Note that startApp() can be called several times if pauseApp() has been called in between. This means that one-time initialization should not take place here but instead should occur within the MIDlet's constructor.
- **pauseApp -** the application may pause its threads. Also, if it is desirable to start with another screen when the application is re-activated, the new screen should be set with setCurrent().
- **destroyApp -** the application should free resources, terminate threads, etc. The behavior of method calls on user interface objects after destroyApp() has returned is undefined.

The user interface objects that are shown on the display device are contained within a `Displayable` object. At any time the application may have at most one Displayable object that it intends to be shown on the display device and through which user interaction occurs. This Displayable is referred to as the *current* Displayable.

The Display class has a `public void setCurrent (Displayable nextDisplayable)` method for setting the current Displayable and a `public Displayable getCurrent ()` method for retrieving the current Displayable. The application has control over its current Displayable and may call setCurrent() at any time. Typically, the application will change the current Displayable in response to some user action. This is not always the case, however. Another thread may change the current Displayable in response to some other stimulus. The current Displayable will also be changed when the timer for an `Alert` elapses.

The application's current Displayable may not physically be drawn on the screen, nor will user events (such as keystrokes) that occur necessarily be directed to the current Displayable. This may occur because of the presence of other MIDlet applications running simultaneously on the same device.

An application is said to be in the *foreground* if its current Displayable is actually visible on the display device and if user input device events will be delivered to it. If the application is not in the foreground, it lacks access to both the display and input devices, and it is said to be in the *background*. The policy for allocation of these devices to different MIDlet applications is outside the scope of this specification and is under the control of an external agent referred to as the *application management software*.

As mentioned above, the application still has a notion of its current Displayable even if it is in the background. The current Displayable is significant, even for background applications, because the current Displayable is always the one that will be shown the next time the application is brought into the foreground. The application can determine whether a Displayable is actually visible on the display by calling `public boolean isShown ()`. In the case of Canvas, the `protected void showNotify ()` and `protected void hideNotify ()` methods are called when the Canvas is made visible and is hidden, respectively.

Each MIDlet application has its own current Displayable. This means that the `public Displayable getCurrent ()` method returns the MIDlet's current Displayable, regardless of the MIDlet's foreground/background state. For example, suppose a MIDlet running in the foreground has current Displayable *F*, and a MIDlet running in the background has current Displayable *B*. When the foreground MIDlet calls getCurrent(), it will return *F*, and when the background MIDlet calls getCurrent(), it will return *B*. Furthermore, if either MIDlet changes its current Displayable by calling setCurrent(), this will not affect the any other MIDlet's current Displayable.

It is possible for getCurrent() to return null. This may occur at startup time, before the MIDlet application has called setCurrent() on its first screen. The getCurrent() method will never return a reference to a Displayable object that was not passed in a prior call to setCurrent() call by this MIDlet.

System Screens

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

If system screen obscures a canvas, its hideNotify() method 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 called after showNotify() is called.

Member Summary

Methods

`void`	`public void callSerially (javax.microedition.lcdui.Runnable r)`
`Displayable`	`public Displayable getCurrent ()`
`Display`	`public static Display getDisplay (javax.microedition.midlet m)`
`boolean`	`public boolean isColor ()`
`int`	`public int numColors ()`
`void`	`public void setCurrent (Alert alert, Displayable nextDisplayable)`
`void`	`public void setCurrent (Displayable nextDisplayable)`

Methods

callSerially(Runnable)

```
public void callSerially (javax.microedition.lcdui.Runnable r)
```

Causes the Runnable object `r` to have its run() method called later, serialized with the event stream, soon after completion of the repaint cycle. As noted in section on event delivery in the Canvas class, the methods that deliver event notifications to the current canvas are all called serially. The call to r.run() will be serialized along with the event calls on the current canvas. The run() method will be called exactly once for each call to callSerially(). Calls to run() will occur in the order in which they were requested by calls to callSerially().

If there is a repaint pending at the time of a call to callSerially(), the current Canvas's paint() method will be called and will return, and a buffer switch will occur (if double buffering is in effect), before the Runnable's run() method is called. Calls to the run() method will occur in a timely fashion, but they are not guaranteed to occur immediately after the repaint cycle finishes, or even before the next event is delivered.

The callSerially() method may be called from any thread. The call to the run() method will occur independently of the call to callSerially(). In particular, callSerially() will *never* block waiting for r.run() to return.

As with other callbacks, the call to r.run() must return quickly. If it is necessary to perform a long-running operation, it may be initiated from within the run() method. The operation itself should be performed within another thread, allowing run() to return.

The callSerially() facility may be used by applications to run an animation that is properly synchronized with the repaint cycle. A typical application will set up a frame to be displayed and then call repaint(). The application must then wait until the frame is actually displayed, after which the setup for the next frame may occur. The call to run() notifies the application that the previous frame has finished painting. The example below shows callSerially() being used for this purpose.

```
class Animation extends Canvas implements Runnable {
    void paint(Graphics g) { ... } // paint the current frame
    void startAnimation() {
        // set up initial frame
        repaint();
        callSerially(this);
    }
    void run() { // called after previous repaint is finished
        if ( &#47;* there are more frames *&#47; ) {
            // set up the next frame
            repaint();
            callSerially(this);
        }
    }
 }
```

Parameters:

`r` - instance of interface Runnable to be called

getCurrent()

`public Displayable getCurrent ()`

Gets the current Displayable object for this MIDlet. The Displayable object returned may not actually be visible on the display if the MIDlet is running in the background, or if the Displayable is obscured by a system screen. The `public boolean isShown ()` method may be called to determine whether the Displayable is actually visible on the display.

The value returned by getCurrent() may be `null`. This occurs after the application has been initialized but before the first call to setCurrent().

Returns: the MIDlet's current Displayable object

getDisplay(MIDlet)

`public static Display getDisplay (javax.microedition.midlet m)`

Gets the Display object that is unique to this MIDlet.

Parameters:
`m` - Midlet of the application

Returns: the display object that application can use for its user interface

Throws: `NullPointerException` - if m is null

isColor()

`public boolean isColor ()`

Gets information about color support of the device.

Returns: true if the display supports color, false otherwise

numColors()

`public int numColors ()`

Gets the number of colors (if isColor() is true) or graylevels (if isColor() is false) that can be represented on the device.

Note that number of Colors for black and white display is 2.

Returns: number of colors

setCurrent(Alert, Displayable)

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

Requests that this Alert be made current, and that nextDisplayable be made current after the Alert is dismissed. This call returns immediately regardless of the Alert's timeout value or whether it is a modal alert. The nextDisplayable must not be an Alert, and it must not be null.

In other respects, this method behaves identically to `public void setCurrent (Displayable nextDisplayable)`.

Parameters:

`alert` - the alert to be shown

`nextDisplayable` - the Displayable to be shown after this alert is dismissed

Throws: `NullPointerException` - if alert or nextDisplayable is null

`IllegalArgumentException` - if nextDisplayable is an Alert

See Also: `Alert`

---

setCurrent(Displayable)

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

Requests that a different Displayable object be made visible on the display. The change will typically not take effect immediately. It may be delayed so that it occurs between event delivery method calls, although it is not guaranteed to occur before the next event delivery method is called. The setCurrent() method returns immediately, without waiting for the change to take place. Because of this delay, a call to getCurrent() shortly after a call to setCurrent() is unlikely to return the value passed to setCurrent().

Calls to setCurrent() are not queued. A delayed request made by a setCurrent() call may be superseded by a subsequent call to setCurrent(). For example, if screen S1 is current, then

```
d.setCurrent(S2);
d.setCurrent(S3);
```

may eventually result in S3 being made current, bypassing S2 entirely.

When a MIDlet application is first started, there is no current Displayable object. It is the responsibility of the application to ensure that a Displayable is visible and can interact with the user at all times. Therefore, the application should always call setCurrent() as part of its initialization.

The application may pass `null` as the argument to setCurrent(). This does not have the effect of setting the current Displayable to `null`; instead, the current Displayable remains unchanged. However, the application management software may interpret this call as a hint from the application that it is requesting to be placed into the background. Similarly, if the application is in the background, passing a non-`null` reference to setCurrent() may be interpreted by the application management software as a hint that the application is requesting to be brought to the foreground. The request should be considered to be made even if the current Displayable is passed to the setCurrent(). For example, the code

```
d.setCurrent(d.getCurrent());
```

generally will have no effect other than requesting that the application be brought to the foreground. These requests are only hints, and there is no requirement that the application management software comply with these requests in a timely fashion if at all.

If the Displayable passed to setCurrent() is an `Alert` , the previous Displayable is restored after the Alert has been dismissed. The effect is as if setCurrent(Alert, getCurrent()) had been called. Note that this will result in an exception being thrown if the current Displayable is already an alert. To specify the Displayable to be shown after an Alert is dismissed, the application should use the `public void setCurrent (Alert alert, Displayable nextDisplayable)` method. If the application calls setCurrent() while an Alert is current, the Alert is removed from the display and any timer it may have set is cancelled.

If the application calls setCurrent() while a system screen is active, the effect may be delayed until after the system screen is dismissed. The implementation may choose to interpret setCurrent() in such a situation as a request to cancel the effect of the system screen, regardless of whether setCurrent() has been delayed.

Parameters:

`nextDisplayable` - the Displayable requested to be made current; null is allowed

javax.microedition.lcdui

Displayable

Syntax

`public abstract class Displayable`

`javax.microedition.lcdui.Displayable`

Direct Known Subclasses: `Canvas`, `Screen`

Description

An object that has the capability of being placed on the display. A Displayable object may have commands and listeners associated with it. The contents displayed and their interaction with the user are defined by subclasses.

Member Summary	
Methods	
`void`	`public void addCommand (Command cmd)`
`boolean`	`public boolean isShown ()`
`void`	`public void removeCommand (Command cmd)`
`void`	`public void setCommandListener (CommandListener l)`

Methods

addCommand(Command)

`public void addCommand (Command cmd)`

Adds a command to the Displayable. The implementation may choose, for example, to add the command to any of the available softbuttons or place it in a menu. If the added command is already in the screen (tested by comparing the object references), the method has no effect. If the Displayable is actually visible on the display, and this call affects the set of visible commands, the implementation should update the display as soon as it is feasible to do so.

Parameters:
`cmd` - the command to be added

Throws: `NullPointerException` - if cmd is null

isShown()

`public boolean isShown ()`

Checks if the Displayable is actually visible on the display. In order for a Displayable to be visible, all of the following must be true: the Display's MIDlet must be running in the foreground, the Displayable must be the Display's current screen, and the Displayable must not be obscured by a system screen.

Returns: true if the Displayable is currently visible

removeCommand(Command)

`public void removeCommand (Command cmd)`

Removes a command from the Displayable. If the command is not in the Displayable (tested by comparing the object references), the method has no effect. If the Displayable is actually visible on the display, and this call affects the set of visible commands, the implementation should update the display as soon as it is feasible to do so.

Parameters:
`cmd` - the command to be removed

setCommandListener(CommandListener)

`public void setCommandListener (CommandListener l)`

Sets a listener for `Command` to this Displayable, replacing any previous CommandListener. A `null` reference is allowed and has the effect of removing any existing listener.

Parameters:
`l` - the new listener, or `null`.

javax.microedition.lcdui
Font

Syntax

```
public final class Font
```

`javax.microedition.lcdui.Font`

Description

The Font class represents fonts and font metrics. Fonts cannot be created by applications. Instead, applications query for fonts based on font attributes and the system will attempt to provide a font that matches the requested attributes as closely as possible.

A Font's attributes are style, size, and face. Values for attributes must be specified in terms of symbolic constants. Values for the style attribute may be combined using the logical OR operator, whereas values for the other attributes may not be combined. For example, the value

```
STYLE_BOLD | STYLE_ITALIC
```

may be used to specify a bold-italic font; however

```
SIZE_LARGE | SIZE_SMALL
```

is illegal.

The values of these constants are arranged so that zero is valid for each attribute and can be used to specify a reasonable default font for the system. For clarity of programming, the following symbolic constants are provided and are defined to have values of zero:

- `STYLE_PLAIN`
- `SIZE_MEDIUM`
- `FACE_SYSTEM`

Values for other attributes are arranged to have disjoint bit patterns in order to raise errors if they are inadvertently misused (for example, using FACE_PROPORTIONAL where a style is required). However, the values for the different attributes are not intended to be combined with each other.

Member Summary		
Fields		
	`int`	`public static final int FACE_MONOSPACE`
	`int`	`public static final int FACE_PROPORTIONAL`
	`int`	`public static final int FACE_SYSTEM`
	`int`	`public static final int SIZE_LARGE`
	`int`	`public static final int SIZE_MEDIUM`
	`int`	`public static final int SIZE_SMALL`
	`int`	`public static final int STYLE_BOLD`
	`int`	`public static final int STYLE_ITALIC`
	`int`	`public static final int STYLE_PLAIN`
	`int`	`public static final int STYLE_UNDERLINED`
Methods		
	`int`	`public int charsWidth (char[] ch, int offset, int length)`
	`int`	`public int charWidth (char ch)`
	`int`	`public int getBaselinePosition ()`

Member Summary	
Font	`public static Font getDefaultFont ()`
int	`public int getFace ()`
Font	`public static Font getFont (int face, int style, int size)`
int	`public int getHeight ()`
int	`public int getSize ()`
int	`public int getStyle ()`
boolean	`public boolean isBold ()`
boolean	`public boolean isItalic ()`
boolean	`public boolean isPlain ()`
boolean	`public boolean isUnderlined ()`
int	`public int stringWidth (java.lang.String str)`
int	`public int substringWidth (java.lang.String str, int offset, int len)`

Fields

FACE_MONOSPACE

```
public static final int FACE_MONOSPACE
```

The "monospace" font face.

Value 32 is assigned to FACE_MONOSPACE.

