Information Module Profile Next Generation

for Java™ 2 Micro Edition

Final Release

JSR 228 Expert Group
jsr-228-comments@jcp.org

Java Community Process

Information Module Profile – Next Generation (IMP-NG) Specification	**("Specification")**
Version:	**1.0.0**
Release:	**October 26st, 2005**
Status:	**FCS**
Specification Leads:	**Siemens AG and Nokia Corporation ("Specification Leads")**

Contents

Contents

Preface

This document defines the *Information Module Profile – Next Generation (IMP-NG) v2.0 Specification* for the Java 2 Platform, Micro Edition (J2ME^{TM}).

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.

Revision History

Date	**Version**	**Description**
10-July-2003	IMP Specification	Final IMP (JSR-195) specification
22-October-2003	IMP-NG, SpecLead Draft 1	First draft provided by the Specification Leads
24-October-2003	IMP-NG, After-Kickoff Draft	Draft as result of Kickoff meeting in Munich at October 23rd/24th, 2003
20-July-2004	IMP-NG, EDR Draft 1	Draft for Early Draft Review, version 1
15-October-2004	IMP-NG, EDR Draft 2	Draft for Early Draft Review, version 2
20-December-2004	IMP-NG, PR Draft	Draft for Public Review
26-October-2005	IMP-NG, FR	Final Release

Who Should Use This Specification

This document is targeted at the following audiences:

- The Java Community Process (JCP) expert group defining this profile
- Implementers of the IMP-NG
- Application developers targeting the IMP-NG
- Network operators deploying infrastructure to support IMP-NG enabled devices

How This Specification Is Organized

This specification consists on the following two parts:

- JavaDoc API Documentation
- A textual specification

There are requirements listed both in this document and the API documentation. Where there are conflicts, the requirements listed in this document override the API documentation.

Related Literature

- *The Java Language Specification, Second Edition* by James Gosling, Bill Joy, and Guy L. Steele. Addison-Wesley, June 2000, ISBN 0-201-31008-2
- *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 (http://jcp.org/jsr/detail/30.jsp).
- Mobile, Information Device Profile (JSR-37), Sun Microsystems, Inc (http://jcp.org/jsr/detail/37.jsp).
- Connected, Limited Device Configuration 1.1 (JSR-139), Sun Microsystems, Inc (http://jcp.org/jsr/detail/139.jsp).
- Mobile Information Device Profile, version 2.0 (JSR-118), Sun Microsystems, Inc and Motorola, Inc (http://jcp.org/jsr/detail/118.jsp).
- Information Module Profile (JSR-195), Siemens AG and Nokia Mobile Phones (http://jcp.org/jsr/detail/195.jsp).
- Scalable Polyphony MIDI Specification Version 1.0, MIDI Manufacturers Association, Los Angeles, CA, USA, February 2002.
- Scalable Polyphony MIDI Device 5-to-24 Note Profile for 3GPP Version 1.0, RP-35, MIDI Manufacturers Association, Los Angeles, CA, USA, February 2002.
- PKCS #1 RSA Encryption Version 2.0 (http://www.ietf.org/rfc/rfc2437)
- Internet X.509 Public Key Infrastructure (http://www.ietf.org/rfc/rfc2459)
- Online Certificate Status Protocol (http://www.ietf.org/rfc/rfc2560)
- WAP Wireless Identity Module Specification (WIM) WAP-260-WIM-20010712-a (http://www.wapforum.org/what/technical.htm)
- WAP Smart Card Provisioning (SCPROV) WAP-186-ProvSC-20010710-a (http://www.wapforum.org/what/technical.htm)
- HTTP 1.1 Specification (http://www.ietf.org/rfc/rfc2616.txt)
- PKCS#15 v.1.1 (http://www.rsasecurity.com/rsalabs/pkcs/pkcs-15/)
- USIM, 3GPP TS 31.102: “Characteristics of the USIM applications” (http://www.3gpp.org/)
- RFC3280 (http://www.ietf.org/rfc)

Report and Contact

Your comments on this specification are welcome and appreciated. Please send your comments to: *jsr-228-comments@jcp.org*

Maintenance Note

Siemens Mobile and Nokia, who are joint specification leads for JSR-228 (Information Module Profile – Next Generation) have granted Sun Microsystems Inc. the rights to implementation and license the reference implementation (RI) and Technology Compatibility Kit (TCK) for JSR-228.
Further, since IMP-NG (JSR-228) is a strict subset of MIDP 2.0 (JSR-118), and in an effort to minimize the risk of fragmentation due to interpretations and TCK exclusions, the Maintenance Leads of JSR-228 and JSR-118 have agreed that:

1) all TCK exclusions for both JSRs will go through the JSR-118 process and be handled by the JSR-118 Maintenance Lead, and
2) all spec interpretation issues will be coordinated between the two parties to ensure agreement and synchronization.

1 Introduction

1.1 Introduction

This document, produced as a result of Java Specification Request (JSR) 228, defines the Information Module Profile – Next Generation (IMP-NG) for the Java 2 Platform, Micro Edition (J2ME™). The goal of this specification is to define an enhanced architecture and the associated APIs required enabling an open, third party, application development environment for information modules, or IMs.

The IMP-NG specification is based on the IMP specification (see JSR-195) and provides backward compatibility with IMP so that IMlets written for IMP can execute in IMP-NG environments. IMP-NG is a strict subset of the Mobile Information Device Profile (MIDP), version 2.0, which is described in *Mobile Information Device Profile, version 2.0 (JSR-118)*, Motorola / Sun Microsystems, Inc.

The IMP-NG is designed to operate on top of the Connected, Limited Device Configuration (CLDC) which is described in *Connected, Limited Device Configuration (JSR-30)* (http://jcp.org/jsr/detail/30.jsp), Sun Microsystems, Inc. While the IMP-NG specification was designed assuming only CLDC 1.0 features, it will also work on top of *CLDC 1.1 (JSR-139)* (http://jcp.org/jsr/detail/139.jsp), and presumably any newer versions. It is anticipated, though, that most IMP-NG implementations will be based on CLDC 1.0.

1.1 Contributors

The Information Module Profile Next Generation Expert Group (IMPNGEG), as a part of the Java Community Process, produced this specification. The following companies and individuals, listed in alphabetical order, were members of the Expert Group:

- Companies:
  - esmertec
  - IOPSIS
  - Motorola Inc.
  - Nokia Mobile Phones[1]
  - Siemens AG[1]
  - Sun Microsystems, Inc.
- Individuals:
  - Peter Kriens
  - C. Enrique Ortiz

[1] Overall Specification Lead

1.2 Document Conventions

1.2.1 Definitions

This document uses definitions based upon those specified in RFC 2119 (http://www.ietf.org/rfc/rfc2119.txt).

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.

1.2.2 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.

2 Requirements and Scope

2.1 Device Requirements

The requirements listed in this chapter are additional requirements above those found in *Connected, Limited Device Configuration, version 1.0 (JSR-30) and version 1.1 (JSR-139)*, Sun Microsystems, Inc.

At a high level, the IMP-NG specification assumes that the IM is limited in its processing power, memory, connectivity.

2.1.1 Hardware

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

- Memory:
    - 128 kilobytes of non-volatile[2] memory for the IMP components
    - 8 kilobytes of non-volatile memory for application-created persistent data
    - 128 kilobytes of volatile memory[3] for the Java runtime (e.g. the Java heap)
- Networking:
    - Two-way, wireless, possibly intermittent, with limited bandwidth
- Display:
    - UI capabilities, if any, that do not allow the use of MIDP UI
    - If the device has UI capabilities that can be addressed by MIDP UI, the IMP MUST NOT be used. Such devices must use MIDP 1.0 or MIDP 2.0 or one of its successors instead.
- Sound:
    - The optional ability to play tones, either via dedicated hardware, or via software algorithm.

Examples of IMs include, but are not restricted to, network cards, routers, tracking devices, and vending machines.

1 Memory requirements cited here are for IMP components only. CLDC and other system software memory requirements are beyond the scope of this specification and therefore not included.

[2] *Non-volatile* means that the memory is expected to retain its contents between the users turning the devices “off” and “on”. For the purpose 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 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.

[3] *Volatile* means that the memory 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.

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 IM space is characterized by a wide variety of system software. For example, some IMs may have a full-featured operating system that supports multi-processing and hierarchical filesystems, while other IMs may have small, thread-based operating systems with no notion of a filesystem. Faced with such variety, the IMP-NG makes minimal assumptions about the IM'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 requirements of the Record Management System (RMS) APIs for persistent storage.
- Read and write access to the device's wireless networking to support the Networking APIs.
- A mechanism to provide a time base for use in time-stamping the records written to Persistent Storage and to provide the basis for the Timer APIs.
- A mechanism for managing the application life-cycle of the device.

2.2 Scope

Information Modules (IMs) span a potentially wide set of capabilities. Following the intentions of the MIDP 1.0 (JSR-037) and MIDP 2.0 (JSR-118) expert groups, respectively, the IMP (JSR-195) and IMP-NG (JSR-228) expert groups agreed to limit the set of APIs specified, rather than try to address all such capabilities. So the IMP and IMP-NG profiles address only those functional areas that were considered absolute requirements to achieve broad portability and successful deployments. These include:

- Application delivery and billing
- Application lifecycle (i.e., defining the semantics of a IMP/IMP-NG application and how it is controlled)
- Application signing model and privileged domains security model
- End-to-end transactional security (https)
- IMlet push registration (server push model)
- Networking
- Persistent storage
- Sound (optional)
- Timers

The above features are discussed in more depth in the associated Javadoc.

By the same reasoning, some areas of functionality were considered to be outside the scope of the IMP and IMP-NG. These areas include:

- System-level APIs: The emphasis on the IMP and IMP-NG APIs is, again, on enabling application programmers, rather than enabling system programming. Thus, low-level APIs that specify a system interface to, for example, an IM's power management are beyond the scope of this specification.
- Low-level security: The IMP and IMP-NG specifies no additional low-level security features other than those provided by the CLDC.

2.3 IMP-NG and Optional Packages

IMs can have different kind of capabilities that are difficult to capture in one specification. Also, as IMP-NG is a strict subset of MIDP 2.0, it cannot include new APIs. Support for such device capabilities can be added to IMP-NG by optional packages, as defined in *J2ME Platform Specification (JSR-68),* Sun Microsystems, Inc.

Following the architecture proposed by the J2ME Platform Specification, such optional packages could also be used in devices that do not use IMP-NG, but rather a different profile, like IMP or even MIDP 1.0 or MIDP 2.0.

3 Architecture

3.1 General Architecture

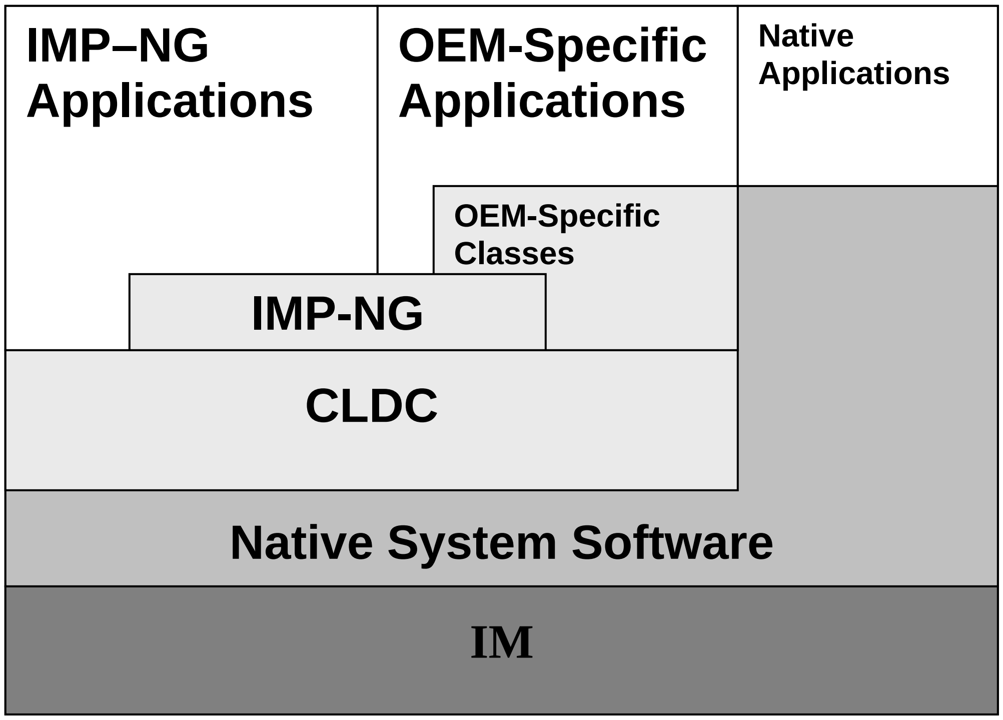
This section addresses issues that both implementers and developers will encounter when implementing and developing IMP-NG. For IMP-NG is a strict subset of MIDP 2.0, most ideas of the IMP-NG architecture originate from the MIDP 2.0 specification (http://jcp.org/jsr/detail/118.jsp).

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

In the High-Level Architecture, the lowest-level block (IM) represents the Information Module 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 Virtual Machine 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:

- IMP-NG APIs: The set of APIs defined in this specification.
- OEM-specific APIs: Given the broad diversity of devices in the IMP-NG space, it is not possible to fully address all device 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 IMs.
  Please note that frequently used APIs are candidates for optional packages (see 2.3. IMP-NG and Optional Packages).

Note that in the figure, the CLDC is shown as the basis of the IMP-NG and device-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 an IM are actually part of the virtual machine, which maps the Java-level APIs to the underlying native implementation.

The top-most blocks in the figure above represent the application types possible on an IM. A short description of each application type is shown in the table below.

TABLE 3-1: IM Application Types

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

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

3.2 IMlets

Applications that use only the APIs defined by the IMP(-NG) and CLDC specifications are called IMlets. Because IMlets have a similar life cycle as MIDlets, IMlets are implemented by `javax.microedition.midlet.MIDlet` class, as defined in MIDP 2.0 specification.

Technically IMlets do not differ from MIDP 2.0 MIDlets, except by the fact that IMlets can not refer to MIDP classes that are not part of IMP(-NG). Besides, the value of the `microedition.profiles` attribute in JAD files and manifests (see below) has a different value, naturally. Nevertheless, IMPNGEG decided to use the term *IMlet*[1] for IM applications, to emphasize the differences of IMP(-NG) applications from MIDP applications.

[1] The word IMlet should be pronounced as one word with "im" like in "image".

3.3 Specification Requirements

This section lists some explicit requirements of this specification. Other requirements can be found in the associated Javadoc. If any requirements listed here differ from requirements listed elsewhere in the specification, the requirements here take precedence and replace the conflicting requirements.

Compliant IMP-NG implementations:

- MUST support IMP and IMP-NG IMlets and IMlet Suites.
- MUST include all non-optional packages, classes, and interfaces described in this specification.
- MUST implement the OTA User Initiated Provisioning specification.
- MAY incorporate zero or more supported protocols for push.
- MAY provide support for accessing any available serial ports on their devices through the CommConnection interface.
- MUST provide support for accessing HTTP 1.1 servers and services either directly, or by using gateway services.
- MUST provide support for secure HTTP connections either directly, or by using gateway services.
- SHOULD provide support for datagram connections.
- SHOULD provide support for server socket stream connections.
- SHOULD provide support for socket stream connections.
- SHOULD provide support for secure socket stream connections.
- MAY support Tone Generation in the media package.
- MUST support 8-bit, 8 KHz, mono linear PCM wav format IF any sampled sound support is provided.
- MAY include support for additional sampled sound formats.
- MUST support Scalable Polyphony MIDI (SP-MIDI) and SP-MIDI Device 5-to-24 Note Profile IF any synthetic sound support is provided.
- MAY include support for additional MIDI formats.
- MUST implement the mechanisms needed to support “Untrusted IMlet Suites”.
- MUST implement “Trusted IMlet Suite Security” unless the device security policy does not permit or support trusted applications.
- MUST implement “Trusted IMlet Suites Using X.509 PKI” to recognize signed IMlet suites as trusted unless PKI is not used by the device for signing applications.
- MUST implement “MIDP x.509 Certificate Profile” for certificate handling of HTTPS and SecureConnections.
- MUST enforce the same security requirements for I/O access from the Media API as from the Generic Connection framework, as specified in the package documentation for `javax.microedition.io`.
- MUST support at least the UTF-8 (http://ietf.org/rfc/rfc2279.txt) character encoding for APIs that allow the application to define character encodings.
- MAY support other character encodings.
- SHOULD NOT allow copies to be made of any IMlet suite unless the device implements a copy protection mechanism.