FACE_PROPORTIONAL

```
public static final int FACE_PROPORTIONAL
```

The "proportional" font face.

Value 64 is assigned to FACE_PROPORTIONAL.

FACE_SYSTEM

```
public static final int FACE_SYSTEM
```

The "system" font face.

Value 0 is assigned to FACE_SYSTEM.

SIZE_LARGE

```
public static final int SIZE_LARGE
```

The "large" system-dependent font size.

Value 16 is assigned to SIZE_LARGE.

SIZE_MEDIUM

```
public static final int SIZE_MEDIUM
```

The "medium" system-dependent font size.

Value 0 is assigned to STYLE_MEDIUM.

SIZE_SMALL

```
public static final int SIZE_SMALL
```

The "small" system-dependent font size.

Value 8 is assigned to STYLE_SMALL.

STYLE_BOLD

```
public static final int STYLE_BOLD
```

The bold style constant. This may be combined with the other style constants for mixed styles.

Value 1 is assigned to STYLE_BOLD.

STYLE_ITALIC

```
public static final int STYLE_ITALIC
```

The italicized style constant. This may be combined with the other style constants for mixed styles.

Value 2 is assigned to STYLE_ITALIC.

STYLE_PLAIN

```
public static final int STYLE_PLAIN
```

The plain style constant. This may be combined with the other style constants for mixed styles.

Value 0 is assigned to STYLE_PLAIN.

STYLE_UNDERLINED

```
public static final int STYLE_UNDERLINED
```

The underlined style constant. This may be combined with the other style constants for mixed styles.

Value 4 is assigned to STYLE_UNDERLINED.

Methods

charsWidth(char[], int, int)

```
public int charsWidth (char[] ch, int offset, int length)
```

Returns the advance width of the characters in ch, starting at the specified offset and for the specified number of characters (length). The advance width is the amount by which the current point is moved from one character to the next in a line of text.

The offset and length parameters must specify a valid range of characters within the character array ch. The offset parameter must be within the range [0..(ch.length)]. The length parameter must be a non-negative integer such that (offset + length) <= ch.length.

Parameters:
`ch` - The array of characters

`offset` - The index of the first character to measure

`length` - The number of characters to measure

Returns: the width of the character range

Throws: `ArrayIndexOutOfBoundsException` - if offset and length specify an invalid range

`NullPointerException` - if ch is null

charWidth(char)

```
public int charWidth (char ch)
```

Gets the advance width of the specified character in this Font. The advance width is the amount by which the current point is moved from one character to the next in a line of text, and thus includes proper inter-character spacing. This spacing occurs to the right of the character.

Parameters:
`ch` - the character to be measured

Returns: the total advance width (a non-negative value)

getBaselinePosition()

```
public int getBaselinePosition ()
```

Gets the distance in pixels from the top of the text to the text's baseline.

Returns: the distance in pixels from the top of the text to the text's baseline

getDefaultFont()

```
public static Font getDefaultFont ()
```

Gets the default font of the system.

getFace()

```
public int getFace ()
```

Gets the face of the font.

Returns: one of FACE_SYSTEM, FACE_PROPORTIONAL, FACE_MONOSPACE

getFont(int, int, int)

```
public static Font getFont (int face, int style, int size)
```

Obtains an object representing a font having the specified face, style, and size. If a matching font does not exist, the system will attempt to provide the closest match. This method *always* returns a valid font object, even if it is not a close match to the request.

Parameters:

`face` - one of FACE_SYSTEM, FACE_MONOSPACE, or FACE_PROPORTIONAL

`style` - STYLE_PLAIN, or a combination of STYLE_BOLD, STYLE_ITALIC, and STYLE_UNDERLINED

`size` - one of SIZE_SMALL, SIZE_MEDIUM, or SIZE_LARGE

Returns: instance the nearest font found

Throws: `IllegalArgumentException` - if face, style, or size are not legal values

getHeight()

```
public int getHeight ()
```

Gets the standard height of a line of text in this font. This value includes sufficient spacing to ensure that lines of text painted this distance from anchor point to anchor point are spaced as intended by the font designer and the device. This extra space (leading) occurs below the text.

Returns: standard height of a line of text in this font (a non-negative value)

getSize()

```
public int getSize ()
```

Gets the size of the font.

Returns: one of SIZE_SMALL, SIZE_MEDIUM, SIZE_LARGE

getStyle()

```
public int getStyle ()
```

Gets the style of the font. The value is an OR'ed combination of STYLE_BOLD, STYLE_ITALIC, and STYLE_UNDERLINED; or the value is zero (STYLE_PLAIN).

Returns: style of the current font

See Also: `public boolean isPlain ()`, `public boolean isBold ()`, `public boolean isItalic ()`

isBold()

`public boolean isBold ()`

Returns true if the font is bold.

Returns: true if font is bold

See Also: `public int getStyle ()`

isItalic()

`public boolean isItalic ()`

Returns true if the font is italic.

Returns: true if font is italic

See Also: `public int getStyle ()`

isPlain()

`public boolean isPlain ()`

Returns true if the font is plain.

Returns: true if font is plain

See Also: `public int getStyle ()`

isUnderlined()

`public boolean isUnderlined ()`

Returns true if the font is underlined.

Returns: true if font is underlined

See Also: `public int getStyle ()`

stringWidth(String)

`public int stringWidth (java.lang.String str)`

Gets the total advance width for showing the specified String in this Font. The advance width is the amount by which the current point is moved from one character to the next in a line of text.

Parameters:
`str` - the String to be measured.

Returns: the total advance width

Throws: `NullPointerException` - if str is null

substringWidth(String, int, int)

```
public int substringWidth (java.lang.String str, int offset, int len)
```

Gets the total advance width for showing the specified substring in this Font. The advance width is the amount by which the current point is moved from one character to the next in a line of text.

The offset and length parameters must specify a valid range of characters within str. The offset parameter must be within the range [0..(str.length())]. The length parameter must be a non-negative integer such that (offset + length) <= str.length().

Parameters:

`str` - the String to be measured.

`offset` - zero-based index of first character in the substring

`len` - length of the substring.

Returns: the total advance width

Throws: `StringIndexOutOfBoundsException` - if offset and length specify an invalid range

`NullPointerException` - if str is null

javax.microedition.lcdui

Form

Syntax

```
public class Form extends Screen
```

```
Displayable
  |
  +--Screen
        |
        +--javax.microedition.lcdui.Form
```

Description

A Form is a Screen that contains an arbitrary mixture of items: images, read-only text fields, editable text fields, editable date fields, gauges, and choice groups. In general, any subclass of the `Item` class may be contained within a form. The implementation handles layout, traversal, and scrolling. None of the components contained within has any internal scrolling; the entire contents scrolls together. Note that this differs from the behavior of other classes, the List for example, where only the interior scrolls.

The items contained within a Form may be edited using append, delete, insert, and set methods. Items within a Form are referred to by their indexes, which are consecutive integers in the range from zero to size()-1, with zero referring to the first item and size()-1 to the last item.

An item may be placed within at most one Form. If the application attempts to place an item into a Form, and the item is already owned by this or another Form, an IllegalStateException is thrown. The application must remove the item from its currently containing Form before inserting it into the new Form.

As with other screens, the layout policy in most devices is vertical. In forms this applies to items involving user input. So, a new line is always started for focusable items like TextField, DateField, Gauge or ChoiceGroup.

Strings and images, which do not involve user interactions, behave differently; they are filled in horizontal lines, unless newline is embedded in the string or layout directives of the `ImageItem` force a new line. Contents will be wrapped (for text) or clipped (for images) to fit the width of the display, and scrolling will occur vertically as necessary. There will be no horizontal scrolling.

If the Form is visible on the display when changes to its contents are requested by the application, the changes take place immediately. That is, applications need not take any special action to refresh a Form's display after its contents have been modified.

When a Form is present on the display the user can interact with it and its Items indefinitely (for instance, traversing from Item to Item and possibly scrolling). These traversing and scrolling operations do not cause application-visible events. The system notifies the application when the user modifies the state of an interactive Item contained within the Form. This notification is accomplished by calling the `public void itemStateChanged (Item item)` method of the listener declared to the Form with the `public void setItemStateListener (ItemStateListener iListener)` method.

As with other Displayable objects, a Form can declare `Command` and declare a command listener with the `public void setCommandListener (CommandListener l)` method. `CommandListener` objects are distinct from `ItemStateListener` objects, and they are declared and invoked separately.

Notes for application developers:

- Although this class allows creation of arbitrary combination of components the application developers should keep the small screen size in mind. Form is designed to contain a *small number of closely related* UI elements.
- If the number of items does not fit on the screen, the implementation may choose to make it scrollable or to fold some components so that a new screen is popping up when the element is edited.

See Also: `Item`

Member Summary

Constructors	
	`public Form (java.lang.String title)`
	`public Form (java.lang.String title, Item[] items)`
Methods	
`int`	`public int append (Image img)`
`int`	`public int append (Item item)`
`int`	`public int append (java.lang.String str)`
`void`	`public void delete (int itemNum)`
`Item`	`public Item get (int itemNum)`
`void`	`public void insert (int itemNum, Item item)`
`void`	`public void set (int itemNum, Item item)`
`void`	`public void setItemStateListener (ItemStateListener iListener)`
`int`	`public int size ()`

Inherited Member Summary

Methods inherited from class `Screen`

`public Ticker getTicker ()`, `public java.lang.String getTitle ()`, `public void setTicker (Ticker ticker)`, `public void setTitle (java.lang.String s)`

Methods inherited from class `Displayable`

`public void addCommand (Command cmd)`, `public boolean isShown ()`, `public void removeCommand (Command cmd)`, `public void setCommandListener (CommandListener l)`

Constructors

Form(String)

```
public Form (java.lang.String title)
```

Creates a new, empty Form.

Parameters:
`title` - the Form's title, or null for no title

Form(String, Item[])

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

Creates a new Form with the specified contents. This is identical to creating an empty Form and then using a set of `append` methods. The items array may be null, in which case the Form is created empty. If the items array is non-null, each element must be a valid Item not already contained within another Form.

Parameters:
`title` - the Form's title string

`items` - the array of items to be placed in the Form, or null if there are no items

Throws: `IllegalStateException` - if one of the items is already owned by another container

`NullPointerException` - if an element of the items array is null

Methods

append(Image)

`public int append (Image img)`

Adds an item consisting of one Image to the form. The effect visible to the application is identical to

`append(new ImageItem(null, img, ImageItem.LAYOUT_DEFAULT, null))`

Parameters:
`img` - the image to be added

Returns: the assigned index of the Item

Throws: `IllegalArgumentException` - if the image is mutable

`NullPointerException` - if img is null

append(Item)

`public int append (Item item)`

Adds an Item into the Form. Strings are filled so that current line is continued if possible. If the text width is greater that the remaining horizontal space on the current line, the implementation inserts a new line and appends the rest of the text. Whenever possible the implementation should avoid breaking words into two lines. Instead, occurrences of white space (space or tab) should be used as potential places for splitting the lines. Also, a newline character in the string causes starting of a new line.

Images are laid out in the same manner as strings, unless the layout directives of `ImageItem` specify otherwise. Focusable items (TextField, ChoiceGroup, DateField, and Gauge) are placed on their own horizontal lines.

Parameters:
`item` - the `Item` to be added.

Returns: the assigned index of the Item

Throws: `IllegalStateException` - if the item is already owned by a container

`NullPointerException` - if item is null

append(String)

```
public int append (java.lang.String str)
```

Adds an item consisting of one String to the form. The effect visible to the application is identical to

`append(new StringItem(null, str))`

Parameters:
`str` - the String to be added

Returns: the assigned index of the Item

Throws: `NullPointerException` - if str is null

delete(int)

```
public void delete (int itemNum)
```

Deletes the Item referenced by itemNum. The size of the Form shrinks by one. It is legal to delete all items from a Form. The itemNum parameter must be within the range [0..size()-1], inclusive.

Parameters:
`itemNum` - the index of the item to be deleted

Throws: `IndexOutOfBoundsException` - if itemNum is invalid

get(int)

```
public Item get (int itemNum)
```

Gets the item at given position. The contents of the Form are left unchanged. The itemNum parameter must be within the range [0..size()-1], inclusive.

Parameters:
`itemNum` - the index of item

Returns: the item at the given position

Throws: `IndexOutOfBoundsException` - if itemNum is invalid

insert(int, Item)

`public void insert (int itemNum, Item item)`

Inserts an item into the Form just prior to the item specified. The size of the Form grows by one. The itemNum parameter must be within the range [0..size()], inclusive. The index of the last item is size()-1, and so there is actually no item whose index is size(). If this value is used for itemNum, the new item is inserted immediately after the last item. In this case, the effect is identical to `public int append (Item item)` .

The semantics are otherwise identical to `public int append (Item item)` .

Parameters:
`itemNum` - the index where insertion is to occur

`item` - the item to be inserted

Throws: `IndexOutOfBoundsException` - if itemNum is invalid

`IllegalStateException` - if the item is already owned by a container

`NullPointerException` - if item is null

set(int, Item)

`public void set (int itemNum, Item item)`

Sets the item referenced by itemNum to the specified item, replacing the previous item. The previous item is removed from this Form. The itemNum parameter must be within the range [0..size()-1], inclusive.

The end result is equal to

```
insert(n, item); delete(n+1);
```

Parameters:
`itemNum` - the index of the item to be replaced

`item` - the new item to be placed in the Form

Throws: `IndexOutOfBoundsException` - if itemNum is invalid

`IllegalStateException` - if the item is already owned by a container

`NullPointerException` - if item is null

setItemStateListener(ItemStateListener)

`public void setItemStateListener (ItemStateListener iListener)`

Sets the ItemStateListener for the Form, replacing any previous ItemStateListener. If iListener is null, simply removes the previous ItemStateListener.

Parameters:
`iListener` - the new listener, or null to remove it

size()

```
public int size ()
```

Gets the number of items in the Form.