3.4 Package Summary

Application Lifecycle Package	
`javax.microedition.midlet`	The MIDlet package defines IMP-NG applications and the interactions between the application and the environment in which the application runs.
Persistence Package	
`javax.microedition.rms`	The IMP-NG provides a mechanism for IMlets to persistently store data and later retrieve it.
Networking Package	
`javax.microedition.io`	IMP-NG includes networking support based on the Generic Connection framework from the *Connected, Limited Device Configuration*.
Public Key Package	
`javax.microedition.pki`	Certificates are used to authenticate information for secure Connections.
Sound and Tone Media	
`javax.microedition.media`	The IMP-NG optional Media API is a directly compatible building block of the Mobile Media API (JSR-135) specification.
`javax.microedition.media.control`	This optional package defines the specific `Control` types that can be used with a `Player`.
Core Packages	
`java.lang`	MID Profile Language Classes included from Java 2 Standard Edition.
`java.util`	MID Profile Utility Classes included from Java 2 Standard Edition.

4 Over The Air User Initiated Provisioning Specification

4.1 Changes since the OTA Recommended Practice

The specification of MIDP 1.0 did not specify how MIDP applications – MIDlets – should be downloaded. After the MIDP 1.0 specification was published though, a document entitled, *Over The Air User Initiated Provisioning Recommended Practice for the Mobile Information Device Profile* was published. But it was never part of the MIDP specification, so if the IMP specification (JSR-195) was created as a pure subset of MIDP 1.0, there was no question about integrating the "recommended practice" into IMP specification.

In the meantime, the situation is different. MIDP 2.0 has been specified (JSR-118), and it contains a chapter about OTA User initiated provisioning. Also, for information modules in the meantime the requirement emerged to download applications.

MIDP 2.0 OTA provisioning is strongly dependent on UI based user interaction, while IMP-NG does not contain any UI at all. So, while this specification is still to be interpreted as a strict subset of MIDP 2.0, necessary changes have to be done to the OTA specification part to enable OTA support without user interaction once the trigger initiating the OTA has been received.

In the MIDP 2.0 specification, the following changes have been realized regarding the recommended practice for MIDP 1.0:

- Removed the Cookie support requirement. This was necessary because in some network architectures the cookie information may not be transmitted to the client. The cookies were used to maintain state information between the Application Descriptor, JAR downloads, and Install-Notify reports. An alternative approach of URL rewriting is possible, and can serve the same purpose. For example, when sending the Application Descriptor, the server can insert unique JAR, MIDlet-Install-Notify, and MIDlet-Delete-Notify URLs that associate these with this a particular download session. Other options may also be possible.

4.2 Over The Air User Initiated Provisioning

4.2.1 Overview and Goals

The purpose of this chapter is to describe how IMlet suites can be deployed Over-The-Air (OTA), and the requirements imposed upon the client device to support these deployments. Following these recommendations will help ensure interoperability between clients and servers from all manufacturers and provide guidance to mobile network operators deploying IMP-NG devices.

MIDP 2.0 requires MIDP enabled devices to provide mechanisms that allow users to discover MIDlet suites that can be loaded into the device. But all these mechanisms require user interaction, and it is therefore not possible at all to make it a requirement for IMs.

On the other hand, other installation mechanisms (e.g. Bluetooth™ wireless technology, serial cable, IrDA™, etc.) are mentioned to be possibly supported by devices, but stated as outside the scope of the MIDP 2.0 specification. For this specification, handling IMP-NG, we will speak of all these as "external triggers" to initiate OTA User Provisioning. Also "internal triggers" such as Timer events or requests from running Java applications are solutions one can think of. Which trigger is used by an implementation is outside the scope of this specification. The goal of such a generic solution is to ensure that an IMlet can be downloaded by implementations of several manufacturers – not depending on what kind of trigger is used.

For every kind of external trigger security is an issue to keep in mind. Depending on security requirements it is strongly recommended to use security mechanisms such as passwords to avoid unauthorized triggering by external sources.

The term Application Management Software (AMS) is a generic term used to describe the software on the device that manages the downloading and lifecycle of IMlets. This term does not refer to any specific implementation and is used for convenience only. In some implementations, the term Java Application Manager (JAM) is used interchangeably.

This chapter describes the general functional requirements on the device and the functions supporting the IMlet suite lifecycle. The lifecycle of an IMlet suite consists of installation, update, invocation and removal. Descriptions are included for additional Application Descriptor attributes and mechanisms that identify the device type and characteristics to servers providing IMlet suites.

4.2.2 Functional Requirements

An IMP-NG compliant device MUST be capable of:

- Transferring an IMlet suite and its associated Application Descriptor to the device from a server using HTTP 1.1 or a session protocol that implements the HTTP 1.1 functionality (including the header and entity fields) as required in this document.
- Responding to a 401 (*Unauthorized*) or 407 (*Proxy Authentication Required*) response to an HTTP request. The device MUST be able to support at least the RFC2617 Basic Authentication Scheme.
- Installing the IMlet suite on the device
- Invoking IMlets
- Allowing the user to delete IMlet suites stored on the device. Single IMlets cannot be deleted since the IMlet suite is the unit of transfer and installation.

4.2.3 IMlet Suite Installation

Application installation is the process by which an IMlet suite is downloaded onto the device and made available. Application installation MUST be supported. The network supporting the devices, as well any proxies and origin servers that are used during provisioning MUST be able to support this requirement. The user retains control of the resources used by IMlet suites on the device and MUST be allowed to delete or install IMlet suites.

The device MUST make the IMlet(s) in the IMlet suite available for execution. The device MAY run an IMlet from the IMlet suite immediately at the user’s option, which has to be part of the trigger then.

If the IMlet suite is already installed on the device, it SHOULD be treated as an update. See IMlet Suite Update for additional information on how to handle an update.

To install an IMlet suite, the AMS performs the following series of steps and checks:

1. An external or internal trigger (see above), sent towards the device, starts the download process.
2. The device initiates the download of the IMlet suite via HTTP. After that there are two ways:
   a) If the value of the *Content-Type* header is *text/vnd.sun.j2me.app-descriptor*, the Application Descriptor is downloaded first, and the request for the IMlet suite MUST be for exactly the URL specified in the descriptor; additional headers are unnecessary.
   b) If the value of the *Content-Type* header is *application/java-archive*, then the IMlet suite as a jar file is downloaded immediately. Note that the downloaded application can only be installed as untrusted in this case.

3. If credencials of an Authorization or Proxy-Authorization header field are required, these must be provided to the IM already as part of the trigger. Otherwise the download fails. Contrary to MIDs, IMs have no possibility to require the credentials from the user while the download is in progress. The download software has to resend the request with the user-supplied credentials autonomically instead. The device MUST be able to support at least the Basic Authentication Scheme as described in RFC2617.
4. The IMlet suite and the headers that are received MUST be checked to verify that the retrieved IMlet suite is valid and can be installed on the device. At least the following problems that prevent installation have to occur in a status report:
   - If there is insufficient memory to store the IMlet suite on the device, the device MUST return *Status Code 901* in the Status Report. The decision about whether there is sufficient memory to store an IMlet suite is done by comparison of the available memory and the size of the IMlet suite as provided in the *MIDlet-Jar-Size* attribute.
   - If the JAR is not available at the *MIDlet-Jar-URL* attribute in the descriptor, the device MUST return *Status Code 907* in the Status Report.
   - If the received JAR file size does not match the size specified in the Application Descriptor, the device MUST return *Status Code 904* in the Status Report.
   - If the manifest or any other file cannot be extracted from the JAR, the device MUST return *Status Code 907* in the Status Report.
   - If the JAR manifest is not in the correct syntax, or if any of the required attributes are missing in the JAR manifest, the device MUST return *Status Code 907* in the Status Report.
   - If the mandatory attributes in the descriptor "*MIDlet-Name*", "*MIDlet-Version*", and "*MIDlet-Vendor*" do not match those in the JAR manifest, the device MUST return *Status Code 905* in the Status Report.
   - If the IMlet suite is trusted, then the values in the application descriptor for MIDlet-* attributes MUST be identical to the corresponding attribute values in the Manifest. If not, the device MUST return *Status Code 905* in the Status Report.
   - If the application failed to be authenticated, the device MUST return *Status Code 909* in the Status Report.
   - If the application is an unsigned version of an installed signed version of the same application, the device MUST return *Status Code 910* in the Status Report.
   - If the application is not authorized for a permission listed in the *MIDlet-Permissions* attribute, the device MUST return *Status Code 910* in the Status Report.
   - If a static push registration fails for a reason other than not being authorized, the device MUST return *Status Code 911* in the Status Report.
   - If the network service is lost during installation, *Status Code 903* SHOULD be used in a Status Report if possible (it may be impossible to deliver the status report due to the network-service outage).
5. Provided there are no problems that prevent installation, the IMlets contained in the IMlet suite MUST be installed and made available for execution by the user.
6. Installation is complete when the IMlet suite has been made available on the device, or an unrecoverable failure has occurred. In either case, the status MUST be reported as described in Installation Status Reports.

4.2.4 IMlet Suite Update

An IMlet suite update is defined as the operation of installing a specific IMlet suite when that same IMlet suite (either the same version or a different version) is already installed on the device. Devices MUST support the updating of IMlet suites. See Device Identification and Request Headers for the attributes that apply to updates.

The RMS record stores of an IMlet suite being updated MUST be managed as follows:

- If the cryptographic signer of the new IMlet suite and the original IMlet suite are identical, then the RMS record stores MUST be retained and made available to the new IMlet suite.
- If the scheme, host, and path of the URL that the new Application Descriptor is downloaded from is identical to the scheme, host, and path of the URL the original Application Descriptor was downloaded from, then the RMS MUST be retained and made available to the new IMlet suite.
- If the scheme, host, and path of the URL that the new IMlet suite is downloaded from is identical to the scheme, host, and path of the URL the original IMlet suite was downloaded from, then the RMS MUST be retained and made available to the new IMlet suite.
- If the above statements are false, then the implementor should be aware, that a possible security risk exists and the existing RMS Record Store could be misused. It is therefore recommended, at least for non-trusted applications, to remove the old Record Store in this case and create a new one for the new application version.

In all cases, an unsigned IMlet suite MUST NOT be allowed to update a signed IMlet suite. The format, contents and versioning of the record stores is the responsibility of the IMlet suite. See below for details.

4.2.5 IMlet Suite Execution

When the user triggers an IMlet to be run, the device MUST invoke the IMlet with the CLDC and IMP-NG classes required by the IMP-NG specification.

4.2.6 IMlet Suite Removal

Devices MUST allow users to remove IMlet suites by triggering as described above.

If the Application Descriptor includes the attribute *MIDlet-Delete-Confirm*, its value SHOULD be included in the delete notification.

4.2.7 Installation/Deletion Status Reports

The success or failure of the installation, upgrade, or deletion of an IMlet suite is of interest to the service providing the IMlet suite, which is usually the originator of the initiating trigger. The service MAY specify URLs in the Application Descriptor that MUST be used to report installation and deletion status. See Additional Descriptor Attributes for more information. If the device cannot send the installation status report, the requested action MUST still be completed. For example, if the device cannot send the installation status report to the *MIDlet-Install-Notify* URL, the IMlet suite MUST still be enabled, and the user MUST be allowed to use it. Likewise if the device cannot send the deletion status report to the *MIDlet-Delete-Notify* URL, the IMlet suite MUST still be deleted.

The operation status is reported by means of an HTTP POST to the URL specified in the *MIDlet-Install-Notify* attribute for installations, or the *MIDlet-Delete-Notify* attribute for deletions. The only protocol that MUST be supported is "*http://*". The device MAY ignore other protocols.

The content of the body of the POST request MUST include a status code and status message on the first line. See the table below for a list of valid codes and status messages.

In the case of a deletion status report, the notification is sent only when the IMlet is deleted; *Status Code 912* MUST be sent, notifying that the deletion occurred.

In response to a status report, the server MUST reply with a "*200 OK*" response. No content SHOULD be returned to the device and, if any is sent, it MUST be ignored. If a response is received the request SHOULD NOT be retried. Contrary to the MIDP 1.0 OTA Recommended Practice, the server MUST NOT include a *Set-Cookie* header with the attribute *Max-Age=0* to request that the *cookie* be discarded. If such an attribute is received, the device MUST ignore it. As an example, please see Example: Install Status via HTTP Post Request.

For installations, if the status report cannot be sent, or if the server reply is not received, the installation status report MAY be sent again (as described above) each time an IMlet in this suite is executed and the device has data network connectivity. This will improve the likelihood of the status report being successfully sent. The number of retries attempted SHOULD be kept small since each one may result in a charge to the user's bill. The IMlet suite MUST be made available for use, whether or not the installation status report has been successfully sent and the acknowledgement have been received.

For deletions, an attempt to send the status report MUST be made the next time either an OTA installation is performed or an installation status report is being sent. This will improve the likelihood of the status report being successfully sent. If the status report cannot be sent, or if the server reply is not received, the deletion status report MAY be sent again (as described above) each time an OTA installation is performed or an installation status report is being sent. The number of retries attempted SHOULD be kept small since each one may result in a charge to the user's bill. The IMlet suite MUST be removed from memory, whether or not the installation status report has been successfully sent and the acknowledgement have been received.

TABLE 4-1: Install Status Codes and Message

Status Code	Status Message
900	Success
901	Insufficient Memory
902	User Cancelled
903	Loss of Service
904	JAR size mismatch
905	Attribute Mismatch
906	Invalid Descriptor
907	Invalid JAR
908	Incompatible Configuration or Profile
909	Application authentication failure
910	Application authorization failure
911	Push registration failure
912	Deletion Notification

4.2.8 Additional Descriptor Attributes

The following additional attributes are defined in the Application Descriptor. Each may appear only once in the descriptor.

TABLE 4-2: IMlet Attributes

Attribute Name	Attribute Description
MIDlet-Install-Notify	The URL to which a POST request is sent to report the installation status (whether a new installation or IMlet suite update) of this IMlet suite. The device MUST use this URL unmodified. The URL MUST be no longer than 256 UTF-8 encoded characters. If the device receives a URL longer than 256 UTF-8 encoded characters it MUST reject the installation and return *Status Code 906* in the status report.
MIDlet-Delete-Notify	The URL to which a POST request is sent to report the deletion of this IMlet suite. The device MUST use this URL unmodified. The URL MUST be no longer than 256 UTF-8 encoded characters. If the device receives a URL longer than 256 UTF-8 encoded characters it MUST reject the installation and return *Status Code 906* in the status report.
MIDlet-Delete-Confirm[1]	A text message to be provided to the user to report the deletion of this IMlet suite

[1] The name of the attribute does not fit the purpose very exactly, but the IMPNGEG decided to choose it due to reasons of compatibility with MIDP 2.0.

4.2.9 User-Agent Product Tokens

The IMP-NG specification identifies HTTP *User-Agent* request headers to identify the client to the server. RFC2616 specifies a format for product tokens such as:

"User-Agent" ":" 1(product | comment)*

The product tokens used to identify the device as supporting CLDC and IMP-NG are specified the Networking portion of the IMP-NG specification. As in RFC2616, the comment field is optional.

In addition, the device SHOULD further identify itself by adding a device-specific product token to the *User-Agent* header as defined by RFC2616. The device-identifying token SHOULD be the first token. The producttoken and product-version values are specific to each device and are outside of the scope of this specification.

4.2.10 Accept Header

The *Accept* HTTP header is used to indicate the type of content being requested. When requesting IMlet suites, this header SHOULD include *application/java-archive*. For retrieving application descriptors, this header SHOULD include *text/vnd.sun.j2me.app-descriptor*.