Returns: the number of items

javax.microedition.lcdui

Gauge

Syntax

```
public class Gauge extends Item
```

```
Item
  |
  +--javax.microedition.lcdui.Gauge
```

Description

The Gauge class implements a bar graph display of a value intended for use in a form. Gauge is optionally interactive. The values accepted by the object are small integers in the range zero through a maximum value established by the application. The application is expected to normalize its values into this range. The device is expected to normalize this range into a smaller set of values for display purposes. Doing so will not change the actual value contained within the object. The range of values specified by the application may be larger than the number of distinct visual states possible on the device, so more than one value may have the same visual representation.

For example, consider a Gauge object that has a range of values from zero to 99, running on a device that displays the Gauge's approximate value using a set of one to ten bars. The device might show one bar for values zero through nine, two bars for values ten through 19, three bars for values 20 through 29, and so forth.

A Gauge may be interactive or non-interactive. Applications may set or retrieve the Gauge's value at any time regardless of the interaction mode. The implementation may change the visual appearance of the bar graph depending on whether the object is created in interactive mode.

In interactive mode, the user is allowed to modify the value. The user will always have the means to change the value up or down by one and may also have the means to change the value in greater increments. The user is prohibited from moving the value outside the established range. The expected behavior is that the application sets the initial value and then allows the user to modify the value thereafter. However, the application is not prohibited from modifying the value even while the user is interacting with it.

In many cases the only means for the user to modify the value will be to press a button to increase or decrease the value by one unit at a time. Therefore, applications should specify a range of no more than a few dozen values.

In non-interactive mode, the user is prohibited from modifying the value. An expected use of the non-interactive mode is as a "progress indicator" to give the user some feedback as progress occurs during a long-running operation. The application is expected to update the value periodically using the setValue() method. An application using the Gauge as a progress indicator should typically also attach a `public static final int STOP` command to the Form containing the Gauge to allow the user to halt the operation in progress.

Member Summary

Constructors	
	`public Gauge (java.lang.String label, boolean interactive, int maxValue, int initialValue)`
Methods	
`int`	`public int getMaxValue ()`
`int`	`public int getValue ()`
`boolean`	`public boolean isInteractive ()`
`void`	`public void setMaxValue (int maxValue)`

Member Summary

`void`	`public void setValue (int value)`

Inherited Member Summary

Methods inherited from class `Item`

`public java.lang.String getLabel ()`, `public void setLabel (java.lang.String label)`

Constructors

Gauge(String, boolean, int, int)

```
public Gauge (java.lang.String label, boolean interactive, int maxValue,
            int initialValue)
```

Creates a new Gauge object with the given label, in interactive or non-interactive mode, with the given maximum and initial values. The maximum value must be greater than zero, otherwise an exception is thrown. The initial value must be within the range zero to maxValue, inclusive. If the initial value is less than zero, the value is set to zero. If the initial value is greater than maxValue, it is set to maxValue.

Parameters:

`label` - the Gauge's label

`interactive` - tells whether the user can change the value

`maxValue` - the maximum value

`initialValue` - the initial value in the range [0..maxValue]

Throws: `IllegalArgumentException` - if maxValue is invalid

Methods

getMaxValue()

```
public int getMaxValue ()
```

Gets the maximum value of this Gauge object.

Returns: the maximum value of the Gauge

getValue()

```
public int getValue ()
```

Gets the current value of this Gauge object.

Returns: current value of the Gauge

isInteractive()

```
public boolean isInteractive ()
```

Tells whether the user is allowed to change the value of the Gauge.

Returns: a boolean indicating whether the Gauge is interactive

setMaxValue(int)

```
public void setMaxValue (int maxValue)
```

Sets the maximum value of this Gauge object. The new maximum value must be greater than zero, otherwise an exception is thrown. If the current value is greater than new maximum value, the current value is set to be equal to the new maximum value.

Parameters:
`maxValue` - the new maximum value

Throws: `IllegalArgumentException` - if maxValue is invalid

setValue(int)

```
public void setValue (int value)
```

Sets the current value of this Gauge object. If the value is less than zero, zero is used. If the current value is greater than the maximum value, the current value is set to be equal to the maximum value.

Parameters:
`value` - the new value

javax.microedition.lcdui

Graphics

Syntax

`public class Graphics`

`javax.microedition.lcdui.Graphics`

Description

Provides simple 2D geometric rendering capability. Drawing primitives are provided for text, images, lines, rectangles, and arcs. Rectangles and arcs may also be filled with a solid color. Rectangles may also be specified with rounded corners.

The only drawing operation provided is pixel replacement. The destination pixel value is simply replaced by the current pixel value specified in the graphics object being used for rendering. No facility for combining pixel values, such as raster-ops or alpha blending, is provided.

A 24-bit color model is provided, with 8 bits for each of red, green, and blue components of a color. Not all devices support a full 24 bits' worth of color and thus they will map colors requested by the application into colors available on the device. Facilities are provided in the `Display` class for obtaining device characteristics, such as whether color is available and how many distinct gray levels are available. This enables applications to adapt their behavior to a device without compromising device independence.

Graphics may be rendered directly to the display or to an off-screen image buffer. The destination of rendered graphics depends on the provenance of the graphics object. A graphics object for rendering to the display is passed to the Canvas object's `protected abstract void paint (Graphics g)` method. This is the only means by which a graphics object may be obtained whose destination is the display. Furthermore, applications may draw using this graphics object only for the duration of the paint() method.

A graphics object for rendering to an off-screen image buffer may be obtained by calling the `public Graphics getGraphics ()` method on the desired image. A graphics object so obtained may be held indefinitely by the application, and requests may be issued on this graphics object at any time.

The default coordinate system's origin is at the upper left-hand corner of the destination. The X-axis direction is positive towards the right, and the Y-axis direction is positive downwards. Applications may assume that horizontal and vertical distances in the coordinate system represent equal distances on the actual device display, that is, pixels are square. A facility is provided for translating the origin of the coordinate system. All coordinates are specified as integers.

The coordinate system represents locations between pixels, not the pixels themselves. Therefore, the first pixel in the upper left corner of the display lies in the square bounded by coordinates (0,0) , (1,0) , (0,1) , (1,1).

Under this definition, the semantics for fill operations are clear. Since coordinate grid lines lie between pixels, fill operations affect pixels that lie entirely within the region bounded by the coordinates of the operation. For example, the operation

```
        g.fillRect(0, 0, 3, 2)
```

paints exactly six pixels. (In this example, and in all subsequent examples, the variable `g` is assumed to contain a reference to a Graphics object.)

Each character of a font contains a set of pixels that forms the shape of the character. When a character is painted, the pixels forming the character's shape are filled with the Graphics object's current color, and the pixels not part of the character's shape are left untouched. The text drawing calls `public void drawChar (char character, int x, int y, int anchor)` , `public void drawChars (char[]`

`data, int offset, int length, int x, int y, int anchor)`, `public void drawString (java.lang.String str, int x, int y, int anchor)`, and `public void drawSubstring (java.lang.String str, int offset, int len, int x, int y, int anchor)` all draw text in this manner.

Lines, arcs, rectangles, and rounded rectangles may be drawn with either a SOLID or a DOTTED stroke style, as set by the `public void setStrokeStyle (int style)` method. The stroke style does not affect fill, text, and image operations.

For the SOLID stroke style, drawing operations are performed with a one-pixel wide pen that fills the pixel immediately below and to the right of the specified coordinate. Drawn lines touch pixels at both endpoints. Thus, the operation

```
        g.drawLine(0, 0, 0, 0)
```

paints exactly one pixel, the first pixel in the upper left corner of the display.

Drawing operations under the DOTTED stroke style will touch a subset of pixels that would have been touched under the SOLID stroke style. The frequency and length of dots is implementation-dependent. The endpoints of lines and arcs are not guaranteed to be drawn, nor are the corner points of rectangles guaranteed to be drawn. Dots are drawn by painting with the current color; spaces between dots are left untouched.

An artifact of the coordinate system is that the area affected by a fill operation differs slightly from the area affected by a draw operation given the same coordinates. For example, consider the operations

```
        g.fillRect(x, y, w, h); // 1
        g.drawRect(x, y, w, h); // 2
```

Statement (1) fills a rectangle w pixels wide and h pixels high. Statement (2) draws a rectangle whose left and top edges are within the area filled by statement (1). However, the bottom and right edges lie one pixel outside the filled area. This is counterintuitive, but it preserves the invariant that

```
        g.drawLine(x, y, x+w, y);
        g.drawLine(x+w, y, x+w, y+h);
        g.drawLine(x+w, y+h, x, y+h);
        g.drawLine(x, y+h, x, y);
```

has an effect identical to statement (2) above.

The exact pixels painted by drawLine() and drawArc() are not specified. Pixels touched by a fill operation must either exactly overlap or directly abut pixels touched by the corresponding draw operation. A fill operation must never leave a gap between the filled area and the pixels touched by the corresponding draw operation, nor may the fill operation touch pixels outside the area bounded by the corresponding draw operation.

There is a single clipping rectangle. Operations are provided for intersecting the current clip rectangle with a given rectangle and for setting the current clip rectangle outright. The only pixels touched by graphics operations are those that lie entirely within the clip rectangle. Pixels outside the clip rectangle are not affected by any graphics operations. It is legal to specify a clipping rectangle whose width or height is zero or negative. In this case the clipping rectangle is considered to be empty, that is, no pixels are contained within it. Therefore, if any graphics operations are issued under such a clipping rectangle, no pixels will be modified.

If a graphics operation is affected by the clip rectangle, the pixels touched by that operation must be the same ones that would be touched as if the clip rectangle did not affect the operation. For example, consider a clip rectangle (cx, cy, cw, ch) and a point (x1, y1) that lies outside this rectangle and a point (x2, y2) that lies within this rectangle. In the following code fragment,

```
g.setClip(0, 0, canvas.getWidth(), canvas.getHeight());
g.drawLine(x1, y1, x2, y2); // 3
g.setClip(cx, cy, cw, ch);
g.drawLine(x1, y1, x2, y2); // 4
```

The pixels touched by statement (4) must be identical to the pixels within (cx, cy, cw, ch) touched by statement (3).

Anchor Points

The drawing of text is based on "anchor points". Anchor points are used to minimize the amount of computation required when placing text. For example, in order to center a piece of text, an application needs to call stringWidth() or charWidth() to get the width and then perform a combination of subtraction and division to compute the proper location. The method to draw text is defined as follows:

```
public void drawString(String text, int x, int y, int anchor);
```

This method draws text in the current color, using the current font with its anchor point at (x,y). The definition of the anchor point must be one of the horizontal constants (LEFT, HCENTER, RIGHT) combined with one of the vertical constants (TOP, BASELINE, BOTTOM) using the logical OR operator.

Vertical centering of the text is not specified since it is not considered useful, it is hard to specify, and it is burdensome to implement. Thus, the VCENTER value is not allowed in the anchor point parameter of text drawing calls.

The actual position of the bounding box of the text relative to the (x, y) location is determined by the anchor point. These anchor points occur at named locations along the outer edge of the bounding box. Thus, if `f` is `g`'s current font (as returned by `g.getFont()`, the following calls will all have identical results:

```
g.drawString(str, x, y, TOP|LEFT);
g.drawString(str, x + f.stringWidth(str)/2, y, TOP|HCENTER);
g.drawString(str, x + f.stringWidth(str), y, TOP|RIGHT);
g.drawString(str, x,
    y + f.getBaselinePosition(), BASELINE|LEFT);
g.drawString(str, x + f.stringWidth(str)/2,
    y + f.getBaselinePosition(), BASELINE|HCENTER);
g.drawString(str, x + f.stringWidth(str),
    y + f.getBaselinePosition(), BASELINE|RIGHT);
drawString(str, x,
    y + f.getHeight(), BOTTOM|LEFT);
drawString(str, x + f.stringWidth(str)/2,
    y + f.getHeight(), BOTTOM|HCENTER);
drawString(str, x + f.stringWidth(str),
    y + f.getHeight(), BOTTOM|RIGHT);
```

For text drawing, the inter-character and inter-line spacing (leading) specified by the font designer are included as part of the values returned in the `public int stringWidth (java.lang.String str)` and `public int getHeight ()` calls of class `Font` . For example, given the following code:

```
// (5)
g.drawString(string1+string2, x, y, TOP|LEFT);
// (6)
g.drawString(string1, x, y, TOP|LEFT);
g.drawString(string2, x + f.stringWidth(string1), y, TOP|LEFT);
```

Code fragments (5) and (6) behave identically. This occurs because f.stringWidth() includes the inter-character spacing. Similarly, reasonable vertical spacing may be achieved simply by adding the font height to the Y-position of subsequent lines. For example:

```
        g.drawString(string1, x, y, TOP|LEFT);
        g.drawString(string2, x, y + f.fontHeight(), TOP|LEFT);
```

draws string1 and string2 on separate lines with an appropriate amount of inter-line spacing.

The stringWidth() of the string and the fontHeight() of the font in which it is drawn define the size of the bounding box of a piece of text. As described above, this box includes inter-line and inter-character spacing. The implementation is required to put this space below and to right of the pixels actually belonging to the characters drawn. Applications that wish to position graphics closely with respect text (for example, to paint a rectangle around a string of text) may assume that there is space below and to the right of a string and that there is *no* space above and to the left of the string.

Anchor points are also used for positioning of images. Similar to text drawing, the anchor point for an image specifies the point on the bounding rectangle of the destination that is to positioned at the (x,y) location given in the graphics request. Unlike text, vertical centering of images is well-defined, and thus the VCENTER value may be used within the anchor point parameter of image drawing requests. Because images have no notion of a baseline, the BASELINE value may not be used within the anchor point parameter of image drawing requests.

Member Summary

Fields		
	int	public static final int BASELINE
	int	public static final int BOTTOM
	int	public static final int DOTTED
	int	public static final int HCENTER
	int	public static final int LEFT
	int	public static final int RIGHT
	int	public static final int SOLID
	int	public static final int TOP
	int	public static final int VCENTER
Methods		
	void	public void clipRect (int x, int y, int width, int height)
	void	public void drawArc (int x, int y, int width, int height, int startAngle, int arcAngle)
	void	public void drawChar (char character, int x, int y, int anchor)
	void	public void drawChars (char[] data, int offset, int length, int x, int y, int anchor)
	void	public void drawImage (Image img, int x, int y, int anchor)
	void	public void drawLine (int x1, int y1, int x2, int y2)
	void	public void drawRect (int x, int y, int width, int height)
	void	public void drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)
	void	public void drawString (java.lang.String str, int x, int y, int anchor)
	void	public void drawSubstring (java.lang.String str, int offset, int len, int x, int y, int anchor)
	void	public void fillArc (int x, int y, int width, int height, int startAngle, int arcAngle)
	void	public void fillRect (int x, int y, int width, int height)
	void	public void fillRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)
	int	public int getBlueComponent ()
	int	public int getClipHeight ()

Member Summary

`int`	`public int getClipWidth ()`
`int`	`public int getClipX ()`
`int`	`public int getClipY ()`
`int`	`public int getColor ()`
`Font`	`public Font getFont ()`
`int`	`public int getGrayScale ()`
`int`	`public int getGreenComponent ()`
`int`	`public int getRedComponent ()`
`int`	`public int getStrokeStyle ()`
`int`	`public int getTranslateX ()`
`int`	`public int getTranslateY ()`
`void`	`public void setClip (int x, int y, int width, int height)`
`void`	`public void setColor (int RGB)`
`void`	`public void setColor (int red, int green, int blue)`
`void`	`public void setFont (Font font)`
`void`	`public void setGrayScale (int value)`
`void`	`public void setStrokeStyle (int style)`
`void`	`public void translate (int x, int y)`

Fields

BASELINE

`public static final int BASELINE`

Constant for positioning the anchor point at the baseline of text.

Value 64 is assigned to BASELINE.

BOTTOM

`public static final int BOTTOM`

Constant for positioning the anchor point of text and images below the text or image.

Value 32 is assigned to BOTTOM.

DOTTED

`public static final int DOTTED`

Constant for the DOTTED stroke style.

Value 1 is assigned to DOTTED.

HCENTER

```
public static final int HCENTER
```

Constant for centering text and images horizontally around the anchor point

Value 1 is assigned to HCENTER.

LEFT

```
public static final int LEFT
```

Constant for positioning the anchor point of text and images to the left of the text or image.

Value 4 is assigned to LEFT.

RIGHT

```
public static final int RIGHT
```

Constant for positioning the anchor point of text and images to the right of the text or image.

Value 8 is assigned to RIGHT.

SOLID

```
public static final int SOLID
```

Constant for the SOLID stroke style.

Value 0 is assigned to SOLID.

TOP

```
public static final int TOP
```

Constant for positioning the anchor point of text and images above the text or image.

Value 16 is assigned to TOP.

VCENTER

```
public static final int VCENTER
```

Constant for centering images vertically around the anchor point.

Value 2 is assigned to VCENTER.

Methods

clipRect(int, int, int, int)

`public void clipRect (int x, int y, int width, int height)`

Intersects the current clip with the specified rectangle. The resulting clipping area is the intersection of the current clipping area and the specified rectangle. This method can only be used to make the current clip smaller. To set the current clip larger, use the setClip method. Rendering operations have no effect outside of the clipping area.

Parameters:

`x` - the x coordinate of the rectangle to intersect the clip with

`y` - the y coordinate of the rectangle to intersect the clip with

`width` - the width of the rectangle to intersect the clip with

`height` - the height of the rectangle to intersect the clip with

See Also: `public void setClip (int x, int y, int width, int height)`

drawArc(int, int, int, int, int, int)

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

Draws the outline of a circular or elliptical arc covering the specified rectangle, using the current color and stroke style.

The resulting arc begins at `startAngle` and extends for `arcAngle` degrees, using the current color. Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation.

The center of the arc is the center of the rectangle whose origin is (*x*, *y*) and whose size is specified by the `width` and `height` arguments.

The resulting arc covers an area `width + 1` pixels wide by `height + 1` pixels tall. If either width or height is less than zero, nothing is drawn.

The angles are specified relative to the non-square extents of the bounding rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the bounding rectangle. As a result, if the bounding rectangle is noticeably longer in one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the bounds.

Parameters:

`x` - the *x* coordinate of the upper-left corner of the arc to be drawn.

`y` - the *y* coordinate of the upper-left corner of the arc to be drawn.

`width` - the width of the arc to be drawn

`height` - the height of the arc to be drawn

`startAngle` - the beginning angle

`arcAngle` - the angular extent of the arc, relative to the start angle.

See Also: `public void fillArc (int x, int y, int width, int height, int startAngle, int arcAngle)`

drawChar(char, int, int, int)

`public void drawChar (char character, int x, int y, int anchor)`

Draws the specified character using the current font and color.

Parameters:

`character` - the character to be drawn

`x` - the x coordinate of the anchor point

`y` - the y coordinate of the anchor point

`anchor` - the anchor point for positioning the text; see anchor points a valid range within the data array

Throws: `IllegalArgumentException` - if anchor is not a legal value

See Also: `public void drawString (java.lang.String str, int x, int y, int anchor)`, `public void drawChars (char[] data, int offset, int length, int x, int y, int anchor)`

drawChars(char[], int, int, int, int, int)

`public void drawChars (char[] data, int offset, int length, int x, int y, int anchor)`

Draws the specified characters using the current font and color.

Parameters:

`data` - the array of characters to be drawn

`offset` - the start offset in the data

`length` - the number of characters to be drawn

`x` - the x coordinate of the anchor point

`y` - the y coordinate of the anchor point

`anchor` - the anchor point for positioning the text; see anchor points

Throws: `ArrayIndexOutOfBoundsException` - if offset and length do not specify a valid range within the data array

`IllegalArgumentException` - if anchor is not a legal value

`NullPointerException` - if data is null

See Also: `public void drawString (java.lang.String str, int x, int y, int anchor)`

drawImage(Image, int, int, int)

`public void drawImage (Image img, int x, int y, int anchor)`

Draws the specified image by using the anchor point. The image can be drawn in different positions relative to the anchor point by passing the appropriate position constants. See anchor points.

Parameters:

`img` - the specified image to be drawn

`x` - the x coordinate of the anchor point `y` - the y coordinate of the anchor point

`anchor` - the anchor point for positioning the image

Throws: `IllegalArgumentException` - if anchor is not a legal value

`NullPointerException` - if img is null

See Also: `Image`

---

drawLine(int, int, int, int)

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

Draws a line between the coordinates (x1,y1) and (x2,y2) using the current color and stroke style.

Parameters:

`x1` - the x coordinate of the start of the line

`y1` - the y coordinate of the start of the line

`x2` - the x coordinate of the end of the line

`y2` - the y coordinate of the end of the line

---

drawRect(int, int, int, int)

```
public void drawRect (int x, int y, int width, int height)
```

Draws the outline of the specified rectangle using the current color and stroke style. The resulting rectangle will cover an area (width + 1) pixels wide by (height + 1) pixels tall. If either width or height is less than zero, nothing is drawn.

Parameters:

`x` - the x coordinate of the rectangle to be drawn

`y` - the y coordinate of the rectangle to be drawn

`width` - the width of the rectangle to be drawn

`height` - the height of the rectangle to be drawn

See Also: `public void fillRect (int x, int y, int width, int height)`

---

drawRoundRect(int, int, int, int, int, int)

```
public void drawRoundRect (int x, int y, int width, int height, int arcWidth,
            int arcHeight)
```

Draws the outline of the specified rounded corner rectangle using the current color and stroke style. The resulting rectangle will cover an area (width + 1) pixels wide by (height + 1) pixels tall. If either width or height is less than zero, nothing is drawn.

Parameters:

`x` - the x coordinate of the rectangle to be drawn

`y` - the y coordinate of the rectangle to be drawn

`width` - the width of the rectangle to be drawn

`height` - the height of the rectangle to be drawn `arcWidth` - the horizontal diameter of the arc at the four corners

`arcHeight` - the vertical diameter of the arc at the four corners

See Also: `public void fillRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)`

---

drawString(String, int, int, int)

```
public void drawString (java.lang.String str, int x, int y, int anchor)
```

Draws the specified String using the current font and color. The x,y position is the position of the anchor point. See anchor points.

Parameters:

`str` - the String to be drawn

`x` - the x coordinate of the anchor point

`y` - the y coordinate of the anchor point

`anchor` - the anchor point for positioning the text

Throws: `NullPointerException` - if str is null

`IllegalArgumentException` - if anchor is not a legal value

See Also: `public void drawChars (char[] data, int offset, int length, int x, int y, int anchor)`

---

drawSubstring(String, int, int, int, int, int)

```
public void drawSubstring (java.lang.String str, int offset, int len, int x, int y,
            int anchor)
```

Draws the specified String using the current font and color. The x,y position is the position of the anchor point. See anchor points.

Parameters:

`str` - the String to be drawn

`offset` - zero-based index of first character in the substring

`len` - length of the substring

`x` - the x coordinate of the anchor point

`y` - the y coordinate of the anchor point

`anchor` - the anchor point for positioning the text

Throws: `StringIndexOutOfBoundsException` - if offset and length do not specify a valid range within the String str

`IllegalArgumentException` - if anchor is not a legal value

`NullPointerException` - if str is null

See Also: `public void drawString (java.lang.String str, int x, int y, int anchor)`

fillArc(int, int, int, int, int, int)

```
public void fillArc (int x, int y, int width, int height, int startAngle, int arcAngle)
```

Fills a circular or elliptical arc covering the specified rectangle.

The resulting arc begins at `startAngle` and extends for `arcAngle` degrees. Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation.

The center of the arc is the center of the rectangle whose origin is (*x*, *y*) and whose size is specified by the `width` and `height` arguments.

If either width or height is zero or less, nothing is drawn.

The filled region consists of the "pie wedge" region bounded by the arc segment as if drawn by drawArc(), the radius extending from the center to this arc at `startAngle` degrees, and radius extending from the center to this arc at `startAngle + arcAngle` degrees.

The angles are specified relative to the non-square extents of the bounding rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the bounding rectangle. As a result, if the bounding rectangle is noticeably longer in one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the bounds.

Parameters:

`x` - the *x* coordinate of the upper-left corner of the arc to be filled.

`y` - the *y* coordinate of the upper-left corner of the arc to be filled.

`width` - the width of the arc to be filled

`height` - the height of the arc to be filled

`startAngle` - the beginning angle.

`arcAngle` - the angular extent of the arc, relative to the start angle.

See Also: `public void drawArc (int x, int y, int width, int height, int startAngle, int arcAngle)`

fillRect(int, int, int, int)

```
public void fillRect (int x, int y, int width, int height)
```

Fills the specified rectangle with the current color. If either width or height is zero or less, nothing is drawn.

Parameters:

`x` - the x coordinate of the rectangle to be filled

`y` - the y coordinate of the rectangle to be filled

`width` - the width of the rectangle to be filled

`height` - the height of the rectangle to be filled

See Also: `public void drawRect (int x, int y, int width, int height)`

fillRoundRect(int, int, int, int, int, int)

```
public void fillRoundRect (int x, int y, int width, int height, int arcWidth,
            int arcHeight)
```

Fills the specified rounded corner rectangle with the current color. If either width or height is zero or less, nothing is drawn.

Parameters:

`x` - the x coordinate of the rectangle to be filled

`y` - the y coordinate of the rectangle to be filled

`width` - the width of the rectangle to be filled

`height` - the height of the rectangle to be filled

`arcWidth` - the horizontal diameter of the arc at the four corners

`arcHeight` - the vertical diameter of the arc at the four corners

See Also: `public void drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)`

getBlueComponent()

```
public int getBlueComponent ()
```

Gets the blue component of the current color.

Returns: integer value in range 0-255

See Also: `public void setColor (int red, int green, int blue)`

getClipHeight()

```
public int getClipHeight ()
```

Gets the height of the current clipping area.

Returns: height of the current clipping area.

See Also: `public void clipRect (int x, int y, int width, int height)`, `public void setClip (int x, int y, int width, int height)`

getClipWidth()

```
public int getClipWidth ()
```

Gets the width of the current clipping area.

Returns: width of the current clipping area.

See Also: `public void clipRect (int x, int y, int width, int height)`, `public void setClip (int x, int y, int width, int height)`

getClipX()

`public int getClipX ()`

Gets the X offset of the current clipping area, relative to the coordinate system origin of this graphics context. Separating the getClip operation into two methods returning integers is more performance and memory efficient than one getClip() call returning an object.

Returns: X offset of the current clipping area

See Also: `public void clipRect (int x, int y, int width, int height)`, `public void setClip (int x, int y, int width, int height)`

getClipY()

`public int getClipY ()`

Gets the Y offset of the current clipping area, relative to the coordinate system origin of this graphics context. Separating the getClip operation into two methods returning integers is more performance and memory efficient than one getClip() call returning an object.

Returns: Y offset of the current clipping area

See Also: `public void clipRect (int x, int y, int width, int height)`, `public void setClip (int x, int y, int width, int height)`

getColor()

`public int getColor ()`

Gets the current color.

Returns: an integer in form 0x00RRGGBB

See Also: `public void setColor (int red, int green, int blue)`

getFont()

`public Font getFont ()`

Gets the current font.

Returns: current font

See Also: `Font`, `public void setFont (Font font)`

getGrayScale()

`public int getGrayScale ()`

Gets the current grayscale value of the color being used for rendering operations. If the color was set by setGrayScale(), that value is simply returned. If the color was set by one of the methods that allows setting of the red, green, and blue components, the value returned is computed from the RGB color components (possibly in a device-specific fashion) that best approximates the brightness of that color.