4.2.11 Example: HTTP Request for Application Descriptor

When requesting the download of an Application Descriptor, the request headers might look as follows:

```
GET http://host.foo.bar/app-dir/weather-station.jad HTTP/1.1
Host: host.foo.bar
Accept: text/vnd.sun.j2me.app-descriptor
User-Agent: CoolDevice/1.4 Profile/IMP-NG Configuration/CLDC-1.0
Accept-Charset: utf-8
```

The response headers from the server might look as follows:

```
HTTP/1.1 200 OK
Server: CoolServer/1.3.12
Content-Length: 2345
Content-Type: text/vnd.sun.j2me.app-descriptor; charset=utf-8
```

4.2.12 Example: HTTP Request to Install/Update an IMlet suite

When requesting the download of an IMlet suite JAR file, the request headers might look as follows:

```
GET http://host.foo.bar/app-dir/weather-station.jar HTTP/1.1
Host: host.foo.bar
Accept: application/java, application/java-archive
```

The response headers from the server might look as follows:

```
HTTP/1.1 200 OK
Server: CoolServer/1.3.12
Content-Length: 25432
Content-Type: application/java-archive
```

4.2.13 Example: Install Status via HTTP Post Request

For example, installing an IMlet suite with an application descriptor given below:

```
...
MIDlet-Install-Notify: http://foo.bar.com/status
...
```

After a successful install of the IMlet suite, the following would be posted:

```
POST http://foo.bar.com/status HTTP/1.1
Host: foo.bar.com
Content-Length: 13
900 Success
```

The response from the server might be:

```
HTTP/1.1 200 OK
Server: CoolServer/1.3.12
```

5 Security for IMP-NG Applications

The IMP specification constrained (like the MIDP 1.0 specification for MIDlets) each IMlet suite to operate in a sandbox wherein all of the APIs available to the IMlets would prevent access to sensitive APIs or functions of the device. That sandbox concept is used in this specification and all untrusted IMlet suites are subject to its limitations. Every implementation of this specification MUST support running untrusted IMlet suites; the untrusted domain policy can deny the access to particular APIs, though. The reason for this practice, which differs of the one for MIDP 2.0 is, that there is no possibility for the user to give permission for the access to sensitive APIs interactively, as it is required for MIDs. This is due to the lack of a user interface at IMs, avoiding interaction of the user during applications are running.

MIDP 2.0 introduced the concept of trusted applications that may be permitted to use APIs that are considered sensitive and are restricted. As a strict subset of MIDP 2.0, IMP-NG will also provide this concept to IMP-NG enabled devices. If and when a device determines that an IMlet suite can be trusted then access is allowed as indicated by the domain policy. The section 5.1. Untrusted and Trusted IMlet Suites below describes the concepts. Any IMlet suite that is not trusted by the device MUST be run as untrusted. If errors occur in the process of verifying that an IMlet suite is trusted then the IMlet suite MUST be rejected.

5.1 Untrusted and Trusted IMlet Suites

Security for Trusted MIDlet suites – as defined in the MIDP 2.0 specification – is based on protection domains. Each protection domain defines the permissions that may be granted to a MIDlet suite in that domain. The protection domain owner specifies how the device identifies and verifies that it can trust a MIDlet suite and bind it to a protection domain with the permissions that authorize access to protected APIs or functions.

IMP-NG will use the same mechanism, but the model is much easier, because there are not so many parties providing applications for IMs. On the other hand, there has to be an additional domain for untrusted applications. This is necessary due to the fact, that there is no possibility for the user to interactive decide about particular method calls to be processed or not.

The chapter 6. Trusted IMlet Suites using X.509 PKI describes a mechanism for identifying trusted IMlet suites though signing and verification. If an implementation of this specification will recognize IMlet suites signed using PKI as trusted IMlet suites they must be signed and verified according to the formats and processes specified in chapter 6.Trusted IMlet Suites using X.509 PKI.

TABLE 5-1: Definition of Terms

Term	Definition
Protection Domain	A set of *Allowed* permissions that may be granted to an IMlet suite
Permission	A named permission defined by an API or function to prevent it from being used without authorization
Trusted IMlet Suite	An IMlet suite for which the authentication and the integrity of JAR file can be trusted by the device and bound to a protection domain which is not the Untrusted protection domain.
Untrusted IMlet Suite	An IMlet suite for which the authentication and the integrity of JAR file cannot be trusted by the device and which is therefore bound to the Untrusted protection domain.

In IMP-NG, security for untrusted IMlet suites is also based on protection domains – there is a protection domain for untrusted applications defining API access for untrusted IMlet suites. An untrusted IMlet suite is an IMlet suite for which the origin and the integrity of the JAR file can NOT be trusted by the device. Untrusted IMlet suites MUST execute in the untrusted domain using a restricted environment where access APIs or functions is ruled by the untrusted domain policy. Any IMP compliant IMlet suite MUST be able to run in an implementation of this specification as untrusted – if the policy of the domain for untrusted applications gives permission for all security sensitive APIs and functions used by this application. Any APIs or functions of this specification which are not security sensitive, having no permissions defined for them, are implicitly accessible by both trusted and untrusted IMlet suites.

While MIDP 2.0 defines APIs that must be accessible from untrusted applications with or without comfirmation by the user, in IMP-NG the access to security sensitive APIs depends on the untrusted domain policy.

Every domain, for trusted or for untrusted applications, MUST allow access to:

API	**Description**
`javax.microedition.rms`	RMS APIs
`javax.microedition.midlet` except the method `platformRequest(String URL)`	MIDlet Lifecycle APIs
`javax.microedition.media` `javax.microedition.media.control`	The multi-media APIs for playback of sound

5.2 Authorization Model

The authorization of an (trusted or untrusted) IMlet suite for IMP-NG depends and only depends on the permissions given by the security protection domain the IMlet suite is bound to.

This is different from MIDP 2.0, where other elements like requested permissions and user decisions are also relevant factors when establishing the authorization.

5.2.1 Assumptions

- IMlets do not need to be aware of the security policy except for security exceptions that may occur when using APIs.
- An IMlet suite – even an untrusted one – is subject to a single protection domain and its permissible actions.
- The internal representation of protection domains and permissions is implementation specific.
- The device must protect the security policy and protection domain information stored in the device from viewing or modification except by authorized parties.
- If the security policy for a device is static and disallows use of some functions of the security framework then the implementation of unused and inaccessible security functions may be removed.
- Security policy allows an implementation to restrict access but MUST NOT be used to avoid implementing functionality. For example, unimplemented protocols under the Generic Connection framework MUST throw `ConnectionNotFoundException`.

5.2.2 Permissions

Permissions are the means to protect access to APIs or functions, which require explicit authorization before being invoked. Permissions described in this section only refer to those APIs and functions which need security protection and do not refer to other APIs which can be accessed by every by both trusted and untrusted IMlet suite and do not need explicit permission. Permissions are checked by the implementation prior to the invocation of the protected function.

The names of permissions have a hierarchical organization similar to Java package names. The names of permissions are case sensitive. All of the permissions for an API MUST use the prefix that is the same as the package name of the API. If the permission is for a function of a specific class in the package then the permission MUST include the package and classname. The set of valid characters for permissions is the same as that for package and class names. The conventions for use of capitalization in package names SHOULD be used for permission names; for example, `javax.microedition.io`. Following the permission name, whether by package or class, additional modifiers may be appended with a separator of "." (Unicode U+002E).

Each API in this specification that provides access to a protected function will define the permissions. For APIs defined outside of IMP-NG there must be a single document that specifies any necessary permissions and the behavior of the API when it is implemented on IMP-NG.

5.2.3 Permissions for Protected Functions

Each function (or entire API), which was identified as protected, must have its permission name defined in the class or package documentation for the API.

Refer to the documentation of the `javax.microedition.io` package for permissions on all Generic Connection schemes defined in this specification. All APIs and functions within this specification that do not explicitly define permissions MUST be made available to all trusted and untrusted IMlet suites.

5.2.4 Requesting Permissions for an IMlet Suite

An IMlet suite that requires access to protected APIs or functions must request the corresponding permissions. Listing the permissions in the attribute `MIDlet-Permissions` can require permissions requested. These permissions are critical to the function of the IMlet suite and it will not operate correctly without them.

If the IMlet suite can function correctly with or without particular permission(s) it should request them using the `MIDlet-Permissions-Opt` attribute. The IMlet suite is able to run with reduced functionality (for example, as a weather station accessing only the measurements of local sensors instead of additionally accessing measurements of remote sensors via HTTP) without these non-critical permissions and MUST be installed and run.

The `MIDlet-Permissions` and `MIDlet-Permissions-Opt` attributes contain a list of one or more permissions. Multiple permissions are separated by a comma (Unicode U+002C). Leading and trailing whitespace (Unicode U+0020) and tabs (Unicode U+0009) are ignored.

5.2.5 Permissions on the Device

Each device that implements this specification and any other Java APIs will have a total set of permissions referring to protected APIs and functions. It is the union of all permissions defined by every protected function or API on the device.

5.2.6 Protection Domain

A protection domain defines a set of permissions. A protection domain consists of a set of permissions that should be allowed.

All permissions explicitly allow access to a given protected API or function on the basis of IMlet suite being associated with the protection domain. Permissions in IMP-NG have the character of *Allowed* permissions in MIDP 2.0 and do not require any user interaction, which cannot be realized on IMs. Naturally, there is nothing like *User* permissions (as defined in MIDP 2.0) for IMP-NG.

5.2.7 Granting Permissions to IMlet Suites

Authorization of IMlet suites uses protection domain information, and permissions on the device. Permissions requested by the application are either critical or non-critical.

To establish the permissions granted to an IMlet suite when it is to be invoked all of the following MUST be true:

- The requested critical permissions are retrieved from the attributes `MIDlet-Permissions` and noncritical permissions from `MIDlet-Permissions-Opt`. If these attributes appear in the application descriptor they MUST be identical to corresponding attributes in the manifest. If they are not identical, the IMlet suite MUST NOT be installed or invoked.
- If any of the requested permissions are unknown to the device and are not marked as critical then they are removed from the requested permissions.
- If any of the requested permissions are unknown to the device and marked as critical, the IMlet suite MUST NOT be installed or invoked.
- If any of the requested permissions are not present in the protection domain permission sets and the requested permission was marked as critical then the IMlet suite does not have sufficient authorization and MUST NOT be installed or invoked.
- If any of the requested permissions are not present in the protection domain permission sets, and the requested permissions are not marked as critical, the application MUST still be installed and MUST be able to be invoked by the user.
- The permissions granted to the IMlet suite are permissions granted by the security protection domain the IMlet is bound to.
- During execution, protected APIs MUST check for the appropriate permissions and throw a `SecurityException` if the permission has not been granted.

The successful result of authorization is that the IMlet suite is granted access to protected APIs or functions used.

5.2.8 Example External Domain Policy Format

An external representation for protection domains allows clear communication between users and manufacturers. This format is provided only as an example. There is no requirement for an implementation of this specification to use this format. The policy file character set is UTF-8 encoding of Unicode to support any language. The policy file syntax is based on the JAR manifest format.

A policy consists of the definitions of domains and aliases. Each domain consists of the definition of permissions. Aliases permit groups of named permissions to be reused in more than one domain and helps keep the policy compact. Aliases may only be defined and used within a single policy file. References to an alias MUST follow the definition of the alias in the policy file.

A domain is defined with a domain identifier and a sequence of permissions. The domain identifier is implementation specific. Each permission line begins with “`allow`”. Multiple permission lines are allowed. The permissions are processed in order and if a permission occurs multiple times within a domain only the last definition of the permission is used. It is not recommended that permissions appear more than once.

BNF Syntax:

```
policy_file = 1*(directive)
directive = (domain_def | alias_def) [newlines]
domain_def = "domain: " *WS domain_id *WS newlines
             1*permission
domain_id = 1*<any Unicode char and continuation, but not newline>
permission = allow ":" api_names newlines
api_names: *WS alias_or_name *(*WS "," *WS alias_or_name) *WS
alias_or_name = alias_ref | api_name
alias_ref = <alias_name from a previous alias_def in the same policy_file>
allow = "allow" ; allow access
alias_def = "alias: " *WS alias_name 1*WS alias_api_names
alias_api_names = api_name
                  *(*WS "," *WS api_name) *WS newlines
alias_name = java_name
api_name = java_class_name
```

```
WS = continuation | SP | HT
continuation = newline SP
newlines = 1*newline ; allow blank lines to be ignored
newline = CR LF | LF | CR <not followed by LF>
CR = <Unicode carriage return (U+000D)>
LF = <Unicode linefeed (U+000A)>
SP = <Unicode space (U+0020)>
HT = <Unicode horizontal-tab (U+0009)>
java_name = 1*<characters allowed in a java_class_name except for "."> 
java_class_name = 1*<characters allowed in a Java class name>
```

Example policy file:

```
domain: untrusted
allow: javax.microedition.io.HttpConnection
domain: O="IMlet Underwriters, Inc. ", C=US
allow: javax.microedition.io.HttpConnection
allow: javax.microedition.io.CommConnection
alias: client_connections javax.microedition.io.SocketConnection,
   javax.microedition.io.SecureConnection,
   javax.microedition.io.HttpConnection,
   javax.microedition.io.HttpsConnection
domain: O=Acme Wireless, OU=Software Assurance
allow: client_connections
allow: javax.microedition.io.ServerSocketConnection,
   javax.microedition.io.UDPDatagramConnection
allow: javax.microedition.io.CommConnection
domain: allnet
allow: client_connections
allow: javax.microedition.io.CommConnection
```

6 Trusted IMlet Suites using X.509 PKI

Signed IMlet suites may become trusted by authenticating the signer of the IMlet suite and binding it to a protection domain that will authorize the IMlet suite to perform protected functions by granting permissions allowed in the protection domain. The mechanisms defined here allow signing and authentication of IMlet suites based on X.509 Public Key Infrastructure so the device can verify the signer and trust the IMlet suite.

If an implementation of this specification will recognize IMlet suites signed using PKI as trusted IMlet suites they MUST be signed and verified according to the formats and processes below.

The IMlet suite is protected by signing the JAR. The signature and certificates are added to the application descriptor as attributes. The device uses them to verify the signature. The device completes the authentication using a root certificate bound to a protection domain on the device. The details of the processes and formats follow.

6.1 Definition of Terms

The terms *Untrusted IMlet suite*, *Trusted IMlet suite*, *Permission*, and *Protection Domain* are defined in chapter 5.Security for IMP-NG Applications.

The following additional term is defined:

Term	**Definition**
Protection Domain Root Certificate	A certificate associated with a protection domain that the device implicitly trusts to verify and authorize downloaded IMlet suites

6.2 Signing an IMlet Suite

The security model involves the IMlet suite, a signer, and public key certificates. As with any public key system authentication is based on a set of root certificates, which are used to verify other certificates. Zero or more root certificates will need to be on the device. Additionally, root certificates may be present in removable media such as SIM (WIM) card/USIM module. Implementations MUST support X.509 Certificates and corresponding algorithms. Devices MAY support additional signing mechanisms and certificate formats.

The signer of the IMlet suite will be usually the IM user or some entity that is responsible for distributing, supporting, and perhaps billing for its use. The signer will need to have a public key certificate that can be validated to one of the protection domain root certificates on the device. The public key is used to verify the signature on the IMlet suite. The public key is provided as a RSA X.509 certificate included in the application descriptor.

Attributes defined within the manifest of the JAR are protected by the signature. Attributes defined within the application descriptor are not secured. When an attribute appears in the manifest it *MUST NOT* be overridden by a different value from the application descriptor. For trusted IMlet suites the value in the application descriptor must be equal to the value of the corresponding attribute in the manifest. If not, the IMlet suite MUST NOT be installed. The `MIDlet.getAppProperty` method must return the attribute value from the manifest if one is defined. If not, the value from the application descriptor (if any) is returned.

Note that the requirement that attributes values be equal differs from what was defined for IMP (JSR-195). The IMP requirements were taken from MIDP 1.0 (for it is a subset of this profile), while the requirements of IMP-NG are derived from MIDP 2.0. They must be used for applications that are signed and verified by these procedures. For untrusted application descriptors, the MIDP 1.0 rule giving priority to application descriptor attributes over manifest attributes must be followed.

6.2.1 Creating the Signing Certificate

1. The signer will need to be aware of the authorization policy for the device and contact the appropriate certificate authority. For example, the signer may need to send its distinguished name (DN) and public key (normally, packaged in a certificate request) to a certificate authority.
2. The CA creates a RSA X.509 (version 3) certificate and returns it to the signer.
3. If multiple CA's are used then all the signer certificates in the application descriptor MUST contain the same public key.