Returns: integer value in range 0-255

getGreenComponent()

`public int getGreenComponent ()`

Gets the green component of the current color.

Returns: integer value in range 0-255

See Also: `public void setColor (int red, int green, int blue)`

getRedComponent()

`public int getRedComponent ()`

Gets the red component of the current color.

Returns: integer value in range 0-255

See Also: `public void setColor (int red, int green, int blue)`

getStrokeStyle()

`public int getStrokeStyle ()`

Gets the stroke style used for drawing operations.

Returns: stroke style, SOLID or DOTTED

getTranslateX()

`public int getTranslateX ()`

Gets the X coordinate of the translated origin of this graphics context.

Returns: X of current origin

getTranslateY()

`public int getTranslateY ()`

Gets the Y coordinate of the translated origin of this graphics context.

Returns: Y of current origin

setClip(int, int, int, int)

`public void setClip (int x, int y, int width, int height)`

Sets the current clip to the rectangle specified by the given coordinates. Rendering operations have no effect outside of the clipping area.

Parameters:

`x` - the x coordinate of the new clip rectangle

`y` - the y coordinate of the new clip rectangle

`width` - the width of the new clip rectangle `height` - the height of the new clip rectangle

See Also: `public void clipRect (int x, int y, int width, int height)`

---

setColor(int)

`public void setColor (int RGB)`

Sets the current color to the specified RGB values. All subsequent rendering operations will use this specified color. The RGB value passed in is interpreted with the least significant eight bits giving the blue component, the next eight more significant bits giving the green component, and the next eight more significant bits giving the red component. That is to say, the color component is specified in the form of 0x00RRGGBB. The high order byte of this value is ignored.

Parameters:
`RGB` - the color being set

---

setColor(int, int, int)

`public void setColor (int red, int green, int blue)`

Sets the current color to the specified RGB values. All subsequent rendering operations will use this specified color.

Parameters:
`red` - The red component of the color being set in range 0-255.

`green` - The green component of the color being set in range 0-255.

`blue` - The blue component of the color being set in range 0-255.

Throws: `IllegalArgumentException` - if any of the color components are outside of range 0-255.

---

setFont(Font)

`public void setFont (Font font)`

Sets the font for all subsequent text rendering operations. If font is null, it is equivalent to `setFont(Font.getDefaultFont())`.

Parameters:
`font` - the specified font

See Also: `Font`, `public Font getFont ()`, `public void drawString (java.lang.String str, int x, int y, int anchor)`, `public void drawChars (char[] data, int offset, int length, int x, int y, int anchor)`

---

setGrayScale(int)

`public void setGrayScale (int value)`

Sets the current grayscale to be used for all subsequent rendering operations. For monochrome displays, the behavior is clear. For color displays, this sets the color for all subsequent drawing operations to be a gray color equivalent to the value passed in. The value must be in the range 0-255.

Parameters:
`value` - the desired grayscale value

Throws: `IllegalArgumentException` - if the gray value is out of range

---

setStrokeStyle(int)

```
public void setStrokeStyle (int style)
```

Sets the stroke style used for drawing lines, arcs, rectangles, and rounded rectangles. This does not affect fill, text, and image operations.

Parameters:
`style` - can be SOLID or DOTTED

Throws: `IllegalArgumentException` - if the style is illegal

---

translate(int, int)

```
public void translate (int x, int y)
```

Translates the origin of the graphics context to the point (x, y) in the current coordinate system. All coordinates used in subsequent rendering operations on this graphics context will be relative to this new origin.

The effect of calls to translate() are cumulative. For example, calling translate(1, 2) and then translate(3, 4) results in a translation of (4, 6).

The application can set an absolute origin (ax, ay) using the following technique:

```
g.translate(ax - g.getTranslateX(), ay - g.getTranslateY())
```

Parameters:
`x` - the x coordinate of the new translation origin

`y` - the y coordinate of the new translation origin

See Also: `public int getTranslateX ()`, `public int getTranslateY ()`

javax.microedition.lcdui

Image

Syntax

`public class Image`

`javax.microedition.lcdui.Image`

Description

The Image class is used to hold graphical image data. Image objects exist independently of the display device. They exist only in off-screen memory and will not be painted on the display unless an explicit command is issued by the application (such as within the paint() method of a Canvas) or when an Image object is placed within a Form screen or an Alert screen and that screen is made current.

Images are either *mutable* or *immutable* depending upon how they are created. Immutable images are generally created by loading image data from resource bundles, from files, or from the network. They may not be modified once created. Mutable images are created in off-screen memory. The application may paint into them after having created a Graphics object expressly for this purpose. Images to be placed within Alert, Choice, Form, or ImageItem objects are required to be immutable because the implementation may use them to update the display at any time, without notifying the application.

An immutable image may be created from a mutable image through the use of the `public static Image createImage (Image source)` method. It is possible to create a mutable copy of an immutable image using a technique similar to the following:

```
Image source; // the image to be copied
source = Image.createImage(...);
Image copy = Image.createImage(source.getWidth(), source.getHeight());
Graphics g = copy.getGraphics();
g.drawImage(source, 0, 0, TOP|LEFT);
```

It is also possible to use this technique to create a copy of a subrectangle of an image, by altering the width and height parameters of the createImage() call that creates the destination image and by altering the x and y parameters of the drawImage() call.

PNG Image Format

Implementations are required to support images stored in the PNG (Portable Network Graphics) format, version 1.0.

Note: The remainder of this section consists of a summary of the minimum set of features required for PNG conformance, along with some considerations for MIDP implementors and application developers. The information about PNG has been condensed from the *PNG (Portable Network Graphics) Specification, Version 1.0.* Any discrepancies between this section and the *PNG Specification* should be resolved in favor of the *PNG Specification.*

All of the 'critical' chunks specified by PNG must be supported. The paragraphs below describe these critical chunks.

The IHDR chunk. MIDP devices must handle the following values in the IHDR chunk:

- All positive values of width and height are supported; however, a very large image may not be readable because of memory constraints.
- All color types are supported, although the appearance of the image will be dependent on the capabilities of the device's screen. Color types that include alpha channel data are supported; however, the implementation may ignore all alpha channel information and treat all pixels as opaque.
- All bit depth values for the given color type are supported.
- Compression method 0 (deflate) is the only supported compression method. This is the same compression method that is used for jar files, and so the decompression (inflate) code may be shared between the jar decoding and PNG decoding implementations.
- The filter method represents a series of encoding schemes that may be used to optimize compression. The PNG spec currently defines a single filter method (method 0) that is an adaptive filtering scheme with five basic filter types. Filtering is essential for optimal compression since it allows the deflate algorithm to exploit spatial similarities within the image. Therefore, MIDP devices must support all five filter types defined by filter method 0.
- MIDP devices are required to read PNG images that are encoded with either interlace method 0 (None) or interlace method 1 (Adam7). Image loading in MIDP is synchronous and cannot be overlapped with image rendering, and so there is no advantage for an application to use interlace method 1. Support for decoding interlaced images is required for compatibility with PNG and for the convenience of developers who may already have interlaced images available.

The PLTE chunk. Palette-based images must be supported.

The IDAT chunk. Image data may be encoded using any of the 5 filter types defined by filter method 0 (None, Sub, Up, Average, Paeth).

The IEND chunk. This chunk must be found in order for the image to be considered valid.

Ancillary chunk support. PNG defines several 'ancillary' chunks that may be present in a PNG image but are not critical for image decoding. A MIDP implementation *may* (but is not required to) support any of these chunks. The implementation *should* silently ignore any unsupported ancillary chunks that it encounters. The currently defined ancillary chunks are:

```
        bKGD cHRM gAMA hIST iCCP iTXt pHYs
        sBIT sPLT sRGB tEXt tIME tRNS zTXt
    Reference
```

PNG (Portable Network Graphics) Specification, Version 1.0. W3C Recommendation, October 1, 1996. http://www.w3.org/TR/REC-png.html. Also available as RFC 2083, http://www.ietf.org/rfc/rfc2083.txt.

Member Summary

Methods

`Image`	`public static Image createImage (byte[] imageData, int imageOffset, int imageLength)`
`Image`	`public static Image createImage (Image source)`
`Image`	`public static Image createImage (int width, int height)`
`Image`	`public static Image createImage (java.lang.String name)`
`Graphics`	`public Graphics getGraphics ()`
`int`	`public int getHeight ()`
`int`	`public int getWidth ()`
`boolean`	`public boolean isMutable ()`

Methods

createImage(byte[], int, int)

`public static Image createImage (byte[] imageData, int imageOffset, int imageLength)`

Creates an immutable image which is decoded from the data stored in the specified byte array at the specified offset and length. The data must be in a self-identifying image file format supported by the implementation, such as PNG.

The imageoffset and imagelength parameters specify a range of data within the imageData byte array. The imageOffset parameter specifies the offset into the array of the first data byte to be used. It must therefore lie within the range [0..(imageData.length-1)]. The imageLength parameter specifies the number of data bytes to be used. It must be a positive integer and it must not cause the range to extend beyond the end of the array. That is, it must be true that imageOffset + imageLength <= imageData.length.

This method is intended for use when loading an image from a variety of sources, such as from persistent storage or from the network.

Parameters:

`imageData` - the array of image data in a supported image format

`imageOffset` - the offset of the start of the data in the array

`imageLength` - the length of the data in the array

Returns: the created image

Throws: `ArrayIndexOutOfBoundsException` - if imageOffset and imageLength specify an invalid range

`NullPointerException` - if imageData is null

`IllegalArgumentException` - if imageData is incorrectly formatted or otherwise cannot be decoded

createImage(Image)

`public static Image createImage (Image source)`

Creates an immutable image from a source image. If the source image is mutable, an immutable copy is created and returned. If the source image is immutable, the implementation may simply return it without creating a new image.

This method is useful for placing images drawn off-screen into Alert, Choice, Form, and StringItem objects. The application can create an off-screen image using the `public static Image createImage (int width, int height)` method, draw into it using a Graphics object obtained with the `public Graphics getGraphics ()` method, and then create an immutable copy of it with this method. The immutable copy may then be placed into the Alert, Choice, Form, and StringItem objects.

Parameters:

`source` - the source image to be copied

Returns: the new, immutable image

Throws: `NullPointerException` - if source is null

createImage(int, int)

```
public static Image createImage (int width, int height)
```

Creates a new, mutable image for off-screen drawing. Every pixel within the newly created image is white. The width and height of the image must both be greater than zero.

Parameters:
`width` - the width of the new image, in pixels

`height` - the height of the new image, in pixels

Returns: the created image

Throws: `IllegalArgumentException` - if either width or height is zero or less

createImage(String)

```
public static Image createImage (java.lang.String name)
```

Creates an immutable image from decoded image data obtained from the named resource. The name parameter is a resource name as defined by `Class.getResourceAsStream(name)` .

Parameters:
`name` - the name of the resource containing the image data in one of the supported image formats

Returns: the created image

Throws: `NullPointerException` - if name is null

`java.io.IOException` - if the resource does not exist, the data cannot be loaded, or the image data cannot be decoded

getGraphics()

```
public Graphics getGraphics ()
```

Creates a new Graphics object that renders to this image. This image must be mutable; it is illegal to call this method on an immutable image. The mutability of an image may be tested with the isMutable() method.

The newly created Graphics object has the following properties:

- the destination is this Image object;
- the clip region encompasses the entire Image;
- the current color is black;
- the font is the same as the font returned by `public static Font getDefaultFont ()` ;
- the stroke style is `public static final int SOLID` ; and
- the origin of the coordinate system is located at the upper-left corner of the Image.

The lifetime of Graphics objects created using this method is indefinite. They may be used at any time, by any thread.

Returns: a Graphics object with this image as its destination

Throws: `IllegalStateException` - if the image is immutable

getHeight()

```
public int getHeight ()
```

Gets the height of the image in pixels.

Returns: height of the image

getWidth()

```
public int getWidth ()
```

Gets the width of the image in pixels.

Returns: width of the image

isMutable()

```
public boolean isMutable ()
```

Check if this image is mutable. Mutable images can be modified by rendering to them through a Graphics object obtained from the getGraphics() method of this object.

Returns: true if the image is mutable, false otherwise

javax.microedition.lcdui

ImageItem

Syntax

```
public class ImageItem extends Item
```

```
Item
  |
  +--javax.microedition.lcdui.ImageItem
```

Description

A class that provides layout control when Image objects are added to a `Form` or to an `Alert` .

Each ImageItem object contains a reference to an Image object. This image must be immutable. (If the image object were not required to be immutable, the application could paint into it at any time, potentially requiring the containing Form or Alert to be updated on every graphics call.) See the definition of the `Image` object for further details on image mutability how to create immutable images.

The value null may be specified for the image contents of an ImageItem. If this occurs (and if the label is also null) the ImageItem will occupy no space on the screen.

Each ImageItem object contains a layout field that is combined from the following values: LAYOUT_LEFT, LAYOUT_RIGHT, LAYOUT_CENTER, LAYOUT_NEWLINE_BEFORE, and LAYOUT_NEWLINE_AFTER. LAYOUT_LEFT + LAYOUT_RIGHT is equal to LAYOUT_CENTER. LAYOUT_DEFAULT may be specified, which indicates that the system should use its default layout policy for this ImageItem. The value of the layout field is merely a hint. Because of device constraints, such as limited screen size, the implementation may choose to ignore layout directions.

There are some implicit rules on how the layout directives can be combined:

- LAYOUT_DEFAULT cannot not be combined with any other directive. In fact, any other value will override LAYOUT_DEFAULT since its value is 0.
- LAYOUT_LEFT, LAYOUT_RIGHT and LAYOUT_CENTER are meant to be mutually exclusive.
- It usually makes sense to combine LAYOUT_LEFT, LAYOUT_RIGHT and LAYOUT_CENTER with LAYOUT_NEWLINE_BEFORE and LAYOUT_NEWLINE_AFTER.