6.2.2 Insert Certificates into the application descriptor

1. The certificate path includes the signer certificate and any necessary certificates but omitting the root certificate. The root certificate will be found on the device.
2. Each certificate in the path is encoded (using base64 but without line breaks) and inserted into the application descriptor as:
   `MIDlet-Certificate-<n>-<m>: <base64 encoding of a certificate>`
   `<n>`:= a number equal to 1 for first certification path in the descriptor or 1 greater than the previous number for additional certification paths. This defines the sequence in which the certificates are tested to see if the corresponding root certificate is on the device. See the Authenticating an IMlet suite section below.
   `<m>`:= a number equal to 1 for the signer's certificate in a certification path or 1 greater than the previous number for any subsequent intermediate certificates.

6.2.3 Creating the RSA SHA-1 signature of the JAR

1. The signature of the JAR is created with the signers private key according to the EMSA-PKCS1-v1_5 encoding method of PKCS #1 version 2.0 standard[RFC2437].
2. The signature is base64 encoded, formatted as a single `MIDlet-Jar-RSA-SHA1` attribute without line breaks and inserted in the application descriptor.
   `MIDlet-Jar-RSA-SHA1: <base64 encoding of Jar signature>`

It should be noted that the signer of the IMlet suite is responsible to its protection domain root certificate owner for protecting the protection domain stake holder's assets and capabilities and, as such, must exercise due-diligence in checking the IMlet suite before signing it. In the case where there is a trusted relationship (possibly bound by legal agreements), a protection domain root certificate owner may delegate signing IMlet suites to a third-party and in some circumstances, the author of the IMlet.

6.2.4 Authenticating an IMlet Suite

When an IMlet suite is downloaded, the device MUST check if authentication is required. If the attribute `MIDlet-Jar-RSA-SHA1` is present in the application descriptor then the JAR MUST be authenticated by verifying the signer certificates and JAR signature as below.

Application descriptors without the `MIDlet-Jar-RSA-SHA1` attribute are not authenticated but are installed and invoked as untrusted IMlet suites.

6.2.5 Verify Signer Certificate

The certification path consists of the signer certificate from the application descriptor and other certificates as needed up to but not including the root certificate.

1. Get the certification path for the signer certificate from the application descriptor attributes `MIDlet-Certificate-1-<m>` where `<m>` starts at 1 and is incremented by 1 until there is no attribute with the given name. The value of each attribute is a base64 encoded certificate that will need to be decoded and parsed.

2. Validate the certification path using the basic path validation processes described in RFC2459 using the protection domains as the authoritative source of protection domain root certificates. Bind the IMlet suite to the protection domain that contains the protection domain root certificate that validates the first chain from signer to root and proceed with installation.

3. If attributes `MIDlet-Certificate-<n>-<m>` with `<n>` greater than 1 are present and full certification path could not be established after verifying `MIDlet-Certificate-<1>-<m>` certificates, repeatedly perform steps 1 and 2 for the value <n> greater by 1 than the previous value. The results of certificate verification are gathered into the Table 1.

TABLE 6-1: Actions upon completion of signer certificate verification.

Result	Action
Attempted to validate <n> paths. No public keys of the issuer for the certificate can be found or none of the certification paths can be validated	Authentication fails, JAR Installation is not allowed.
More than one full certificate path established and validated	Implementation proceeds with the signature verification using the first successfully verified certificate path is used for authentication and authorization.
Only one full certificate path established and validated	Implementation proceeds with the signature verification

6.2.6 Verify the IMlet Suite JAR

1. Get the public key from the verified signer certificate (above).
2. Get the `MIDlet-Jar-RSA-SHA1` attribute from the application descriptor.
3. Decode the attribute value from base64 yielding a PKCS #1 signature [RFC2437].
4. Use the signer's public key, signature, and SHA-1 digest of the JAR, to verify the signature. If the signature verification fails, reject the application descriptor and IMlet suite. The implementation MUST NOT install the JAR on failure or allow IMlets from the IMlet suite to be invoked.

Once the steps of verifying the certificate, verifying the signature and verifying the JAR all succeed then the IMlet suite contents are known to be intact and the identity of the signer is known. This process must be performed during installation.

6.2.7 Summary of IMlet suite source verification results

It is essential that the steps performed to verify the digital signature as described above lead to the proof of the identity of the IMlet suite signer. The results of the verification have a direct impact on authorization. The following, Table 2, summarizes the states to which the signature verification led and which are further used for authorization at install time.

TABLE 6-2: Summary of IMlet suite source verification

Initial state	Verification result
JAD not present, JAR downloaded	Authentication can not be performed, may install JAR. IMlet suite is treated as untrusted
JAD present but is JAR is unsigned	Authentication can not be performed, may install JAR. IMlet suite is treated as untrusted
JAR signed but no root certificate present in the keystore to validate the certificate chain	Authentication can not be performed, JAR installation is not allowed
JAR signed, a certificate on the path is expired	Authentication can not be completed, JAR installation is not allowed
JAR signed, a certificate rejected for reasons other than expiration	JAD rejected, JAR installation is not allowed
JAR signed, certificate path validated but signature verification fails	JAD rejected, JAR installation is not allowed
JAR signed, certificate path validated, signature verified	JAR installation is allowed

6.2.8 Caching of Authentication and Authorization Results

The implementation of the authentication and authorization process may store and transfer the results for subsequent use and MUST ensure that the cached information practically can not be tampered with or otherwise compromised between the time it is computed from the JAR, application descriptor, and authentication information and the authorization information is used.

It is essential that the IMlet suite and security information used to authenticate and authorize an IMlet suite is not compromised, for example, by use of removable media or other access to IMlet suite storage that might be corrupted.

6.2.9 Security in Split-VM Implementations

In environments that make use of a split VM (CLDC 5.4.6), it is possible to implement the security mechanism using JARs but this relies on converting the JAR to the device format when the JAR enters the network while faithfully preserving the semantics of the IMlet. Once the conversion has happened, the device format of the application must be secured against tampering and retain its authorized permissions. This network security is often based on similar digital signature techniques to IMlet security and it may be the case that this network security infrastructure is already present and active. If and only if this kind of network security infrastructure already exists and it can support all forms of protection required by this specification (and any future JSRs based on this specification), a permissible implementation of IMlet Suite Security can be based on authenticating and authorizing the device format of the IMlet since these implementation formats are the actual executable content that will be used on the device. The details of authenticating and authorizing a device format version of an IMlet suite are implementation specific and thus not covered by this specification.

6.3 IMP-NG X.509 Certificate Profile for Trusted IMlet Suites

Secured trusted IMlet suites utilize the same base certificate profile as does HTTPS. The profile is based on the WAP Certificate Profile, WAP-211-WAPCert-20010522-a [WAPCert] which is based on RFC2459 Internet X.509 Public Key Infrastructure Certificate and CRL Profile [RFC2459]. Refer to the package documentation for `javax.microedition.pki` for details.

6.3.1 Certificate Processing for OTA

Devices MUST recognize the key usage extension and when present verify that the extension has the `digitalSignature` bit set. Devices MUST recognize the critical extended key usage extension and when present verify that the extension contains the `id-kp-codeSigning` object identifier (see RFC2459 sec. 4.2.1.13).

The application descriptor SHOULD NOT include a self-issued root certificate in a descriptor certificate chain. However IMP-NG devices SHOULD treat the certificate as any other in a chain and NOT explicitly reject a chain with an X.509v3 self-issued CA certificate in its chain.

6.3.2 Certificate Expiration and Revocation

Expiration and revocation of certificates supplied in the application descriptor is checked during the authorization procedure, specifically during certificate path validation. Certificate expiration is checked locally on the device as such information is retrievable from the certificate itself. Certificate expiration verification is an intrinsic and mandatory part of certificate path validation.

Certificate revocation is a more complex check as it requires sending a request to a server and the decision is made based on the received response. Certificate revocation can be performed if the appropriate mechanism is implemented on the device. Such mechanisms are not part of an IMP-NG implementation and hence do not form a part of the IMP-NG security framework.

If certificate revocation is implemented in the device, it SHOULD support Online Certificate Status protocol (OCSP) [RFC2560]. If other certificate revocation protocols are supported, support for these other protocols may indicate that a certificate has been revoked; in this case, it is permissible to consider the certificate as revoked regardless of the result returned by the OCSP protocol.

6.3.3 Example – User Generates Own Signing Certificate

There are many ways to structure protection domain root certificates and their associated signing policies. This example is provided to illustrate some of the concepts in this specification and is not meant to limit the ways IMlet PKI signing can be used. The example allows IMlets to be revoked (provided the device supports certificate revocation) but at differing granularities.

This encodes the origin of the IMlet suite into the JAD (via the identity of the signer). If the certificate is revoked, all of the developer’s signed IMlets on every device will have their execution permissions revoked.

1. User creates IMlet network application
2. User encodes permissions into JAR manifest and creates final IMlet JAR
3. User generates a private-public key pair with a signing certificate and has the certificate signed by one or more protection domain root certificates
4. The user’s certificate is used to sign the IMlet JAR and create the associated JAD entries
5. IMlet JAR can be distributed with a suitably populated JAD and run on the IMP-NG compliant device with the appropriate protection domain root certificate

7 Package java.lang

Description

IM Profile – Next Generation Language Classes are the same as MID Profile version 2.0 Language Classes included from Java 2 Standard Edition. In addition to the `java.lang` classes specified in the Connected Limited Device Configuration the IMP-NG includes the following class from Java 2 Standard Edition.

- `java.lang.IllegalStateException`

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

System Functions

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

System Properties

IMP-NG defines the following property values (in addition to those defined in the CLDC specification) that MUST be made available to the application using `java.lang.System.getProperty` :

System Properties Defined by IMP-NG

System Property	Description
`microedition.locale`	The current locale of the device, may be `null`
`microedition.profiles`	is a blank (Unicode x20) separated list of the J2ME profiles that this device supports. For IMP-NG devices, this property MUST contain at least “IMP-NG”

Other properties may be available from other profiles or the implementation.

Property microedition.locale

The locale property, if not `null`, MUST consist of the language and MAY optionally also contain the country code, and variant separated by “-” (Unicode x2D). For example, “fr-FR” or “en-US.” (Note: the IMP and MIDP 1.0 specifications used the HTTP formatting of language tags as defined in RFC3066 (http://www.ietf.org/rfc/rfc3066.txt) *Tags for the Identification of Languages*. This is different from the J2SE definition for `Locale` printed strings where fields are separated by “_” (Unicode x5F).)

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).

Application Resource Files

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

Resource names refer to the contents of the IMlet Suite JAR file. Absolute pathnames, beginning with “/” are fully qualified file names within the jar file.

Relative pathnames, not beginning with “/” are relative to the class upon which `getResourceAsStream` is called. Relative names are converted to absolute by prepending a “/” followed by the fully qualified package with “.” characters converted to “/” and a separator of “/”. The resulting string is reduced to canonical form as follows:

- All occurences of “/./” are replaced with “/”.
- All occurences of “/segment/../” are replaced with “/” where segment does not contain “/”.

The canonical resource name is the absolute pathname of the resource within the JAR.

In no case can the path extend outside the JAR file, and resources outside the JAR file MUST NOT be accessible. For example, using “../../” does NOT point outside the JAR file. Also, devices MUST NOT allow classfiles to be read from the JAR file as resources, but all other files MUST be accessible.

`System.exit`

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

`Runtime.exit`

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

Since: IMP / MIDP 1.0

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

java.lang

IllegalStateException

Declaration

```
public class IllegalStateException extends RuntimeException

Object
   |
   +--Throwable
         |
         +--Exception
               |
               +--RuntimeException
                     |
                     +--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: IMP / MIDP 1.0

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

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

Constructors

IllegalStateException()

Declaration:
`public IllegalStateException()`

Description:
Constructs an IllegalStateException with no detail message.

IllegalStateException(String)

Declaration:
`public IllegalStateException(String s)`

Description:
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

8 Package java.util

Description

IM Profile – Next Generation Language Classes are the same as MID Profile version 2.0 Language Classes included from Java 2 Standard Edition. In addition to the `java.util` classes specified in the Connected Limited Device Configuration the IMP-NG 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.

Since: IMP / MIDP 1.0

Class Summary	
Interfaces	
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`.
Exceptions	

java.util
Timer

Declaration

```
public class Timer

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, 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. The resolution of the Timer is implementation and device dependent.

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: IMP / MIDP 1.0

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

Package
java.util

Member Summary

Constructors

```
            Timer()
```

Methods

```
      void   cancel()
      void   schedule(TimerTask task, Date time)
      void   schedule(TimerTask task, Date firstTime, long period)
      void   schedule(TimerTask task, long delay)
      void   schedule(TimerTask task, long delay, long period)
      void   scheduleAtFixedRate(TimerTask task, Date firstTime, long period)
      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(), wait()`

Constructors

Timer()

Declaration:
`public Timer()`

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

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

Methods

cancel()

Declaration:
`public void cancel()`

Description:
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)

Declaration:
`public void schedule(java.util.TimerTask task, java.util.Date time)`

Description:
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)

Declaration:
`public void schedule(java.util.TimerTask task, java.util.Date firstTime, long period)`

Description:
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)

Declaration:
`public void schedule(java.util.TimerTask task, long delay)`

Description:
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. Note that the actual delay may be different than the amount requested since the resolution of the Timer is implementation and device dependent.

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)

Declaration:
```
public void schedule(java.util.TimerTask46 task, long delay, long period)
```

Description:
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. Note that the actual delay may be different than the amount requested since the resolution of the Timer is implementation and device dependent.
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. Note that the actual delay may be different than the amount requested since the resolution of the Timer is implementation and device dependent.
`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)

Declaration:
```
public void scheduleAtFixedRate(java.util.TimerTask task, java.util.Date firstTime,
             long period)
```

Description:
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)

Declaration:
`public void scheduleAtFixedRate(java.util.TimerTask task, long delay, long period)`

Description:
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. Note that the actual delay may be different than the amount requested since the resolution of the Timer is implementation and device dependent.
`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.

Package
java.util

java.util
TimerTask

Declaration

```
public abstract class TimerTask implements Runnable

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: IMP / MIDP 1.0

See Also: `Timer`

Member Summary

Constructors

protected	TimerTask()

Methods

boolean	cancel()
abstract void	run()
long	scheduledExecutionTime()

Inherited Member Summary

Methods inherited from class `Object`

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

Constructors

TimerTask()

Declaration:
`protected TimerTask()`

Description:
Creates a new timer task.

Methods

cancel()

Declaration:
`public boolean cancel()`

Description:
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()

Declaration:
`public abstract void run()`

Description:
The action to be performed by this timer task.

Specified By: `run` in interface `Runnable`

scheduledExecutionTime()

Declaration:
`public long scheduledExecutionTime()`

Description:
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()`

9 Package javax.microedition.io

Description

Like MID Profile, IMP-NG also includes networking support based on the `Generic Connection` framework from the *Connected, Limited Device Configuration*.

HTTP Networking

In addition to the `javax.microedition.io` classes specified in the *Connected Limited Device Configuration* the IMP-NG includes the following interface for the HTTP access. An `HttpConnection` is returned from `Connector.open()` when an `"http:// "` connection string is accessed.