The altText parameter specifies a string to be displayed in place of the image if the image exceeds the capacity of the display. The altText parameter may be null.

Member Summary

Fields		
	`int`	`public static final int LAYOUT_CENTER`
	`int`	`public static final int LAYOUT_DEFAULT`
	`int`	`public static final int LAYOUT_LEFT`
	`int`	`public static final int LAYOUT_NEWLINE_AFTER`
	`int`	`public static final int LAYOUT_NEWLINE_BEFORE`
	`int`	`public static final int LAYOUT_RIGHT`
Constructors		
		`public ImageItem (java.lang.String label, Image img, int layout, java.lang.String altText)`
Methods		
	`String`	`public java.lang.String getAltText ()`

Member Summary

`Image`	`public Image getImage ()`
`int`	`public int getLayout ()`
`void`	`public void setAltText (java.lang.String text)`
`void`	`public void setImage (Image img)`
`void`	`public void setLayout (int layout)`

Inherited Member Summary

Methods inherited from class `Item`

`public java.lang.String getLabel ()`, `public void setLabel (java.lang.String label)`

Fields

LAYOUT_CENTER

```
public static final int LAYOUT_CENTER
```

Image should be horizontally centered.

Value 3 is assigned to LAYOUT_CENTER.

LAYOUT_DEFAULT

```
public static final int LAYOUT_DEFAULT
```

Use the default formatting of the "container" of the image.

Value 0 is assigned to LAYOUT_DEFAULT.

LAYOUT_LEFT

```
public static final int LAYOUT_LEFT
```

Image should be close to left-edge of the drawing area.

Value 1 is assigned to LAYOUT_LEFT.

LAYOUT_NEWLINE_AFTER

```
public static final int LAYOUT_NEWLINE_AFTER
```

A new line should be started after the image is drawn.

Value 0x200 is assigned to LAYOUT_DEFAULT.

LAYOUT_NEWLINE_BEFORE

```
public static final int LAYOUT_NEWLINE_BEFORE
```

A new line should be started before the image is drawn.

Value 0x100 is assigned to LAYOUT_NEWLINE_BEFORE.

LAYOUT_RIGHT

```
public static final int LAYOUT_RIGHT
```

Image should be close to right-edge of the drawing area.

Value 2 is assigned to LAYOUT_RIGHT.

Constructors

ImageItem(String, Image, int, String)

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

Creates a new ImageItem with the given label, image, layout directive, and alternate text string.

Parameters:

`label` - the label string

`img` - the image, must be immutable

`layout` - a combination of layout directives

`altText` - the text that may be used in place of the image

Throws: `IllegalArgumentException` - if the image is mutable

`IllegalArgumentException` - if the layout value is not a legal combination of directives

Methods

getAltText()

```
public java.lang.String getAltText ()
```

Gets the text string to be used if the image exceeds the device's capacity to display it.

Returns: the alternate text value, or null if none

getImage()

`public Image getImage ()`

Gets the image contained within the ImageItem, or null if there is no contained image.

Returns: image used by the ImageItem

getLayout()

`public int getLayout ()`

Gets the layout directives used for placing the image.

Returns: a combination of layout directive values

setAltText(String)

`public void setAltText (java.lang.String text)`

Sets the alternate text of the ImageItem, or null if no alternate text is provided.

Parameters:
`text` - the new alternate text

setImage(Image)

`public void setImage (Image img)`

Sets the image object contained within the ImageItem. The image must be immutable. If img is null, the ImageItem is set to be empty.

Parameters:
`img` - the new image

Throws: `IllegalArgumentException` - if the image is mutable

setLayout(int)

`public void setLayout (int layout)`

Sets the layout directives.

Parameters:
`layout` - a combination of layout directive values

Throws: `IllegalArgumentException` - if the value of layout is not a valid combination of layout directives

javax.microedition.lcdui

Item

Syntax

`public abstract class Item`

`javax.microedition.lcdui.Item`

Direct Known Subclasses: `ChoiceGroup`, `DateField`, `Gauge`, `ImageItem`, `StringItem`, `TextField`

Description

A superclass for components that can be added to a `Form` and `Alert` . All Item objects have a label field, which is a string that is attached to the item. The label is typically displayed near the component when it is displayed within a screen. This means that the implementation tries to keep the label on the same horizontal row with the item or directly above the item. If the screen is scrolling, the implementation tries to keep the label visible at the same time with the Item.

In some cases, when the user attempts to interact with an Item, the system will switch to a system-generated screen where the actual interaction takes places. If this occurs, the label will generally be carried along and displayed within this new screen in order to provide the user with some context for the operation. For this reason it is recommended that applications supply a label to all interactive Item objects. However, this is not required, and a null value for a label is legal and specifies the absence of a label.

Member Summary

Methods

`String`	`public java.lang.String getLabel ()`
`void`	`public void setLabel (java.lang.String label)`

Methods

getLabel()

`public java.lang.String getLabel ()`

Gets the label of this Item object.

Returns: the label string

setLabel(String)

`public void setLabel (java.lang.String label)`

Sets the label of the Item. If label is null, specifies that this item has no label.

Parameters:

`label` - the label string

javax.microedition.lcdui
ItemStateListener

Syntax

```
public interface ItemStateListener
```

Description

This interface is used by applications which need to receive events that indicate changes in the internal state of the interactive items within a `Form` screen.

See Also: `public void setItemStateListener (ItemStateListener iListener)`

Member Summary	
Methods	
`void`	`public void itemStateChanged (Item item)`

Methods

itemStateChanged(Item)

```
public void itemStateChanged (Item item)
```

Called when internal state of an Item has been changed by the user. This happens when the user:

- changes the set of selected values in a ChoiceGroup;
- adjusts the value of an interactive Gauge;
- enters or modifies the value in a TextField; and
- enters a new date or time in a DateField.

It is up to the device to decide when it considers a new value to have been entered into an Item. For example, implementations of text editing within a TextField vary greatly from device to device.

In general, it is not expected that the listener will be called after every change is made. However, if an item's value has been changed, the listener will be called to notify the application of the change before it is called for a change on another item, and before a command is delivered to the Form's CommandListener. For implementations that have the concept of an input focus, the listener should be called no later than when the focus moves away from an item whose state has been changed. The listener should be called only if the item's value has actually been changed.

The listener is not called if the application changes the value of an interactive item.

Parameters:
`item` - the item that was changed

javax.microedition.lcdui
List

Syntax

```
public class List extends Screen implements Choice
```

```
Displayable
  |
  +--Screen
        |
        +--javax.microedition.lcdui.List
```

All Implemented Interfaces: Choice

Description

The List class is a Screen containing list of choices. Most of the behavior is common with class `ChoiceGroup` and the common API is defined in interface `Choice` . When a List is present on the display the user can interact with it indefinitely (for instance, traversing from element to element and possibly scrolling). These traversing and scrolling operations do not cause application-visible events. The system notifies the application when some `Command` is fired. The notification of the application is done with `public void commandAction (Command c, Displayable d)` .

List, like any Choice, utilizes a dedicated "select" or "go" functionality of the devices. Typically, the select functionality is distinct from the soft-buttons, but some devices may use soft-buttons for the select. In any case, the application does not have a mean to set a label for a select key.

In respect to select functionality here are three types of Lists:

- IMPLICIT where select causes immediate notification of the application if there is an `CommandListener` registered. The element that has the focus will be selected before any CommandListener for this List is called. An implicit `public static final Command SELECT_COMMAND` is a parameter for the notification.
- EXCLUSIVE where select operation changes the selected element in the list. Application is not notified.
- MULTIPLE where select operation toggles the selected state of the focused Element. Application is not notified.

IMPLICIT List can be used to construct menus by placing logical commands to elements. In this case no application defined `Command` have to be attached. Application just has to register a CommandListener that is called when user "selects".

Another use might be implementation of a Screen with a default operation that takes place when "select" is pressed. For example, the List may contain email headers, and three operations: read, reply, and delete. Read is consider to be the default operation.

```
    void initialize() {
        myScreen = new List("EMAIL", List.IMPLICIT);
        readCommand = new Command("read", Command.SCREEN, 1);
        replyCommand = new Command("reply", Command.SCREEN, 1);
        deleteCommand = new Command("delete", Command.SCREEN, 1);
        myScreen.addCommand(readCommand);
        myScreen.addCommand(replyCommand);
        myScreen.addCommand(deleteCommand);
        myScreen.setCommandListener(this);
    }
```

Because the list is of type IMPLICIT, the select operation also calls the method `public void commandAction (Command c, Displayable d)` with parameter `public static final Command SELECT_COMMAND`. The implementation of commandAction() can now do the obvious thing and start the read operation:

```
    public void commandAction (Command c, Displayable d) {
        if (d == myScreen) {
            if (c == readCommand || c == List.SELECT_COMMAND) {
                // show the mail to the user
            }
            // ...
        }
    }
```

It should be noted that this kind of default operation must be used carefully and the usability of the resulting user interface must always kept in mind.

The application can also set the currently selected element(s) prior to displaying the List.

Note: Many of the essential methods have been documented in `Choice`.

Member Summary

Fields	
`Command`	`public static final Command SELECT_COMMAND`
Constructors	
	`public List (java.lang.String title, int listType)`
	`public List (java.lang.String title, int listType, java.lang.String[] stringElements, Image[] imageElements)`
Methods	
`int`	`public int append (java.lang.String stringPart, Image imagePart)`
`void`	`public void delete (int elementNum)`
`Image`	`public Image getImage (int elementNum)`
`int`	`public int getSelectedFlags (boolean[] selectedArray_return)`
`int`	`public int getSelectedIndex ()`
`String`	`public java.lang.String getString (int elementNum)`
`void`	`public void insert (int elementNum, java.lang.String stringPart, Image imagePart)`
`boolean`	`public boolean isSelected (int elementNum)`
`void`	`public void set (int elementNum, java.lang.String stringPart, Image imagePart)`
`void`	`public void setSelectedFlags (boolean[] selectedArray)`
`void`	`public void setSelectedIndex (int elementNum, boolean selected)`
`int`	`public int size ()`

Inherited Member Summary

Fields inherited from interface `Choice`

`public static final int EXCLUSIVE`, `public static final int IMPLICIT`, `public static final int MULTIPLE`

Methods inherited from class `Screen`

`public Ticker getTicker ()`, `public java.lang.String getTitle ()`, `public void setTicker (Ticker ticker)`, `public void setTitle (java.lang.String s)`

Methods inherited from class `Displayable`

`public void addCommand (Command cmd)`, `public boolean isShown ()`, `public void removeCommand (Command cmd)`, `public void setCommandListener (CommandListener l)`

Fields

SELECT_COMMAND

```
public static final Command SELECT_COMMAND
```

SELECT_COMMAND is a special command that `public void commandAction (Command c, Displayable d)` can use to recognize the user did the select operation on a IMPLICIT List. The field values of SELECT_COMMAND are:
`label = ""` (an empty string)

`type = SCREEN`

`priority = 0`

The application should not use these values for recognizing the SELECT_COMMAND. Instead, object identities of the Command and Displayable (List) should be used.

Constructors

List(String, int)

```
public List (java.lang.String title, int listType)
```

Creates a new, empty List, specifying its title and the type of the list.

Parameters:
`title` - the screen's title (see `Screen`)

`listType` - one of IMPLICIT, EXCLUSIVE, or MULTIPLE

Throws: `IllegalArgumentException` - if listType is not one of IMPLICIT, EXCLUSIVE, or MULTIPLE.

See Also: `Choice`

List(String, int, String[], Image[])

```
public List (java.lang.String title, int listType, java.lang.String[] stringElements,
            Image[] imageElements)
```

Creates a new List, specifying its title, the type of the List, and an array of Strings and Images to be used as its initial contents.

The stringElements array must be non-null and every array element must also be non-null. The length of the stringElements array determines the number of elements in the List. The imageElements array may be null to indicate that the List elements have no images. If the imageElements array is non-null, it must be the same length as the stringElements array. Individual elements of the imageElements array may be null in order to indicate the absence of an image for the corresponding List element. Any elements present in the imageElements array must refer to immutable images.

Parameters:

`title` - the screen's title (see `Screen`)

`listType` - one of IMPLICIT, EXCLUSIVE, or MULTIPLE

`stringElements` - set of strings specifying the string parts of the List elements

`imageElements` - set of images specifying the image parts of the List elements

Throws: `NullPointerException` - if stringElements is null

`NullPointerException` - if the stringElements array contains any null elements

`IllegalArgumentException` - if the imageElements array is non-null and has a different length from the stringElements array

`IllegalArgumentException` - if listType is not one of IMPLICIT, EXCLUSIVE, or MULTIPLE.