- `javax.microedition.io.HttpConnection`

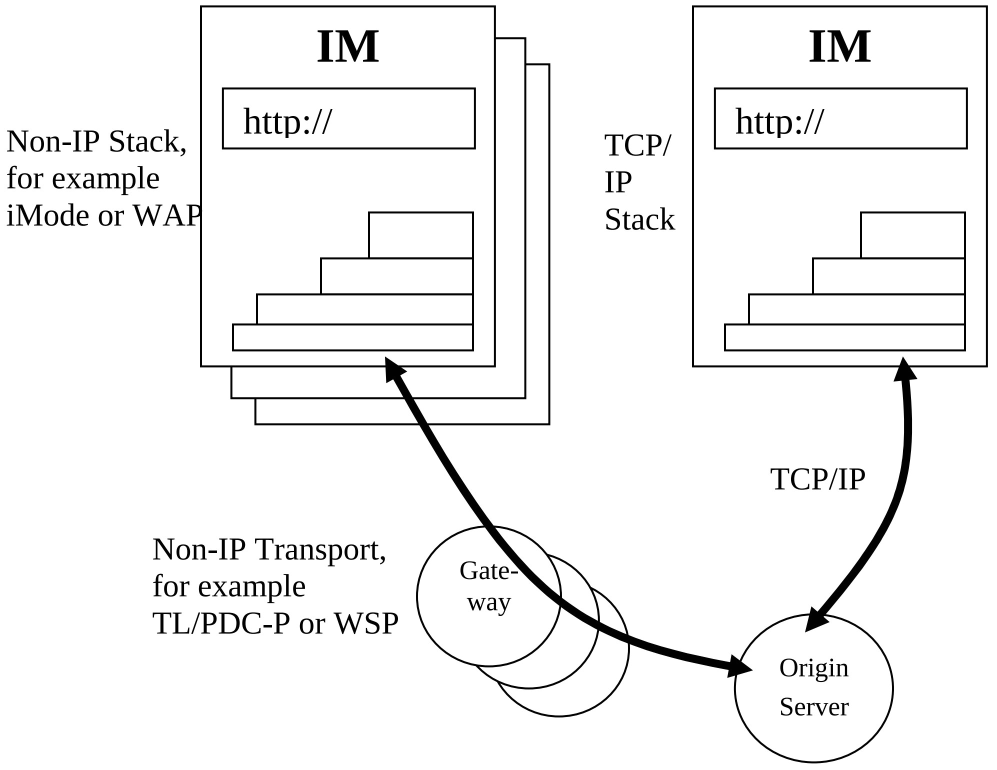
IMP-NG extends the connectivity support provided by the Connected, Limited Device Configuration (CLDC) with specific functionality for the *GenericConnection* framework. Like MIDP, IMP-NG supports a subset of the HTTP protocol, which can be implemented using both IP protocols such as TCP/IP and non-IP 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 IMP-NG will allow the application to be portable to all IMs. IMP-NG 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, an IM MAY have no in-device support for the Internet Protocol (IP). In this case, it would utilize a gateway 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.

HTTP Network Connection


IM
http://
Non-IP Stack, for example iMode or WAP
IM
http://
TCP/ IP Stack
TCP/IP
Non-IP Transport, for example TL/PDC-P or WSP
Gate-way
Origin Server

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.

The interface MUST support:

- HTTP 1.1

Each device implementing IMP-NG MUST support opening connections using the following URL schemes (RFC2396 Uniform Resource Identifiers (URI): Generic Syntax)

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

Each device implementing IMP-NG 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.(RFC2616 takes great care to not to mandate TCP streams as the only required transport mechanism.)

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 IMP-NG, 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 and Content-Language Request Headers

For IMP-NG, 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.

The user-agent and content-language fields SHOULD contain the following features as defined by system properties using `java.lang.System.getProperty`. If multiple values are present they will need to be reformatted into individual fields in the request header.

System Properties Used for User-Agent and Content-Language Request Headers

System Property	Description
`microedition.profiles`	A blank (Unicode x20) separated list of the J2ME profiles that this device supports. For IMP-NG devices, this property MUST contain at least “IMP-NG”.
`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.”

HTTP Request Header Example
User-Agent: Profile/IMP-NG Configuration/CLDC-1.0
Accept-Language: en-US

StreamConnection Behavior

All IMP-NG `StreamConnections` have one underlying `InputStream` and one `OutputStream`. Opening a `DataInputStream` counts as opening an `InputStream` and opening a `DataOutputStream` counts as opening an `OutputStream`. Trying to open another `InputStream` or another `OutputStream` from a `StreamConnection` causes an `IOException`. Trying to open `InputStream` or `OutputStream` after they have been closed causes an `IOException`.

After calling the `close` method, regardless of open streams, further method calls to connection will result in IOExceptions for those methods that are declared to throw `IOExceptions`. For the methods that do not throw exceptions, unknown results may be returned.

The methods of `StreamConnections` are not synchronized. The only stream method that can be called safely in another thread is `close`. When `close` is invoked on a stream that is executing in another thread, any pending I/O method MUST throw an `InterruptedIOException`. In the above case implementations SHOULD try to throw the exception in a timely manner. When all open streams have been closed, and when the `StreamConnection` is closed, any pending I/O operations MUST be interrupted in a timely manner.

Secure Networking

Since the MIDP 2.0 release additional interfaces are available for secure communication with WWW network services. IMP-NG as a strict subset of MIDP 2.0 supports these additional interfaces. Secure interfaces are provided by HTTPS and SSL/TLS protocol access over the IP network. Refer to the package documentation of `javax.microedition.pki` for the details of certificate profile that applies to secure connections. An `HttpsConnection` is returned from `Connector.open()` when an `"https:// "` connection string is accessed. A `SecureConnection` is returned from `Connector.open()` when an `"ssl:// "` connection string is accessed.

- `javax.microedition.io.HttpsConnection`
- `javax.microedition.io.SecureConnection`
- `javax.microedition.io.SecurityInfo`
- `javax.microedition.pki.Certificate`
- `javax.microedition.pki.CertificateException`

Low Level IP Networking

Since the MIDP 2.0 release, the MIDP specification also includes optional networking support for TCP/IP sockets and UDP/IP datagrams. IMP-NG as a strict subset of MIDP 2.0 supports these additional interfaces. For each of the following schemes, a host is specified for an outbound connection and the host is omitted for an inbound connection. The host can be a host name, a literal IPv4 address or a literal IPv6 addresss (according to RFC2732 square bracket characters '[' ']' may be used to designate an IPv6 address in URL strings). Implementations MUST be able to parse the URL string and recognize the address format used, but are not required to support all address formats and associated protocols.

When the host and port number are both omitted from the `socket` or `datagram` connection, the system will allocate an available port. The host and port numbers allocated in this fashion can be discovered using the `getLocalAddress` and `getLocalPort` methods. The colon (:) may be omitted when the connection string does not include the port parameter.

A `SocketConnection` is returned from `Connector.open()` when a `"socket://host:port"` connection string is accessed. A `ServerSocketConnection` is returned from `Connector.open()` when a `"socket://:port"` connection string is accessed. A `UDPDatagramConnection` is returned from `Connector.open()` when a `"datagram://host:port"` connection string is accessed.

- `javax.microedition.io.SocketConnection`
- `javax.microedition.io.ServerSocketConnection`
- `javax.microedition.io.DatagramConnection`
- `javax.microedition.io.Datagram`
- `javax.microedition.io.UDPDatagramConnection`

Push Applications

A `PushRegistry` is available in the IMP-NG release which provides an IMlet with a means of registering for network connection events, which may be delivered when the application is not currently running.

- `javax.microedition.io.PushRegistry`

Serial Port Communications

A `CommConnection` is available in the IMP-NG release which provides an IMlet with a means of registering for network accessing a local serial port as a stream connection.

- `javax.microedition.io.CommConnection`

Security of Networking Functions

The security model is found in the package `javax.microedition.midlet` and provides a framework that allows APIs and functions to be restricted to IMlet suites that have been granted permissions either by signing or explicitly by the user. (See chapter **Error! Reference source not found.**. **Error! Reference source not found.** for details about granting specific permissions to an IMlet suite.)

The risks associated with an IMlet suite's use of the network are related the potential for network abuse and to costs to the device owner since network use may result in charges. IMP-NG provides a security framework in which network functions can be protected and allowed only to those applications that are bound to a Security Protection Domain containing the appropriate permissions.

Package javax.microedition.io

Each protocol is accessed by invoking `javax.microedition.io.Connector.open` with a URI including the protocol and arguments. The permissions below allow access to be granted individually to protocols. The functionality of the protocols is specified by subclasses of `Connection` interface that defines the syntax of the URI and any protocol specific methods. Devices are NOT REQUIRED to implement every protocol. If a protocol is implemented, the security framework specifies the naming of permissions according to the package and class name of the APIs used to access the protocol extended with the protocol name. The API providing access is `javax.microedition.io.Connector.open`. The table below defines the corresponding permissions for the protocols defined within this specification.

Permission	Protocol
`javax.microedition.io.Connector.http`	`http`
`javax.microedition.io.Connector.https`	`https`
`javax.microedition.io.Connector.datagram`	`datagram`
`javax.microedition.io.Connector.datagramreceiver`	`datagram server (without host)`
`javax.microedition.io.Connector.socket`	`socket`
`javax.microedition.io.Connector.serversocket`	`server socket (without host)`
`javax.microedition.io.Connector.ssl`	`ssl`
`javax.microedition.io.Connector.comm.`	`comm`

Security of `Push Registry`

The `PushRegistry` is protected using the security framework and permissions. The IMlet suite must have the `javax.microedition.io.PushRegistry` permission to register an alarm based launch, to register dynamically using the `PushRegistry`, or to make a static registration in the application descriptor. How the launch of another IMlet via `PushRegistry` is regulated regarding security issues is up to the implementation. An implementation is also free to avoid external Push by not implementing any protocol for it.

The push mechanism uses protocols in which the device is acting as the server and connections can be accepted from other elements of the network. To use the push mechanisms the IMlet suite will need the permission to use the server connection. For example, to register a sample program that can be started via push might use the following attributes in the manifest:

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

Since: IMP / MIDP 1.0

Class Summary	
Interfaces	
`CommConnection`	This interface defines a logical serial port connection.
`HttpConnection`	This interface defines the necessary methods and constants for an HTTP connection.
`HttpsConnection`	This interface defines the necessary methods and constants to establish a secure network connection.
`SecureConnection`	This interface defines the secure socket stream connection.
`SecurityInfo`	This interface defines methods to access information about a secure network connection.
`ServerSocketConnection`	This interface defines the server socket stream connection.
`SocketConnection`	This interface defines the socket stream connection.
`UDPDatagramConnection`	This interface defines a datagram connection which knows it’s local end point address.
Classes	
`Connector`	Factory class for creating new Connection objects.
`PushRegistry`	The `PushRegistry` maintains a list of inbound connections.
Exceptions	

javax.microedition.io

CommConnection

Declaration

`public interface` **`CommConnection extends StreamConnection`**

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

Description

This interface defines a logical serial port connection. A "logical" serial port is defined as a logical connection through which bytes are transferring serially. The logical serial port is defined within the underlying operating system and may not necessarily correspond to a physical RS-232 serial port. For instance, IrDA IRCOMM ports can commonly be configured as a logical serial port within the operating system so that it can act as a "logical" serial port.

A comm port is accessed using a Generic Connection Framework string with an explicit port identifier and embedded configuration parameters, each separated with a semi-colon (;).

Only one application may be connected to a particular serial port at a given time. A `java.io.IOException` is thrown, if an attempt is made to open the serial port with `Connector.open()` and the connection is already open.

A URI with the type and parameters is used to open the connection. The scheme (defined in RFC 2396) must be:

```
comm:<port identifier>[<optional parameters>]
```

The first parameter must be a port identifier, which is a logical device name. These identifiers are most likely device specific and should be used with care.

The valid identifiers for a particular device and OS can be queried through the method `System.getProperty()` using the key `"microedition.commports"`. A comma separated list of ports is returned which can be combined with a `comm:` prefix as the URL string to be used to open a serial port connection. (See port naming convention below.)

Any additional parameters must be separated by a semi-colon (;) and spaces are not allowed in the string. If a particular optional parameter is not applicable to a particular port, the parameter MAY be ignored. The port identifier MUST NOT contain a semi-colon (;).

Legal parameters are defined by the defintion of the parameters below. Illegal or unrecognized parameters cause an `IllegalArgumentException`. If the value of a parameter is supported by the device, it must be honored. If the value of a parameter is not supported a `java.io.IOException` is thrown. If a `baudrate` parameter is requested, it is treated in the same way that the `setBaudRate` method handles baudrates. e.g., if the baudrate requested is not supported the system MAY substitute a valid baudrate, which can be discovered using the `getBaudRate` method.

Optional Parameters

Parameter	Default	Description
`baudrate`	`platform dependent`	The speed of the port.
`Bitsperchar`	`8`	The number bits per character (7 or 8).
`Stopbits`	`1`	The number of stop bits per char (1 or 2)
`parity`	`none`	The parity can be odd, even, or none.
`Blocking`	`on`	If on, wait for a full buffer when reading.
`Autocts`	`on`	If on, wait for the CTS line to be on before writing.
`Autorts`	`on`	If on, turn on the RTS line when the input buffer is not full. If off, the RTS line is always on.

BNF Format for `Connector.open()` string

The URI must conform to the BNF syntax specified below. If the URI does not conform to this syntax, an `IllegalArgumentException` is thrown.

`<comm_connection_string>`	`::=` **`"comm:"`**`<port_id>[<options_list>] ;`
`<port_id>`	`::=` *string of alphanumeric characters*

`<options_list>`	`::= *(<baud_rate_string>|<bitsperchar>|<stopbits>| <parity>| <blocking>| <autocts>| <autorts>) ;`
	 `; if an option duplicates a previous option in the`
	 `; option list, that option overrides the previous`
	 `; option`

`<baud_rate_string>`	`::=` **`";baudrate="`**`<baud_rate>`
`<baud_rate>`	`::=` *string of digits*
`<bitsperchar>`	`::=` **`";bitsperchar="`**`<bit_value>`
`<bit_value>`	`::= "7" | "8"`
`<stopbits>`	`::=` **`";stopbits="`**`<stop_value>`
`<stop_value>`	`::= "1" | "2"`
`<parity>`	`::=` **`";parity="`**`<parity_value>`
`<parity_value>`	`::= "even" | "odd" | "none"`
`<blocking>`	`::=` **`";blocking="`**`<on_off>`
`<autocts>`	`::=` **`";autocts="`**`<on_off>`
`<autorts>`	`::=` **`";autorts="`**`<on_off>`
`<on_off>`	`::= "on" | "off"`

Security

Access to serial ports is restricted to prevent unauthorized transmission or reception of data. The security model applied to the serial port connection is defined in the implementing profile. The security model may be applied on the invocation of the `Connector.open()` method with a valid serial port connection string. Should the application not be granted access to the serial port through the profile authorization scheme, a `java.lang.SecurityException` will be thrown from the `Connector.open()` method. The security model MAY also be applied during execution, specifically when the methods `openInputStream()`, `openDataInputStream()`, `openOutputStream()`, and `openDataOutputStream()` are invoked.

Examples

The following example shows how a `CommConnection` would be used to access a simple loopback program.

```
CommConnection cc = (CommConnection)Connector.open("comm:com0;baudrate=19200");
int baudrate = cc.getBaudRate();
InputStream is = cc.openInputStream();
OutputStream os = cc.openOutputStream();
int ch = 0;
while(ch != 'Z') {
    os.write(ch);
    ch = is.read();
    ch++;
}
is.close();
os.close();
cc.close();
```

The following example shows how a `CommConnection` would be used to discover available comm ports.

```
String port1;
String ports = System.getProperty("microedition.commports");
int comma = ports.indexOf(',');
if (comma > 0) {
    // Parse the first port from the available ports list.
    port1 = ports.substring(0, comma);
} else {
    // Only one serial port available.
    port1 =ports;
}
```

Recommended Port Naming Convention

Logical port names can be defined to match platform naming conventions using any combination of alphanumeric characters. However, it is recommended that ports be named consistently among the implementations of this class according to a proposed convention. VM implementations should follow the following convention:

Port names contain a text abbreviation indicating port capabilities followed by a sequential number for the port. The following device name types should be used:

- COM#, where COM is for RS-232 ports and # is a number assigned to the port
- IR#, where IR is for IrDA IRCOMM ports and # is a number assigned to the port

This naming scheme allows API users to generally determine the type of port that they would like to use. For instance, if an application desires to “beam” a piece of data, the app could look for “IR#” ports for opening the connection. The alternative is a trial and error approach with all available ports.

Since: IMP-NG / MIDP 2.0

Member Summary	
Methods	
`int`	`getBaudRate()`
`int`	`setBaudRate(int baudrate)`

Inherited Member Summary
Methods inherited from interface `Connection`
`close()`
Methods inherited from interface `InputConnection`
`openDataInputStream(), openInputStream()`
Methods inherited from interface `OutputConnection`
`openDataOutputStream(), openOutputStream()`

Methods

getBaudRate()

Declaration:
`public int` **`getBaudRate`**`()`

Description:
Gets the baudrate for the serial port connection.

Returns: the baudrate of the connection

See Also: `setBaudRate(int)`

setBaudRate(int)

Declaration:
`public int` **`setBaudRate`**`(int baudrate)`

Description:
Sets the baudrate for the serial port connection. If the requested `baudrate` is not supported on the platform, then the system MAY use an alternate valid setting. The alternate value can be accessed using the `getBaudRate` method.

Parameters:
`baudrate` - the baudrate for the connection

Returns: the previous baudrate of the connection

See Also: `getBaudRate()`

Package
javax.microedition.io

javax.microedition.io
Connector

Declaration

```
public class Connector

Object
     |
     +--javax.microedition.io.Connector
```

Description

Factory class for creating new `Connection` objects.

The creation of connections is performed dynamically by looking up a protocol implementation class whose name is formed from the platform name (read from a system property) and the protocol name of the requested connection (extracted from the parameter string supplied by the application programmer.) The parameter string that describes the target should conform to the URL format as described in RFC 2396. This takes the general form:

`{scheme}:[{target}][{parms}]`

where `{scheme}` is the name of a protocol such as *http*.

The `{target}` is normally some kind of network address.

Any `{parms}` are formed as a series of equates of the form “;x=y”. Example: “;type=a”.

Configuration of OTA connections

While the developer of a MIDlet as defined in MIDP 2.0 usually does not need to know technical details about the network used by the application to be programmed, because it is up to the user of the MIDlet to choose the appropriate network configuration. The situation for an IMlet programmer is different. He is usually the user himself, and therefore has to configure the network by means of the application, because once the application is running there is no possibility to interact. In order to allow configuring OTA connections on an IMlet level the capability of the argument string to the `Connector.open` method to contain named parameters is used. The parameters for this configuration can be used in the argument string for `Connector.open` installing the following connections:

- `HttpConnection`
- `HttpsConnection`
- `SecureConnection`
- `SocketConnection`
- `ServerSocketConnection`
- `UDPDatagramConnection`

Optional Parameters

Parameter	Default	Description
`bearer_type`	`none`	Protocol to be used.
`access_point`	`none`	Access point for GPRS
`username`	`none`	User name
`password`	`none`	Password
`dns`	`none`	DNS
`timeout`	`0 (for "never")`	timeout, when to hang up if no network activities take place

Package javax.microedition.io

BNF Format for `Connector.open()` string

The URI must conform to the BNF syntax specified below. If the URI does not conform to this syntax, an `IllegalArgumentException` is thrown.

`<bearer_type>`	`::= ";`**`bearer_type`**`="<bearer_type_value>`
`<bearer_type_value>`	`::= "GPRS" | "CSD" | …`
`<access_point>`	`::= ";`**`access_point`**`="<access_point_value>`
`<access_point_value>`	`::= <string>`
`<username>`	`::= ";`**`username`**`="<username_value>`
`<username_value>`	`::= 1*<characters allowed in user names>`
`<password>`	`::= ";`**`password`**`="<password_value>`
`<password_value>`	`::= 1*<characters allowed in passwords>`
`<dns>`	`::= ";`**`dns`**`="<dns_value>`[1]
~~`<dns_value>`~~	~~`::= <number>.<number>.<number>.<number>`~~
`<phone_number>`	`::= ";`**`phone_number`**`="<phone_number_value>`
`<phone_number_value>`	`::= <phone_number_digits>`
`<timeout>`	`::= ";`**`timeout`**`="<timeout_value>`
`<timeout_value>`	`::= <int> where 0 means "never"`

Configuration of the Calling Mode

An optional second parameter may be specified to the `open` function. This is a mode flag that indicates to the protocol handler the intentions of the calling code. The options here specify if the connection is going to be read (READ), written (WRITE), or both (READ_WRITE). The validity of these flag settings is protocol dependent. For instance, a connection for a printer would not allow read access, and would throw an `IllegalArgumentException`. If the mode parameter is not specified, READ_WRITE is used by default.

An optional third parameter to the `open` function is a boolean flag that indicates if the calling code can handle timeout exceptions. If this flag is set, the protocol implementation may throw an `InterruptedIOException` when it detects a timeout condition. This flag is only a hint to the protocol handler, and it does not guarantee that such exceptions will actually be thrown. If this parameter is not set, no timeout exceptions will be thrown.

Because connections are frequently opened just to gain access to a specific input or output stream, four convenience functions are provided for this purpose. See also: `DatagramConnection` for information relating to datagram addressing.

Since: CLDC 1.0

Member Summary

Fields

```
static int READ
static int READ_WRITE
static int WRITE
```

Methods

```
static Connection                open(String name)
static Connection                open(String name, int mode)
static Connection                open(String name, int mode, boolean timeouts)
static java.io.DataInputStream   openDataInputStream(String name)
```

[1] The syntax of the DNS value is dependent on what protocol is supported (e.g. IPv4 or IPv6). It is therefore out of scope of this specification.

static java.io.DataOutputStream	openDataOutputStream(String name)
static java.io.InputStream	openInputStream(String name)
static java.io.OutputStream	openOutputStream(String name)

Inherited Member Summary

Methods inherited from class `Object`

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

Fields

READ

Declaration:
`public static final int` **`READ`**

Description:
Access mode READ.
The value 1 is assigned to `READ`.

READ_WRITE

Declaration:
`public static final int` **`READ_WRITE`**

Description:
Access mode READ_WRITE.
The value 3 is assigned to `READ_WRITE`.

WRITE

Declaration:
`public static final int` **`WRITE`**
Description:
Access mode WRITE.
The value 2 is assigned to `WRITE`.

Methods

open(String)

Declaration:
`public static javax.microedition.io.Connection` **`open`**`(String name) throws IOException`

Description:
Create and open a Connection.

Parameters:
`name` - The URL for the connection.

Returns: A new Connection object.

Throws:
`IllegalArgumentException` - If a parameter is invalid.
`ConnectionNotFoundException` - If the requested connection cannot be make, or the protocol type does not exist.
`java.io.IOException` - If some other kind of I/O error occurs.
`SecurityException` - If a requested protocol handler is not permitted.

Package javax.microedition.io

open(String, int)

Declaration:
```
public static javax.microedition.io.Connection open(String name, int mode)
                 throws IOException
```

Description:
Create and open a Connection.

Parameters:
`name` - The URL for the connection.
`mode` - The access mode.

Returns: A new Connection object.

Throws:
`IllegalArgumentException` - If a parameter is invalid.
`ConnectionNotFoundException` - If the requested connection cannot be make, or the protocol type does not exist.
`java.io.IOException` - If some other kind of I/O error occurs.
`SecurityException` - If a requested protocol handler is not permitted.

open(String, int, boolean)

Declaration:
```
public static javax.microedition.io.Connection open(String name, int mode,
                 boolean timeouts)
                 throws IOException
```

Description:
Create and open a Connection.

Parameters:
`name` - The URL for the connection
`mode` - The access mode
`timeouts` - A flag to indicate that the caller wants timeout exceptions

Returns: A new Connection object

Throws:
`IllegalArgumentException` - If a parameter is invalid.
`ConnectionNotFoundException` - if the requested connection cannot be make, or the protocol type does not exist.
`java.io.IOException` - If some other kind of I/O error occurs.
`SecurityException` - If a requested protocol handler is not permitted.

openDataInputStream(String)

Declaration:
```
public static java.io.DataInputStream openDataInputStream(String name)
                 throws IOException
```

Description:
Create and open a connection input stream.

Parameters:
`name` - The URL for the connection.

Returns: A DataInputStream.

Throws:

`IllegalArgumentException` - If a parameter is invalid.
`ConnectionNotFoundException` - If the connection cannot be found.
`java.io.IOException` - If some other kind of I/O error occurs.
`SecurityException` - If access to the requested stream is not permitted.

openDataOutputStream(String)

Declaration:
```
public static java.io.DataOutputStream openDataOutputStream(String name)
                 throws IOException
```

Description:
Create and open a connection output stream.

Parameters:
`name` - The URL for the connection.

Returns: A DataOutputStream.

Throws:

`IllegalArgumentException` - If a parameter is invalid.
`ConnectionNotFoundException` - If the connection cannot be found.
`java.io.IOException` - If some other kind of I/O error occurs.
`SecurityException` - If access to the requested stream is not permitted.

openInputStream(String)

Declaration:
```
public static java.io.InputStream openInputStream(String name)
                 throws IOException
```

Description:
Create and open a connection input stream.

Parameters:
`name` - The URL for the connection.

Returns: An InputStream.

Throws:

`IllegalArgumentException` - If a parameter is invalid.
`ConnectionNotFoundException` - If the connection cannot be found.
`java.io.IOException` - If some other kind of I/O error occurs.
`SecurityException` - If access to the requested stream is not permitted.

openOutputStream(String)

Declaration:
```
public static java.io.OutputStream openOutputStream(String name)
                 throws IOException
```

Description:
Create and open a connection output stream.

Parameters:
`name` - The URL for the connection.

Returns: An OutputStream.

Throws:

`IllegalArgumentException` - If a parameter is invalid.
`ConnectionNotFoundException` - If the connection cannot be found.
`java.io.IOException` - If some other kind of I/O error occurs.
`SecurityException` - If access to the requested stream is not permitted.

javax.microedition.io HttpConnection

Declaration

`public interface` **`HttpConnection extends ContentConnection`**

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

All Known Subinterfaces: `HttpsConnection`

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 request parameters can be set
- Connected, in which request parameters have been sent and the response is expected
- Closed, the final state, in which the HTTP connection as been terminated

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 when the connection is in Setup state.

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

The following methods may be invoked while the connection is in Setup or Connected state.

- `close`

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

After an output stream has been opened by the `openOutputStream` or `openDataOutputStream` methods, attempts to change the request parameters via `setRequestMethod` or the `setRequestProperty` are ignored. Once the request parameters have been sent, these methods will throw an `IOException`. When an output stream is closed via the `OutputStream.close` or `DataOutputStream.close` methods, the connection enters the Connected state. When the output stream is flushed via the `OutputStream.flush` or `DataOutputStream.flush` methods, the request parameters MUST be sent along with any data written to the stream.

The transition to Closed state from any other state is caused by the `close` method and the closing all of the streams that were opened from the connection.

Example using StreamConnection

Simple read of a URL using `StreamConnection`. No HTTP specific behavior is needed or used. (**Note:** this example ignores all HTTP response headers and the HTTP response code. Since a proxy or server may have sent an error response page, an application ca not distinquish which data is retreived in the `InputStream`.) `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;
     DataInputStream is = null;
     try {
              c = (ContentConnection)Connector.open(url);
              int len = (int)c.getLength();
              dis = c.openDataInputStream();
              if (len > 0) {
                       byte[] data = new byte[len];
                       dis.readFully(data);
              } else {
                       int ch;
                       while ((ch = dis.read()) != -1) {
                                ...
                       }
              }
     } finally {
              if (dis != null)
                       dis.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;
     int rc;
     try {
              c = (HttpConnection)Connector.open(url);
              // Getting the response code will open the connection,
              // send the request, and read the HTTP response headers.
              // The headers are stored until requested.
              rc = c.getResponseCode();
              if (rc != HttpConnection.HTTP_OK) {
                       throw new IOException("HTTP response code: " + rc);
              }
              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) {
                       int actual = 0;
                       int bytesread = 0 ;
                       byte[] data = new byte[len];
                       while ((bytesread != len) && (actual != -1)) {
                                actual = is.read(data, bytesread, len - bytesread);
                                bytesread += actual;
                       }
              } else {
                       int ch;
                       while ((ch = is.read()) != -1) {
                                ...
                       }
              }
     } catch (ClassCastException e) {
              throw IllegalArgumentException("Not an HTTP URL");
     } 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;
    int rc;
    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/IMP-NG 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, getResponseCode will flush
        // Getting the response code will open the connection,
        // send the request, and read the HTTP response headers.
        // The headers are stored until requested.
        rc = c.getResponseCode();
        if (rc != HttpConnection.HTTP_OK) {
            throw new IOException("HTTP response code: " + rc);
        }
        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) {
            int actual = 0;
            int bytesread = 0 ;
            byte[] data = new byte[len];
            while ((bytesread != len) && (actual != -1)) {
                actual = is.read(data, bytesread, len - bytesread);
                bytesread += actual;
            }
            process(data);
        } else {
            int ch;
            while ((ch = is.read()) != -1) {
                process((byte)ch);
            }
        }
    } catch (ClassCastException e) {
        throw IllegalArgumentException("Not an HTTP URL");
    } 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 openInputStream(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`

Since: IMP / MIDP 1.0

Member Summary

Fields

static java.lang.String	GET
static java.lang.String	HEAD
static int	HTTP_ACCEPTED
static int	HTTP_BAD_GATEWAY
static int	HTTP_BAD_METHOD
static int	HTTP_BAD_REQUEST
static int	HTTP_CLIENT_TIMEOUT
static int	HTTP_CONFLICT
static int	HTTP_CREATED
static int	HTTP_ENTITY_TOO_LARGE
static int	HTTP_EXPECT_FAILED
static int	HTTP_FORBIDDEN
static int	HTTP_GATEWAY_TIMEOUT

Member Summary

static int	HTTP_GONE
static int	HTTP_INTERNAL_ERROR
static int	HTTP_LENGTH_REQUIRED
static int	HTTP_MOVED_PERM
static int	HTTP_MOVED_TEMP
static int	HTTP_MULT_CHOICE
static int	HTTP_NO_CONTENT
static int	HTTP_NOT_ACCEPTABLE
static int	HTTP_NOT_AUTHORITATIVE
static int	HTTP_NOT_FOUND
static int	HTTP_NOT_IMPLEMENTED
static int	HTTP_NOT_MODIFIED
static int	HTTP_OK
static int	HTTP_PARTIAL
static int	HTTP_PAYMENT_REQUIRED
static int	HTTP_PRECON_FAILED
static int	HTTP_PROXY_AUTH
static int	HTTP_REQ_TOO_LONG
static int	HTTP_RESET
static int	HTTP_SEE_OTHER
static int	HTTP_TEMP_REDIRECT
static int	HTTP_UNAUTHORIZED
static int	HTTP_UNAVAILABLE
static int	HTTP_UNSUPPORTED_RANGE
static int	HTTP_UNSUPPORTED_TYPE
static int	HTTP_USE_PROXY
static int	HTTP_VERSION
static java.lang.String	POST

Methods

long	getDate()
long	getExpiration()
java.lang.String	getFile()
java.lang.String	getHeaderField(int n)
java.lang.String	getHeaderField(String name)
long	getHeaderFieldDate(String name, long def)
int	getHeaderFieldInt(String name, int def)
java.lang.String	getHeaderFieldKey(int n)
java.lang.String	getHost()
long	getLastModified()
int	getPort()
java.lang.String	getProtocol()
java.lang.String	getQuery()
java.lang.String	getRef()
java.lang.String	getRequestMethod()
java.lang.String	getRequestProperty(String key)
int	getResponseCode()
java.lang.String	getResponseMessage()
java.lang.String	getURL()
void	setRequestMethod(String method)
void	setRequestProperty(String key, String value)

Inherited Member Summary

Methods inherited from interface `Connection`

`close()`

Methods inherited from interface `ContentConnection`

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

Methods inherited from interface `InputConnection`

`openDataInputStream(), openInputStream()`

Methods inherited from interface `OutputConnection`

`openDataOutputStream(), openOutputStream()`

Fields

GET

Declaration:
`public static final String` **`GET`**

Description:
HTTP Get method.

HEAD

Declaration:
`public static final String` **`HEAD`**

Description:
HTTP Head method.

HTTP_ACCEPTED

Declaration:
`public static final int` **`HTTP_ACCEPTED`**

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

HTTP_BAD_GATEWAY

Declaration:
`public static final int` **`HTTP_BAD_GATEWAY`**

Description:
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

Declaration:
`public static final int` **`HTTP_BAD_METHOD`**

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

HTTP_BAD_REQUEST

Declaration:
`public static final int` **`HTTP_BAD_REQUEST`**

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

HTTP_CLIENT_TIMEOUT

Declaration:
`public static final int` **`HTTP_CLIENT_TIMEOUT`**

Description:
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

Declaration:
`public static final int` **`HTTP_CONFLICT`**

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

HTTP_CREATED

Declaration:
`public static final int` **`HTTP_CREATED`**

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

HTTP_ENTITY_TOO_LARGE

Declaration:
`public static final int` **`HTTP_ENTITY_TOO_LARGE`**

Description:
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

Declaration:
`public static final int` **`HTTP_EXPECT_FAILED`**

Description:
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

Declaration:
`public static final int` **`HTTP_FORBIDDEN`**

Description:
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

Declaration:
`public static final int` **`HTTP_GATEWAY_TIMEOUT`**

Description:
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

Declaration:
`public static final int` **`HTTP_GONE`**

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

HTTP_INTERNAL_ERROR

Declaration:
`public static final int` **`HTTP_INTERNAL_ERROR`**

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

HTTP_LENGTH_REQUIRED

Declaration:
`public static final int` **`HTTP_LENGTH_REQUIRED`**

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

HTTP_MOVED_PERM

Declaration:
`public static final int` **`HTTP_MOVED_PERM`**

Description:
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

Declaration:
`public static final int` **`HTTP_MOVED_TEMP`**

Description:
302: The requested resource resides temporarily under a different URI. (Note: the name of this status code reflects the earlier publication of RFC2068, which was changed in RFC2616 from “moved temporalily” to “found”. The semantics were not changed. The `Location` header indicates where the application should resend the request.)