`IllegalArgumentException` - if any image in the imageElements array is mutable

See Also: `public static final int EXCLUSIVE`, `public static final int MULTIPLE`, `public static final int IMPLICIT`

Methods

---

append(String, Image)

```
public int append (java.lang.String stringPart, Image imagePart)
```

Specified By: `public int append (java.lang.String stringPart, Image imagePart)` in interface `Choice`

Parameters:

`stringPart` - the string part of the element to be added `imagePart` - the image part of the element to be added, or null if there is no image part

Returns: the assigned index of the element

Throws: `IllegalArgumentException` - if the image is mutable

`NullPointerException` - if stringPart is null

delete(int)

`public void delete (int elementNum)`

Specified By: `public void delete (int elementNum)` in interface `Choice`

Parameters:
`elementNum` - the index of the element to be deleted

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

getImage(int)

`public Image getImage (int elementNum)`

Specified By: `public Image getImage (int elementNum)` in interface `Choice`

Parameters:
`elementNum` - the number of the element to be queried

Returns: the image part of the element, or null if there is no image

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

See Also: `public java.lang.String getString (int elementNum)`, `public java.lang.String getString (int elementNum)`

getSelectedFlags(boolean[])

`public int getSelectedFlags (boolean[] selectedArray_return)`

Specified By: `public int getSelectedFlags (boolean[] selectedArray_return)` in interface `Choice`

Parameters:
`selectedArray_return` - array to contain the results

Returns: the number of selected elements in the Choice

Throws: `IllegalArgumentException` - if selectedArray_return is shorter than the size of the List

`NullPointerException` - if selectedArray_return is null

getSelectedIndex()

`public int getSelectedIndex ()`

Specified By: `public int getSelectedIndex ()` in interface `Choice`

Returns: index of selected element, or -1 if none

getString(int)

`public java.lang.String getString (int elementNum)`

Specified By: `public java.lang.String getString (int elementNum)` in interface `Choice`

Parameters:

`elementNum` - the index of the element to be queried

Returns: the string part of the element

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

See Also: `public Image getImage (int elementNum)`

insert(int, String, Image)

`public void insert (int elementNum, java.lang.String stringPart, Image imagePart)`

Specified By: `public void insert (int elementNum, java.lang.String stringPart, Image imagePart)` in interface `Choice`

Parameters:

`elementNum` - the index of the element where insertion is to occur

`stringPart` - the string part of the element to be inserted

`imagePart` - the image part of the element to be inserted, or null if there is no image part

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

`IllegalArgumentException` - if the image is mutable

`NullPointerException` - if stringPart is null

isSelected(int)

`public boolean isSelected (int elementNum)`

Specified By: `public boolean isSelected (int elementNum)` in interface `Choice`

Parameters:

`elementNum` - index to element to be queried

Returns: selection state of the element

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

set(int, String, Image)

`public void set (int elementNum, java.lang.String stringPart, Image imagePart)`

Specified By: `public void set (int elementNum, java.lang.String stringPart, Image imagePart)` in interface `Choice`

Parameters:

`elementNum` - the index of the element to be set

`stringPart` - the string part of the new element `imagePart` - the image part of the element, or null if there is no image part

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

`IllegalArgumentException` - if the image is mutable

`NullPointerException` - if stringPart is null

setSelectedFlags(boolean[])

```
public void setSelectedFlags (boolean[] selectedArray)
```

Specified By: `public void setSelectedFlags (boolean[] selectedArray)` in interface `Choice`

Parameters:
`selectedArray` - an array in which the method collect the selection status

Throws: `IllegalArgumentException` - if selectedArray is shorter than the size of the List

`NullPointerException` - if selectedArray is null

setSelectedIndex(int, boolean)

```
public void setSelectedIndex (int elementNum, boolean selected)
```

Specified By: `public void setSelectedIndex (int elementNum, boolean selected)` in interface `Choice`

Parameters:
`elementNum` - the index of the element, starting from zero

`selected` - the state of the element, where `true` means selected and `false` means not selected

Throws: `IndexOutOfBoundsException` - if elementNum is invalid

size()

```
public int size ()
```

Specified By: `public int size ()` in interface `Choice`

Returns: the number of elements in the List

javax.microedition.lcdui

Screen

Syntax

```
public abstract class Screen extends Displayable
```

```
Displayable
  |
  +--javax.microedition.lcdui.Screen
```

Direct Known Subclasses: `Alert`, `Form`, `List`, `TextBox`

Description

The common superclass of all high-level user interface classes. Adds optional title and ticker-tape output to the Displayable class. The contents displayed and their interaction with the user are defined by subclasses.

Using subclass-defined methods, the application may change the contents of a Screen object while it is shown to the user. If this occurs, and the Screen object is visible, the display will be updated automatically. That is, the implementation will refresh the display in a timely fashion without waiting for any further action by the application. For example, suppose a List object is currently displayed, and every element of the List is visible. If the application inserts a new element at the beginning of the List, it is displayed immediately, and the other elements will be rearranged appropriately. There is no need for the application to call another method to refresh the display.

It is recommended that applications change the contents of a Screen only while it is not visible (that is, while another Displayable is current). Changing the contents of a Screen while it is visible may result in performance problems on some devices, and it may also be confusing if the Screen's contents changes while the user is interacting with it.

Member Summary

Methods

`Ticker`	`public Ticker getTicker ()`
`String`	`public java.lang.String getTitle ()`
`void`	`public void setTicker (Ticker ticker)`
`void`	`public void setTitle (java.lang.String s)`

Inherited Member Summary

Methods inherited from class `Displayable`

`public void addCommand (Command cmd)`, `public boolean isShown ()`, `public void remove-Command (Command cmd)`, `public void setCommandListener (CommandListener l)`

Methods

getTicker()

`public Ticker getTicker ()`

Gets the ticker used by this Screen.

Returns: ticker object used, or null if no ticker is present

getTitle()

`public java.lang.String getTitle ()`

Gets the title of the Screen. Returns null if there is no title.

setTicker(Ticker)

`public void setTicker (Ticker ticker)`

Set a ticker for use with this Screen, replacing any previous ticker. If null, removes the ticker object from this screen. The same ticker is may be shared by several Screen objects within an application. This is done by calling setTicker() on different screens with the same Ticker object. If the Screen is physically visible, the visible effect should take place no later than immediately after the callback or `protected abstract void startApp ()` returns back to the implementation.

Parameters:
`ticker` - the ticker object used on this screen

setTitle(String)

`public void setTitle (java.lang.String s)`

Sets the title of the Screen. If null is given, removes the title.

If the Screen is physically visible, the visible effect should take place no later than immediately after the callback or `protected abstract void startApp ()` returns back to the implementation.

Parameters:
`s` - the new title, or null for no title

javax.microedition.lcdui
StringItem

Syntax

```
public class StringItem extends Item
```

```
Item
  |
  +--javax.microedition.lcdui.StringItem
```

Description

An item that can contain a string. A StringItem is display-only; the user cannot edit the contents. Both the label and the textual content of a StringItem may be modified by the application. The visual representation of the label may differ from that of the textual contents.

Member Summary		
Constructors		
		`public StringItem (java.lang.String label, java.lang.String text)`
Methods		
	`String`	`public java.lang.String getText ()`
	`void`	`public void setText (java.lang.String text)`

Inherited Member Summary
Methods inherited from class `Item`
`public java.lang.String getLabel ()`, `public void setLabel (java.lang.String label)`

Constructors

StringItem(String, String)

```
public StringItem (java.lang.String label, java.lang.String text)
```

Creates a new StringItem object with the given label and textual content. Either label or text may be present or null.

Parameters:

`label` - the Item label

`text` - the text contents

Methods

getText()

```
public java.lang.String getText ()
```

Gets the text contents of the StringItem, or null if the StringItem is empty.

Returns: a string with the content of the item

setText(String)

```
public void setText (java.lang.String text)
```

Sets the text contents of the StringItem. If text is null, the StringItem is set to be empty.

Parameters:
`text` - the new content

javax.microedition.lcdui
TextBox

Syntax

```
public class TextBox extends Screen
```

```
Displayable
  |
  +--Screen
        |
        +--javax.microedition.lcdui.TextBox
```

Description

The TextBox class is a Screen that allows the user to enter and edit text.

A TextBox has a maximum size, which is the maximum number of characters that can be stored in the object at any time (its capacity). This limit is enforced when the TextBox instance is constructed, when the user is editing text within the TextBox, as well as when the application program calls methods on the TextBox that modify its contents. The maximum size is the maximum stored capacity and is unrelated to the number of characters that may be displayed at any given time. The number of characters displayed and their arrangement into rows and columns are determined by the device.

The implementation may place a boundary on the maximum size, and the maximum size actually assigned may be smaller than the application had requested. The value actually assigned will be reflected in the value returned by `public int getMaxSize ()` . A defensively-written application should compare this value to the maximum size requested and be prepared to handle cases where they differ.

The text contained within a TextBox may be more than can be displayed at one time. If this is the case, the implementation will let the user scroll to view and edit any part of the text. This scrolling occurs transparently to the application.

TextBox has the concept of *input constraints* that is identical to TextField. The `constraints` parameters of methods within the TextBox class use constants defined in the `TextField` class. See the description of input constraints in the TextField class for the definition of these constants.

Member Summary

Constructors		
		`public TextBox (java.lang.String title, java.lang.String text, int maxSize, int constraints)`
Methods		
	`void`	`public void delete (int offset, int length)`
	`int`	`public int getCaretPosition ()`
	`int`	`public int getChars (char[] data)`
	`int`	`public int getConstraints ()`
	`int`	`public int getMaxSize ()`
	`String`	`public java.lang.String getString ()`
	`void`	`public void insert (char[] data, int offset, int length, int position)`
	`void`	`public void insert (java.lang.String src, int position)`
	`void`	`public void setChars (char[] data, int offset, int length)`
	`void`	`public void setConstraints (int constraints)`
	`int`	`public int setMaxSize (int maxSize)`

Member Summary	
`void`	`public void setString (java.lang.String text)`
`int`	`public int size ()`

Inherited Member Summary

Methods inherited from class `Screen`

`public Ticker getTicker ()`, `public java.lang.String getTitle ()`, `public void setTicker (Ticker ticker)`, `public void setTitle (java.lang.String s)`

Methods inherited from class `Displayable`

`public void addCommand (Command cmd)`, `public boolean isShown ()`, `public void removeCommand (Command cmd)`, `public void setCommandListener (CommandListener l)`

Constructors

TextBox(String, String, int, int)

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

Creates a new TextBox object with the given title string, initial contents, maximum size in characters, and constraints. If the text parameter is null, the TextBox is created empty. The maxSize parameter must be greater than zero.

Parameters:

`title` - the title text to be shown with the display

`text` - the initial contents of the text editing area, null may be used to indicate no initial content.

`maxSize` - the maximum capacity in characters. The implementation may limit boundary maximum capacity and the actually assigned capacity may me smaller than requested. A defensive application will test the actually given capacity with `public int getMaxSize ()` .

`constraints` - see input constraints

Throws: `IllegalArgumentException` - if maxSize is zero or less

`IllegalArgumentException` - if the constraints parameter is invalid

`IllegalArgumentException` - if text is illegal for the specified constraints

`IllegalArgumentException` - if the length of the string exceeds the requested maximum capacity or the maximum capacity actually assigned

Methods

delete(int, int)

`public void delete (int offset, int length)`

Deletes characters from the TextBox.

Parameters:
`offset` - the beginning of the region to be deleted

`length` - the number of characters to be deleted

Throws: `StringIndexOutOfBoundsException` - if offset and length do not specify a valid range within the contents of the TextBox

getCaretPosition()

`public int getCaretPosition ()`

Gets the current input position. For some UIs this may block some time and ask the user about the intended caret position, on some UIs may just return the caret position.

Returns: the current caret position, 0 if in the beginning.

getChars(char[])

`public int getChars (char[] data)`

Copies the contents of the TextBox into a character array starting at index zero. Array elements beyond the characters copied are left unchanged.

Parameters:
`data` - the character array to receive the value

Returns: the number of characters copied

Throws: `ArrayIndexOutOfBoundsException` - if the array is too short for the contents

`NullPointerException` - if `data` is `null`

getConstraints()

`public int getConstraints ()`

Get the current input constraints of the TextBox.

Returns: the current constraints value (see input constraints)

getMaxSize()

`public int getMaxSize ()`

Returns the maximum size (number of characters) that can be stored in this TextBox.

Returns: the maximum size in characters

getString()

`public java.lang.String getString ()`

Gets the contents of the TextBox as a string value.

Returns: the current contents

insert(char[], int, int, int)

`public void insert (char[] data, int offset, int length, int position)`

Inserts a subrange of an array of characters into the contents of the TextBox. The offset and length parameters indicate the subrange of the data array to be used for insertion. Behavior is otherwise identical to `public void insert (java.lang.String src, int position)` .

Parameters:

`data` - the source of the character data

`offset` - the beginning of the region of characters to copy

`length` - the number of characters to copy

`position` - the position at which insertion is to occur

Throws: `ArrayIndexOutOfBoundsException` - if offset and length do not specify a valid range within the data array

`IllegalArgumentException` - if the resulting contents are illegal for the current input constraints

`IllegalArgumentException` - if the insertion would exceed the current maximum capacity

`NullPointerException` - if `data` is `null`

insert(String, int)

`public void insert (java.lang.String src, int position)`

Inserts a string into the contents of the TextBox. The string is inserted just prior to the character indicated by the `position` parameter, where zero specifies the first character of the contents of the TextBox. If `position` is less than or equal to zero, the insertion occurs at the beginning of the contents, thus effecting a prepend operation. If `position` is greater than or equal to the current size of the contents, the insertion occurs immediately after the end of the contents, thus effecting an append operation. For example, `text.insert(s, text.size())` always appends the string s to the current contents.

The current size of the contents is increased by the number of inserted characters. The resulting string must fit within the current maximum capacity.

If the application needs to simulate typing of characters it can determining the location of the current insertion point ("caret") using the with `public int getCaretPosition ()` method. For example, `text.insert(s, text.getCaretPosition())` inserts the string s at the current caret position.

Parameters:

`src` - the String to be inserted

`position` - the position at which insertion is to occur

Throws: `IllegalArgumentException` - if the resulting contents are illegal for the current input constraints

`IllegalArgumentException` - if the insertion would exceed the current maximum capacity

`NullPointerException` - if `src` is `null`

---

setChars(char[], int, int)

```
public void setChars (char[] data, int offset, int length)
```

Sets the contents of the TextBox from a character array, replacing the previous contents. Characters are copied from the region of the `data` array starting at array index `offset` and running for `length` characters. If the data array is null, the TextBox is set to be empty and the other parameters are ignored.