HTTP_MULT_CHOICE

Declaration:
`public static final int` **`HTTP_MULT_CHOICE`**

Description:
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

Declaration:
`public static final int` **`HTTP_NO_CONTENT`**

Description:
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

Declaration:
`public static final int` **`HTTP_NOT_ACCEPTABLE`**

Description:
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

Declaration:
`public static final int` **`HTTP_NOT_AUTHORITATIVE`**

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

HTTP_NOT_FOUND

Declaration:
`public static final int` **`HTTP_NOT_FOUND`**

Description:
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

Declaration:
`public static final int` **`HTTP_NOT_IMPLEMENTED`**

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

HTTP_NOT_MODIFIED

Declaration:
`public static final int` **`HTTP_NOT_MODIFIED`**

Description:
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

Declaration:
`public static final int` **`HTTP_OK`**

Description:
200: The request has succeeded.

HTTP_PARTIAL

Declaration:
`public static final int` **`HTTP_PARTIAL`**

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

HTTP_PAYMENT_REQUIRED

Declaration:
`public static final int` **`HTTP_PAYMENT_REQUIRED`**

Description:
402: This code is reserved for future use.

HTTP_PRECON_FAILED

Declaration:
`public static final int` **`HTTP_PRECON_FAILED`**

Description:
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

Declaration:
`public static final int` **`HTTP_PROXY_AUTH`**

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

HTTP_REQ_TOO_LONG

Declaration:
`public static final int` **`HTTP_REQ_TOO_LONG`**

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

HTTP_RESET

Declaration:
`public static final int` **`HTTP_RESET`**

Description:
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

Declaration:
`public static final int` **`HTTP_SEE_OTHER`**

Description:
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

Declaration:
`public static final int` **`HTTP_TEMP_REDIRECT`**

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

HTTP_UNAUTHORIZED

Declaration:
`public static final int` **`HTTP_UNAUTHORIZED`**

Description:
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

Declaration:
`public static final int HTTP_UNAVAILABLE`

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

HTTP_UNSUPPORTED_RANGE

Declaration:
`public static final int HTTP_UNSUPPORTED_RANGE`

Description:
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

Declaration:
`public static final int HTTP_UNSUPPORTED_TYPE`

Description:
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

Declaration:
`public static final int HTTP_USE_PROXY`

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

HTTP_VERSION

Declaration:
`public static final int HTTP_VERSION`

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

POST

Declaration:
`public static final String POST`

Description:
HTTP Post method.

Methods

getDate()

Declaration:
`public long getDate() throws IOException`

Description:
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:
`java.io.IOException` - if an error occurred connecting to the server.

getExpiration()

Declaration:
`public long getExpiration() throws IOException`

Description:
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:
`java.io.IOException` - if an error occurred connecting to the server.

getFile()

Declaration:
`public String getFile()`

Description:
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)

Declaration:
`public String getHeaderField(int n) throws IOException`

Description:
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:
`java.io.IOException` - if an error occurred connecting to the server.

getHeaderField(String)

Declaration:
`public String getHeaderField(String name) throws IOException`

Description:
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:
`java.io.IOException` - if an error occurred connecting to the server.

getHeaderFieldDate(String, long)

Declaration:
`public long getHeaderFieldDate(String name, long def) throws IOException`

Description:
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:
`java.io.IOException` - if an error occurred connecting to the server.

getHeaderFieldInt(String, int)

Declaration:
`public int getHeaderFieldInt(String name, int def) throws IOException`

Description:
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:
`java.io.IOException` - if an error occurred connecting to the server.

getHeaderFieldKey(int)

Declaration:
`public String getHeaderFieldKey(int n) throws IOException`

Description:
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:

`java.io.IOException` - if an error occurred connecting to the server.

getHost()

Declaration:
`public String getHost()`

Description:
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`.

getLastModified()

Declaration:
`public long getLastModified() throws IOException`

Description:
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:

`java.io.IOException` - if an error occurred connecting to the server.

getPort()

Declaration:
`public int getPort()`

Description:
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()

Declaration:
`public String getProtocol()`

Description:
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()

Declaration:
`public String getQuery()`

Description:
Returns the query portion of the URL of this `HttpConnection`. RFC2396 defines the query component as the text after the first 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()

Declaration:
`public String getRef()`

Description:
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()

Declaration:
`public String getRequestMethod()`

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

Returns: the HTTP request method

See Also: `setRequestMethod(String)`

getRequestProperty(String)

Declaration:
`public String getRequestProperty(String key)`

Description:
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.

See Also: `setRequestProperty(String, String)`

getResponseCode()

Declaration:
`public int getResponseCode() throws IOException`

Description:
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:
`java.io.IOException` - if an error occurred connecting to the server.

getResponseMessage()

Declaration:
`public String getResponseMessage() throws IOException`

Description:
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:
`java.io.IOException` - if an error occurred connecting to the server.

getURL()

Declaration:
`public String getURL()`

Description:
Return a string representation of the URL for this connection.

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

setRequestMethod(String)

Declaration:
`public void setRequestMethod(String method) throws IOException`

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

See Also: `getRequestMethod()`

setRequestProperty(String, String)

Declaration:
`public void setRequestProperty(String key, String value) throws IOException`

Description:
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:

`java.io.IOException` - is thrown if the connection is in the connected state.

See Also: `getRequestProperty(String)`

javax.microedition.io
HttpsConnection

Declaration

`public interface` **`HttpsConnection extends HttpConnection`**

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

Description

This interface defines the necessary methods and constants to establish a secure network connection. The URI format with scheme `https` when passed to `Connector.open` will return an `HttpsConnection`. RFC 2818 (http://www.ietf.org/rfc/rfc2818.txt) defines the scheme.

A secure connection MUST be implemented by one or more of the following specifications:

- HTTP over TLS as documented in RFC 2818 (http://www.ietf.org/rfc/rfc2818.txt) and TLS Protocol Version 1.0 as specified in RFC 2246 (http://www.ietf.org/rfc/rfc2246.txt).
- SSL V3 as specified in The SSL Protocol Version 3.0 (http://home.netscape.com/eng/ssl3/draft302.txt)
- WTLS as specified in WAP Forum Specifications June 2000 (WAP 1.2.1) conformance release (http://www.wapforum.org/what/technical_1_2_1.htm) - Wireless Transport Layer Security document WAP-199.
- WAP(TM) TLS Profile and Tunneling Specification as specified in WAP-219-TLS-20010411-a (http://www.wapforum.com/what/technical.htm)

HTTPS is the secure version of HTTP (IETF RFC2616), a request-response protocol in which the parameters of the request must be set before the request is sent.

In addition to the normal `IOExceptions` that may occur during invocation of the various methods that cause a transition to the Connected state, `CertificateException` (a subtype of `IOException`) may be thrown to indicate various failures related to establishing the secure link. The secure link is necessary in the `Connected` state so the headers can be sent securely. The secure link may be established as early as the invocation of `Connector.open()` and related methods for opening input and output streams and failure related to certificate exceptions may be reported.

Example

Open an HTTPS connection, set its parameters, then read the HTTP response.

`Connector.open` is used to open the URL and an `HttpsConnection` 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 `HttpsConnection` 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 getViaHttpsConnection(String url) throws CertificateException, IOException {
    HttpsConnection c = null;
    InputStream is = null;
    try {
            c = (HttpsConnection)Connector.open(url);
            // Getting the InputStream ensures that the connection
            // is opened (if it was not already handled by
            // Connector.open()) and the SSL handshake is exchanged,
            // and the HTTP response headers are read.
            // These are stored until requested.
            is = c.openDataInputStream();
            if c.getResponseCode() == HttpConnection.HTTP_OK) {
                    // Get the length and process the data
                    int len = (int)c.getLength();
                    if (len > 0) {
                            byte[] data = new byte[len];
                            int actual = is.readFully(data);
                            ...
                    } else {
                            int ch;
                            while ((ch = is.read()) != -1) {
                                    ...
                            }
                    }
            } else {
                    ...
            }
    } finally {
            if (is != null)
                    is.close();
            if (c != null)
                    c.close();
    }
}
```

Since: IMP-NG / MIDP 2.0

See Also: `javax.microedition.pki.CertificateException`

Member Summary

Methods

`int`	`getPort()`
`SecurityInfo`	`getSecurityInfo()`

Inherited Member Summary

Fields inherited from interface `HttpConnection`

`GET, HEAD, HTTP_ACCEPTED, HTTP_BAD_GATEWAY, HTTP_BAD_METHOD, HTTP_BAD_REQUEST, HTTP_CLIENT_TIMEOUT, HTTP_CONFLICT, HTTP_CREATED, HTTP_ENTITY_TOO_LARGE, HTTP_EXPECT_FAILED, HTTP_FORBIDDEN, HTTP_GATEWAY_TIMEOUT, HTTP_GONE, HTTP_INTERNAL_ERROR, HTTP_LENGTH_REQUIRED, HTTP_MOVED_PERM, HTTP_MOVED_TEMP, HTTP_MULT_CHOICE, HTTP_NOT_ACCEPTABLE, HTTP_NOT_AUTHORITATIVE, HTTP_NOT_FOUND, HTTP_NOT_IMPLEMENTED, HTTP_NOT_MODIFIED, HTTP_NO_CONTENT, HTTP_OK, HTTP_PARTIAL, HTTP_PAYMENT_REQUIRED, HTTP_PRECON_FAILED, HTTP_PROXY_AUTH, HTTP_REQ_TOO_LONG, HTTP_RESET, HTTP_SEE_OTHER, HTTP_TEMP_REDIRECT, HTTP_UNAUTHORIZED, HTTP_UNAVAILABLE, HTTP_UNSUPPORTED_RANGE, HTTP_UNSUPPORTED_TYPE, HTTP_USE_PROXY, HTTP_VERSION, POST`

Methods inherited from interface `Connection`

`close()`

Methods inherited from interface `ContentConnection`

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

Methods inherited from interface `HttpConnection`

`getDate(), getExpiration(), getFile(), getHeaderField(int), getHeaderField(int), getHeaderFieldDate(String, long), getHeaderFieldInt(String, int), getHeaderFieldKey(int), getHost(), getLastModified(), getProtocol(), getQuery(), getRef(), getRequestMethod(), getRequestProperty(String), getResponseCode(), getResponseMessage(), getURL(), setRequestMethod(String), setRequestProperty(String, String)`

Methods inherited from interface `InputConnection`

`openDataInputStream(), openInputStream()`

Methods inherited from interface `OutputConnection`

`openDataOutputStream(), openOutputStream()`

Methods

getPort()

Declaration:
`public int getPort()`

Description:
Returns the network port number of the URL for this `HttpsConnection`.

Overrides: `getPort` in interface `HttpConnection`

Returns: the network port number of the URL for this `HttpsConnection`. The default HTTPS port number (443) is returned if there was no port number in the string passed to `Connector.open`.

getSecurityInfo()

Declaration:
`public javax.microedition.io.SecurityInfo getSecurityInfo() throws IOException`

Description:
Return the security information associated with this successfully opened connection. If the connection is still in `Setup` state then the connection is initiated to establish the secure connection to the server. The method returns when the connection is established and the `Certificate` supplied by the server has been validated. The `SecurityInfo` is only returned if the connection has been successfully made to the server.

Returns: the security information associated with this open connection.

Throws:
`java.io.IOException` - if an arbitrary connection failure occurs

javax.microedition.io PushRegistry

Declaration

```
public class PushRegistry

Object
  |
  +--javax.microedition.io.PushRegistry
```

Description

The `PushRegistry` maintains a list of inbound connections. An application can register the inbound connections with an entry in the application descriptor file or dynamically by calling the `registerConnection` method.

While an application is running, it is responsible for all I/O operations associated with the inbound connection. When the application is not running, the application management software (AMS) listens for inbound notification requests. When a notification arrives for a registered IMlet, the AMS will start the IMlet via the normal invocation of `MIDlet.startApp` method.

Installation Handling of Declared Connections

To avoid collisions on inbound generic connections, the application descriptor file MUST include information about static connections that are needed by the IMlet suite. If all the static Push declarations in the application descriptor cannot be fulfilled during the installation, the IMlet suite MUST NOT be installed. (See chapter ***Error! Reference source not found.*. *Error! Reference source not found.*** for errors reported in the event of conflicts.) Conditions when the declarations can not be fulfilled include:

- syntax errors in the Push attributes,
- declaration for a connection end point (e.g. port number) that is already reserved in the device,
- declaration for a protocol that is not supported for Push in the device,
- declaration referencing an IMlet class that is not valid and
- declaration referencing an IMlet class that is not listed in the `MIDlet-<n>` attributes of the same application descriptor.

If the IMlet suite can function meaningfully even if a Push registration can't be fulfilled, it MUST register the Push connections using the dynamic registration methods in the `PushRegistry`.

A conflict-free installation reserves each requested connection for the exclusive use of the IMlets in the suite. While the suite is installed, any attempt by other applications to open one of the reserved connections will fail with a `ConnectionNotFoundException`. A call from an IMlet to `Connector.open()` on a connection reserved for its suite will always succeed, assuming the suite does not already have the connection open.

If two IMlet suites have a static push connection in common, they cannot be installed together and both function correctly. The user would typically have to uninstall one before being able to successfully install the other.

Push Registration Attribute

Each push registration entry contains the following information :

`MIDlet-Push-``<n>: <ConnectionURL>, <MIDletClassName>, <AllowedSender>`

where :

- `MIDlet-Push-<n>` = the Push registration attribute name. Multiple push registrations can be provided in an IMlet suite. The numeric value for `<n>` starts from `1` and MUST use consecutive ordinal numbers for additional entries. The first missing entry terminates the list. Any additional entries are ignored

- `ConnectionURL` = the connection string used in `Connector.open()`
- `MIDletClassName` = the IMlet that is responsible for the connection. The named IMlet MUST be registered in the descriptor file or the jar file manifest with a `MIDlet-<n>` record. (This information is needed when displaying messages to the user about the application when push connections are detected, or when the user grants/revokes priveleges for the application.) If the named IMlet appears more than once in the suite, the first matching entry is used.
- `AllowedSender` = a designated filter that restricts which senders are valid for launching the requested IMlet. The syntax and semantics of the `AllowedSender` field depend on the addressing format used for the protocol. However, every syntax for this field MUST support using the wildcard characters `"*"` and `"?"`. The semantics of those wildcard are:
  - `"*"` matches any string, including an empty string
  - `"?"` matches any single character

When the value of this field is just the wildcard character `"*"`, connections will be accepted from any originating source. For Push attributes using the `datagram` and `socket` URLs (if supported by the platform), this field contains a numeric IP address in the same format for IPv4 and IPv6 as used in the respective URLs (IPv6 address including the square brackets as in the URL). It is possible to use the wildcards also in these IP addresses, e.g. `"129.70.40.*"` would allow subnet resolution. Note that the port number is not part of the filter for `datagram` and `socket` connections.

The IMP-NG specification defines the syntax for `datagram` and `socket` inbound connections. When other specifications define push semantics for additional connection types, they must define the expected syntax for the filter field, as well as the expected format for the connection URL string.

Example Descriptor File Declarative Notation

The following is a sample descriptor file entry that would reserve a stream socket at port 79 and a datagram connection at port 50000. (Port numbers are maintained by IANA and cover well-known, user-registered and dynamic port numbers) [See IANA Port Number Registry (http://www.iana.org/numbers.html#P)]