Parameters:

`data` - the source of the character data

`offset` - the beginning of the region of characters to copy

`length` - the number of characters to copy

Throws: `ArrayIndexOutOfBoundsException` - if offset and length do not specify a valid range within the data array

`IllegalArgumentException` - if the text is illegal for the current input constraints

`IllegalArgumentException` - if the text would exceed the current maximum capacity

---

setConstraints(int)

```
public void setConstraints (int constraints)
```

Sets the input constraints of the TextBox. If the current contents of the TextBox do not match the new constraints, the contents are set to empty.

Parameters:

`constraints` - see input constraints

Throws: `IllegalArgumentException` - if the value of the constraints parameter is invalid

---

setMaxSize(int)

```
public int setMaxSize (int maxSize)
```

Sets the maximum size (number of characters) that can be contained in this TextBox. If the current contents of the TextBox are larger than maxSize, the contents are truncated to fit.

Parameters:

`maxSize` - the new maximum size

Returns: assigned maximum capacity - may be smaller than requested.

Throws: `IllegalArgumentException` - if maxSize is zero or less.

---

setString(String)

```
public void setString (java.lang.String text)
```

Sets the contents of the TextBox as a string value, replacing the previous contents.

Parameters:
`text` - the new value of the TextBox, or null if the TextBox is to be made empty

Throws: `IllegalArgumentException` - if the text is illegal for the current input constraints

`IllegalArgumentException` - if the text would exceed the current maximum capacity

---

size()

```
public int size ()
```

Gets the number of characters that are currently stored in this TextBox.

Returns: the number of characters

javax.microedition.lcdui
TextField

Syntax

```
public class TextField extends Item
```

```
Item
  |
  +--javax.microedition.lcdui.TextField
```

Description

A TextField is an editable text component that may be placed into a `Form` . It can be given a piece of text that is used as the initial value.

A TextField has a maximum size, which is the maximum number of characters that can be stored in the object at any time (its capacity). This limit is enforced when the TextField instance is constructed, when the user is editing text within the TextField, as well as when the application program calls methods on the TextField that modify its contents. The maximum size is the maximum stored capacity and is unrelated to the number of characters that may be displayed at any given time. The number of characters displayed and their arrangement into rows and columns are determined by the device.

The implementation may place a boundary on the maximum size, and the maximum size actually assigned may be smaller than the application had requested. The value actually assigned will be reflected in the value returned by `public int getMaxSize ()` . A defensively-written application should compare this value to the maximum size requested and be prepared to handle cases where they differ.

Input Constraints

The TextField shares the concept of *input constraints* with the `TextBox` object. The different constraints allow the application to request that the user's input be restricted in a variety of ways. The implementation is required to restrict the user's input as requested by the application. For example, if the application requests the NUMERIC constraint on a TextField, the implementation must allow only numeric characters to be entered.

The implementation is not required to do any syntactic validation of the contents of the text object. Applications must be prepared to perform such checking themselves.

The implementation may provide special formatting for the value entered. For example, a PHONENUMBER field may be separated and punctuated as appropriate for the phone number conventions in use, grouping the digits into country code, area code, prefix, etc. Any spaces or punctuation provided are not considered part of the text field's value. For example, a TextField with the PHONENUMBER constraint might display as follows:

```
    (408) 555-1212
```

but the value of the field visible to the application would be a string representing a legal phone number like "4085551212". Note that in some networks a '+' prefix is part of the number and returned as a part of the string.

Member Summary

Fields

	`int`	`public static final int ANY`
	`int`	`public static final int CONSTRAINT_MASK`
	`int`	`public static final int EMAILADDR`
	`int`	`public static final int NUMERIC`
	`int`	`public static final int PASSWORD`

Member Summary

int	public static final int PHONENUMBER
int	public static final int URL
Constructors	
	public TextField (java.lang.String label, java.lang.String text, int maxSize, int constraints)
Methods	
void	public void delete (int offset, int length)
int	public int getCaretPosition ()
int	public int getChars (char[] data)
int	public int getConstraints ()
int	public int getMaxSize ()
String	public java.lang.String getString ()
void	public void insert (char[] data, int offset, int length, int position)
void	public void insert (java.lang.String src, int position)
void	public void setChars (char[] data, int offset, int length)
void	public void setConstraints (int constraints)
int	public int setMaxSize (int maxSize)
void	public void setString (java.lang.String text)
int	public int size ()

Inherited Member Summary

Methods inherited from class Item

`public java.lang.String getLabel ()`, `public void setLabel (java.lang.String label)`

Fields

ANY

`public static final int ANY`

The user is allowed to enter any text.

Constant 0 is assigned to ANY.

CONSTRAINT_MASK

`public static final int CONSTRAINT_MASK`

The mask value for determining the constraint mode. The application should use the logical AND operation with a value returned by getConstraints() and CONSTRAINT_MASK in order to retrieve the current constraint mode, in order to remove any modifier flags such as the PASSWORD flag.

Constant 0xFFFF is assigned to CONSTRAINT_MASK.

EMAILADDR

```
public static final int EMAILADDR
```

The user is allowed to enter an e-mail address.

Constant 1 is assigned to EMAILADDDR.

NUMERIC

```
public static final int NUMERIC
```

The user is allowed to enter only an integer value. The implementation must restrict the contents to consist of an optional minus sign followed by an optional string of numerals.

Constant 2 is assigned to NUMERIC.

PASSWORD

```
public static final int PASSWORD
```

The text entered must be masked so that the characters typed are not visible. The actual contents of the text field are not affected, but each character is displayed using a mask character such as "*". The character chosen as the mask character is implementation-dependent. This is useful for entering confidential information such as passwords or PINs (personal identification numbers).

The PASSWORD modifier can be combined with other input constraints by using the logical OR operator (|). However, The PASSWORD modifier is nonsensical with some constraint values such as EMAILADDR, PHONENUMBER, and URL.

Constant 0x10000 is assigned to PASSWORD.

PHONENUMBER

```
public static final int PHONENUMBER
```

The user is allowed to enter a phone number. The phone number is a special case, since a phone-based implementation may be linked to the native phone dialing application. The implementation may automatically start a phone dialer application that is initialized so that pressing a single key would be enough to make a call. The call must not made automatically without requiring user's confirmation. The exact set of characters allowed is specific to the device and to the device's network and may include non-numeric characters.

Constant 3 is assigned to PHONENUMBER.

URL

```
public static final int URL
```

The user is allowed to enter a URL.

Constant 4 is assigned to URL.

Constructors

TextField(String, String, int, int)

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

Creates a new TextField object with the given label, initial contents, maximum size in characters, and constraints. If the text parameter is null, the TextField is created empty. The maxSize parameter must be greater than zero.

Parameters:

`label` - item label

`text` - the initial contents, or null if the TextField is to be empty

`maxSize` - the maximum capacity in characters

`constraints` - see input constraints

Throws: `IllegalArgumentException` - if maxSize is zero or less

`IllegalArgumentException` - if the value of the constraints parameter is invalid

`IllegalArgumentException` - if text is illegal for the specified constraints

`IllegalArgumentException` - if the length of the string exceeds the requested maximum capacity or the maximum capacity actually assigned

Methods

delete(int, int)

```
public void delete (int offset, int length)
```

Deletes characters from the TextField.

Parameters:

`offset` - the beginning of the region to be deleted

`length` - the number of characters to be deleted

Throws: `StringIndexOutOfBoundsException` - if offset and length do not specify a valid range within the contents of the TextField

getCaretPosition()

```
public int getCaretPosition ()
```

Gets the current input position. For some UIs this may block some time and ask the user about the intended caret position, on some UIs may just return the caret position.

Returns: the current caret position, 0 if in the beginning.

getChars(char[])

```
public int getChars (char[] data)
```

Copies the contents of the TextField into a character array starting at index zero. Array elements beyond the characters copied are left unchanged.

Parameters:
`data` - the character array to receive the value

Returns: the number of characters copied

Throws: `ArrayIndexOutOfBoundsException` - if the array is too short for the contents

`NullPointerException` - if `data` is `null`

getConstraints()

```
public int getConstraints ()
```

Get the current input constraints of the TextField.

Returns: the current constraints value (see input constraints)

getMaxSize()

```
public int getMaxSize ()
```

Returns the maximum size (number of characters) that can be stored in this TextField.

Returns: the maximum size in characters

getString()

```
public java.lang.String getString ()
```

Gets the contents of the TextField as a string value.

Returns: the current contents

insert(char[], int, int, int)

```
public void insert (char[] data, int offset, int length, int position)
```

Inserts a subrange of an array of characters into the contents of the TextField. The offset and length parameters indicate the subrange of the data array to be used for insertion. Behavior is otherwise identical to `public void insert (java.lang.String src, int position)`.

Parameters:
`data` - the source of the character data

`offset` - the beginning of the region of characters to copy

`length` - the number of characters to copy

`position` - the position at which insertion is to occur

Throws: `ArrayIndexOutOfBoundsException` - if offset and length do not specify a valid range within the data array

`IllegalArgumentException` - if the resulting contents are illegal for the current input constraints

`IllegalArgumentException` - if the insertion would exceed the current maximum capacity

`NullPointerException` - if `data` is `null`

---

insert(String, int)

```
public void insert (java.lang.String src, int position)
```

Inserts a string into the contents of the TextField. The string is inserted just prior to the character indicated by the `position` parameter, where zero specifies the first character of the contents of the TextField. If `position` is less than or equal to zero, the insertion occurs at the beginning of the contents, thus effecting a prepend operation. If `position` is greater than or equal to the current size of the contents, the insertion occurs immediately after the end of the contents, thus effecting an append operation. For example, `text.insert(s, text.size())` always appends the string s to the current contents.

The current size of the contents is increased by the number of inserted characters. The resulting string must fit within the current maximum capacity.

If the application needs to simulate typing of characters it can determining the location of the current insertion point ("caret") using the with `public int getCaretPosition ()` method. For example, `text.insert(s, text.getCaretPosition())` inserts the string s at the current caret position.

Parameters:
`src` - the String to be inserted

`position` - the position at which insertion is to occur

Throws: `IllegalArgumentException` - if the resulting contents are illegal for the current input constraints

`IllegalArgumentException` - if the insertion would exceed the current maximum capacity

`NullPointerException` - if `src` is `null`

---

setChars(char[], int, int)

```
public void setChars (char[] data, int offset, int length)
```

Sets the contents of the TextField from a character array, replacing the previous contents. Characters are copied from the region of the `data` array starting at array index `offset` and running for `length` characters. If the data array is null, the TextField is set to be empty and the other parameters are ignored.

Parameters:
`data` - the source of the character data

`offset` - the beginning of the region of characters to copy

`length` - the number of characters to copy

Throws: `ArrayIndexOutOfBoundsException` - if offset and length do not specify a valid range within the data array

`IllegalArgumentException` - if the text is illegal for the current input constraints

`IllegalArgumentException` - if the text would exceed the current maximum capacity

setConstraints(int)

```
public void setConstraints (int constraints)
```

Sets the input constraints of the TextField. If the the current contents of the TextField do not match the new constraints, the contents are set to empty.

Parameters:
`constraints` - see input constraints

Throws: `IllegalArgumentException` - if constraints is not any of the ones specified in input constraints

setMaxSize(int)

```
public int setMaxSize (int maxSize)
```

Sets the maximum size (number of characters) that can be contained in this TextField. If the current contents of the TextField are larger than maxSize, the contents are truncated to fit.

Parameters:
`maxSize` - the new maximum size

Returns: assigned maximum capacity - may be smaller than requested.

Throws: `IllegalArgumentException` - if maxSize is zero or less.

setString(String)

```
public void setString (java.lang.String text)
```

Sets the contents of the TextField as a string value, replacing the previous contents.

Parameters:
`text` - the new value of the TextField, or null if the TextField is to be made empty

Throws: `IllegalArgumentException` - if the text is illegal for the current input constraints

`IllegalArgumentException` - if the text would exceed the current maximum capacity

size()

```
public int size ()
```

Gets the number of characters that are currently stored in this TextField.

Returns: number of characters in the TextField

javax.microedition.lcdui

Ticker

Syntax

`public class Ticker`

`javax.microedition.lcdui.Ticker`

Description

Implements a "ticker-tape," a piece of text that runs continuously across the display. The direction and speed of scrolling are determined by the implementation. While animating, the ticker string scrolls continuously. That is, when the string finishes scrolling off the display, the ticker starts over at the beginning of the string.

There is no API provided for starting and stopping the ticker. The application model is that the ticker is always scrolling continuously. However, the implementation is allowed to pause the scrolling for power consumption purposes, for example, if the user doesn't interact with the device for a certain period of time. The implementation should resume scrolling the ticker when the user interacts with the device again.

The same ticker may be shared by several Screen objects. This can be accomplished by calling `public void setTicker (Ticker ticker)` on all such screens. Typical usage is for an application to place the same ticker on all of its screens. When the application switches between two screens that have the same ticker, a desirable effect is for the ticker to be displayed at the same location on the display and to continue scrolling its contents at the same position. This gives the illusion of the ticker being attached to the display instead of to each screen.

An alternative usage model is for the application to use different tickers on different sets of screens or even a different one on each screen. The ticker is an attribute of the Screen class so that applications may implement this model without having to update the ticker to be displayed as the user switches among screens.

Member Summary		
Constructors		
		`public Ticker (java.lang.String str)`
Methods		
	`String`	`public java.lang.String getString ()`
	`void`	`public void setString (java.lang.String str)`

Constructors

Ticker(String)

`public Ticker (java.lang.String str)`

Constructs a new Ticker object, given its initial contents string.

Parameters:
`str` - string to be set for the Ticker

Throws: `NullPointerException` - if str is null

Methods

getString()

```
public java.lang.String getString ()
```

Gets the string currently being scrolled by the ticker.

Returns: string of the ticker

setString(String)

```
public void setString (java.lang.String str)
```

Sets the string to be displayed by this ticker. If this ticker is active and is on the display, it immediately begins showing the new string.

Parameters:
`str` - string to be set for the Ticker

Throws: `NullPointerException` - if str is null

Index