```
MIDlet-Push-1: socket://:79, com.siemens.Sample, *
MIDlet-Push-2: datagram://:50000, com.siemens.Sample, *
```

Buffered Messages

The requirements for buffering of messages are specific to each protocol used for Push and are defined separately for each protocol. There is no general requirement related to buffering that would apply to all protocols. If the implementation buffers messages, these messages MUST be provided to the IMlet when the IMlet is started and it opens the related `Connection` that it has registered for Push.

When datagram connections are supported with Push, the implementation MUST guarantee that when an IMlet registered for datagram Push is started in response to an incoming datagram, at least the datagram that caused the startup of the IMlet is buffered by the implementation and will be available to the IMlet when the IMlet opens the `UDPDatagramConnection` after startup.

When socket connections are supported with Push, the implementation MUST guarantee that when an IMlet registered for socket Push is started in response to an incoming socket connection, this connection can be accepted by the IMlet by opening the `ServerSocketConnection` after startup, provided that the connection hasn't timed out meanwhile.

Connection vs Push Registration Support

Not all generic connections will be appropriate for use as push application transport. Even if a protocol is supported on the device as an inbound connection type, it is not required to be enabled as a valid push mechanism. e.g. a platform might support server socket connections in an IMlet, but might not support inbound socket connections for push launch capability. A `ConnectionNotFoundException` is thrown from the `registerConnection` and from the `registerAlarm` methods, when the platform does not support that optional capability.

AMS Connection Handoff

Responsibility for registered push connections is shared between the AMS and the IMlet that handles the I/O operations on the inbound connection. To prevent any data from being lost, an application is responsible for all I/O operations on the connection from the time it calls `Connector.open()` until it calls `Connection.close()`.

The AMS listens for inbound connection notifications. This MAY be handled via a native callback or polling mechanism looking for new inbound data. The AMS is responsible for enforcing the security of PushRegistry before invoking the IMlet suite.

The AMS is responsible for the shutdown of any running applications (if necessary) prior to the invocation of the push IMlet method.

After the AMS has started the push application, the IMlet is responsible for opening the connections and for all subsequent I/O operations. An application that needs to perform blocking I/O operations SHOULD use a separate thread to allow for interactive user operations. Once the application has been started and the connection has been opened, the AMS is no longer responsible for listening for push notifications for that connection. The application is responsible for reading all inbound data.

If an application has finished with all inbound data it MAY `close()` the connection. If the connection is closed, then neither the AMS nor the application will be listening for push notifications. Inbound data could be lost, if the application closes the connection before all data has been received.

When the application is destroyed, the AMS resumes its responsiblity to watch for inbound connections.

Dynamic Connections Registered from a Running IMlet

There are cases when defining a well known port registered with IANA is not necessary. Simple applications may just wish to exchange data using a private protocol between an IMlet and server application.

To accomodate this type of application, a mechanism is provided to dynamically allocate a connection and to register that information, as if it was known, when the application was installed. This information can then be sent to an agent on the network to use as the mechanism to communicate with the registered IMlet.

For instance, if a `UDPDatagramConnection` is opened and a port number, was not specified, then the application is requesting a dynamic port to be allocated from the ports that are currently available. By calling `PushRegistry.registerConnection()` the IMlet informs the AMS that it is the target for inbound communication, even after the IMlet has been destroyed (See IMlet life cycle for definition of “destroyed” state). If the application is deleted from the IM, then its dynamic communication connections are unregistered automatically.

AMS Runtime Handling - Implementation Notes

During installation each IMlet that is expecting inbound communication on a well-known address has the information recorded with the AMS from the push registration attribute in the manifest or application descriptor file. Once the installation has been successfully completed, (e.g. for the OTA recommended practices – when the *Installation notification message* has been successfully transmitted, the application is officially installed.) the IMlet MAY then receive inbound communication. e.g. the push notification event.

When the AMS is started, it checks the list of registered connections and begins listening for inbound communication. When a notification arrives the AMS starts the registered IMlet. The IMlet then opens the connection with `Connector.open()` method to perform whatever I/O operations are needed for the particular connection type. e.g. for a server socket the application uses `acceptAndOpen()` to get the socket connected and for a datagram connection the application uses `receive()` to read the delivered message.

For message-oriented transports the inbound message MAY be read by the AMS and saved for delivery to the IMlet when it requests to read the data. For stream oriented transports the connection MAY be lost if the connection is not accepted before the server end of the connection request timeouts.

When an IMlet is started in response to a registered push connection notification, it is platform dependent what happens to the current running application. The IMlet life cycle defines the expected behaviors that an interrupted IMlet could see from a call to `pauseApp()` or from `destroyApp()`.

Sample Usage Scenarios

Usage scenario 1: The suite includes an IMlet with a well-known port for communication. During the `startApp` processing a thread is launched to handle the incoming data. Using a separate thread is the recommended practice for avoiding conflicts between blocking I/O operations and the normal user interaction events. The thread continues to receive messages until the IMlet is destroyed.

Sample Descriptor File -

In this sample, the descriptor file includes a static push connection registration. It also includes an indication that this IMlet requires permission to use a datagram connection for inbound push messages. (See Security of Push Functions in the package overview for details about IMlet permissions.) **Note:** this sample is appropriate for bursts of datagrams. It is written to loop on the connection, processing received messages.

```
MIDlet-Name: Incoming Message Handler Demo
MIDlet-Version: 1.0
MIDlet-Vendor: Siemens Wireless Modules
MIDlet-Description: Network demonstration programs for IMP-NG
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.0
MIDlet-1: MessageProcessor, , com.siemens.MessageProcessor
MIDlet-Push-1: datagram://:79, com.Siemens.MessageProcessor, *
MIDlet-Permissions: javax.microedition.io.PushRegistry, \\
   javax.microedition.io.Connector.datagramreceiver
```

Sample Message Handler IMlet Processing -

```
public class MessageProcessor extends MIDlet {
    // Current inbound message connection.
    DatagramConnection conn;
    // Flag to terminate the message reading thread.
    boolean done_reading;
    public void startApp() {
        // List of active connections.
        String connections[];
        // Check to see if this session was started due to
        // inbound connection notification.
        connections = PushRegistry.listConnections(true);
        // Start an inbound message thread for available
        // inbound messages for the statically configured
        // connection in the descriptor file.
        for (int i=0; i < connections.length; i++) {
            Thread t = new Thread (new MessageHandler(connections[i]));
            t.start();
        }
        ...
    }
    // Stop reading inbound messages and release the push
    // connection to the AMS listener.
    public void destroyApp(boolean conditional) {
        done_reading = true;
        if (conn != null)
            conn.close();
        // Optionally, notify network service that we're
        // done with the current session.
        ...
    }
    // Optionally, notify network service.
    public void pauseApp() {
        ...
    }
    // Inner class to handle inbound messages on a separate thread.
    class MessageHandler implements Runnable {
        String connUrl ;
        MessageHandler(String url) {
            connUrl = url ;
        }
        // Fetch messages in a blocking receive loop.
        public void run() {
            try {
                // Get a connection handle for inbound messages
                // and a buffer to hold the inbound message.
                DatagramConnection conn = (DatagramConnection)Connector.open(connUrl);
                Datagram data = conn.newDatagram(conn.getMaximumLength());
                // Read the inbound messages
                while (!done_reading) {
                     conn.receive(data);
                     ...
                }
            } catch (IOException ioe) {
                ...
            }
            ...
```

Usage scenario 2: The suite includes an IMlet that dynamically allocates port the first time it is started.

Sample Ping Descriptor File -

In this sample, the descriptor file includes an entry indicating that the application will need permission to use the datagram connection for inbound push messages. The dynamic connection is allocated in the constructor the first time it is run. The open connection is used during this session and can be reopened in a subsequent session in response to a inbound connection notification.

```
MIDlet-Name: Ping Demo
MIDlet-Version: 1.0
MIDlet-Vendor: Siemens Wireless Modules
MIDlet-Description: Network demonstration programs for IMP-NG
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.0
MIDlet-1: JustCallMe, , com.siemens.SamplePing, *
MIDlet-Permissions: javax.microedition.io.PushRegistry, \\
   javax.microedition.io.Connector.datagramreceiver
```

Sample Ping MIDlet Processing -

```
public class SamplePing extends MIDlet {
    // Name of the current application for push registration.
    String myName = "com.siemens.SamplePing";
    // List of registered push connections.
    String connections[];
    // Inbound datagram connection
    UDPDatagramConnection dconn;
    public SamplePing() {
        // Check to see if the ping connection has been registered.
        // This is a dynamic connection allocated on first
        // time execution of this IMlet.
        connections = PushRegistry.listConnections(false);
        if (connections.length == 0) {
            // Request a dynamic port for out-of-band notices.
            // (Omitting the port number let's the system allocate
            // an available port number.)
            try {
                dconn = (UDPDatagramConnection)
                Connector.open("datagram:// ");
                String dport = "datagram://: " + dconn.getLocalPort();
                // Register the port so the IMlet will wake up, if messages
                // are posted after the IMlet exits.
                PushRegistry.registerConnection(dport, myName, "*");
                // Post my datagram address to the network
                ...
            } catch (IOException ioe) {
                ...
            } catch (ClassNotFoundException cnfe) {
                ...
            }
        ...
    }
    public void startApp() {
        // Open the connection if it's not already open.
        if (dconn == null) {
            // This is not the first time this is run, because the
            // dconn hasn't been opened by the constructor.
            // Check if the startup has been due to an incoming
            // datagram.
            connections = PushRegistry.listConnections(true);
            if (connections.length > 0) {
                // There is a pending datagram that can be received.
                dconn = Connector.open(connections[0]);
                // Read the datagram
                Datagram d = dconn.newDatagram(dconn.getMaximumLength());
                dconn.receive(d);
            } else {
                // There are not any pending datagrams, but open
                // the connection for later use.
                connections = PushRegistry.listConnections(false);
                if (connections.length > 0) {
                    dconn = (UDPDatagramConnection)Connector.open(connections[0]);
                }
            }
        ...
    }
    public void destroyApp(boolean unconditional) {
        // Close the connection before exiting
        if(dconn != null){
            dconn.close()
            dconn = null
        }
    }
    ...
```

Since: IMP-NG / MIDP 2.0

Member Summary

Methods

static java.lang.String	getFilter(String connection)
static java.lang.String	getMIDlet(String connection)
static java.lang.String[]	listConnections(boolean available)
static long	registerAlarm(String midlet, long time)
static void	registerConnection(String connection, String midlet, String filter)
static Boolean	unregisterConnection(String connection)

Inherited Member Summary

Methods inherited from class `Object`

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

Methods

getFilter(String)

Declaration:
`public static String getFilter(String connection)`

Description:
Retrieve the registered filter for a requested connection.

Parameters:
`connection` - generic connection *protocol*, *host* and *port number* (optional parameters may be included separated with semi-colons (;))

Returns: a filter string indicating which senders are allowed to cause the IMlet to be launched or `null`, if the connection was not registered by the current IMlet suite or if the connection argument was `null`

See Also: `registerConnection(String, String, String)`

getMIDlet(String)

Declaration:
`public static String getMIDlet(String connection)`

Description:
Retrieve the registered IMlet for a requested connection. The name of the method is `getMIDlet` just for the fact that IMP-NG is a subset of MIDP 2.0, and an IMlet is technically a MIDlet, after all.

Parameters:
`connection` - generic connection *protocol*, *host* and *port number* (optional parameters may be included separated with semi-colons (;))

Returns: class name of the IMlet to be launched, when new external data is available, or `null` if the connection was not registered by the current IMlet suite or if the connection argument was `null`

See Also: `registerConnection(String, String, String)`

listConnections(boolean)

Declaration:
`public static String[] listConnections(boolean available)`

Description:
Return a list of registered connections for the current IMlet suite.

Parameters:
`available` - if `true`, only return the list of connections with input available, otherwise return the complete list of registered connections for the current IMlet suite

Returns: array of registered connection strings, where each connection is represented by the generic connection *protocol*, *host* and *port number* identification

registerAlarm(String, long)

Declaration:
```
public static long registerAlarm(String midlet, long time)
        throws ClassNotFoundException, ConnectionNotFoundException
```

Description:
Register a time to launch the specified application. The `PushRegistry` supports one outstanding wake up time per IMlet in the current suite. An application is expected to use a `TimerTask` for notification of time based events while the application is running.
If a wakeup time is already registered, the previous value will be returned, otherwise a zero is returned the first time the alarm is registered.

Parameters:

`midlet` - class name of the IMlet within the current running IMlet suite to be launched, when the alarm time has been reached. The named IMlet MUST be registered in the descriptor file or the jar file manifest with a `MIDlet-<n>` record. This parameter has the same semantics as the `MIDletClassName` in the Push registration attribute defined above in the class description.
`time` - time at which the IMlet is to be executed in the format returned by `Date.getTime()`

Returns: the time at which the most recent execution of this IMlet was scheduled to occur, in the format returned by `Date.getTime()`

Throws:

`ConnectionNotFoundException` - if the runtime system does not support alarm based application launch
`ClassNotFoundException` - if the IMlet class name can not be found in the current IMlet suite or if this class is not included in any of the `MIDlet-<n>` records in the descriptor file or the jar file manifest or if the `midlet` argument is `null`
`SecurityException` - if the IMlet does not have permission to register an alarm

See Also: `java.util.Date.getTime()`, `java.util.Timer`, `java.util.TimerTask`

registerConnection(String, String, String)

Declaration:
```
public static void registerConnection(String connection, String midlet, String filter)
          throws ClassNotFoundException, IOException
```

Description:
Register a dynamic connection with the application management software. Once registered, the dynamic connection acts just like a connection preallocated from the descriptor file.
The arguments for the dynamic connection registration are the same as the Push Registration Attribute used for static registrations.
If the `connection` or `filter` arguments are `null`, then an `IllegalArgumentException` will be thrown.
If the `midlet` argument is `null` a `ClassNotFoundException` will be thrown.

Parameters:

`connection` - generic connection *protocol*, *host* and *port number* (optional parameters may be included separated with semi-colons (;))
`midlet` - class name of the IMlet to be launched, when new external data is available. The named IMlet MUST be registered in the descriptor file or the jar file manifest with a `MIDlet-<n>` record. This parameter has the same semantics as the `MIDletClassName` in the Push registration attribute defined above in the class description.
`filter` - a connection URL string indicating which senders are allowed to cause the IMlet to be launched

Throws:

`IllegalArgumentException` - if the connection string is not valid, or if the filter string is not valid
`ConnectionNotFoundException` - if the runtime system does not support push delivery for the requested connection protocol
`java.io.IOException` - if the connection is already registered or if there are insufficient resources to handle the registration request
`ClassNotFoundException` - if the IMlet class name can not be found in the current IMlet suite or if this class is not included in any of the `MIDlet-<n>` records in the descriptor file or the jar file manifest
`SecurityException` - if the IMlet does not have permission to register a connection

See Also: `unregisterConnection(String)`

unregisterConnection(String)

Declaration:
`public static boolean **unregisterConnection**(String connection)`

Description:
Remove a dynamic connection registration.

Parameters:
`connection` - generic connection *protocol*, *host* and *port number*

Returns: `true` if the unregistration was successful, `false` if the connection was not registered or if the connection argument was `null`

Throws:
`SecurityException` - if the connection was registered by another IMlet suite

See Also: `registerConnection(String, String, String)`

javax.microedition.io SecureConnection

Declaration

`public interface` **`SecureConnection extends SocketConnection`**

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

Description

This interface defines the secure socket stream connection. A secure connection is established using `Connector.open` with the scheme `"ssl"` and the secure connection is established before `open` returns. If the secure connection cannot be established due to errors related to certificates a `CertificateException` is thrown.

A secure socket is accessed using a generic connection string with an explicit host and port number. The host may be specified as a fully qualified host name or IPv4 number. e.g. `ssl://host.com:79` defines a target socket on the `host.com` system at port `79`.

Note that RFC1900 recommends the use of names rather than IP numbers for best results in the event of IP number reassignment.

A secure connection MUST be implemented by one or more of the following specifications:

- TLS Protocol Version 1.0 as specified in RFC 2246 (http://www.ietf.org/rfc/rfc2246.txt).
- SSL V3 as specified in The SSL Protocol Version 3.0 (http://home.netscape.com/eng/ssl3/draft302.txt)
- WAP(TM) TLS Profile and Tunneling Specification as specified in WAP-219-TLS-20010411-a (http://www.wapforum.com/what/technical.htm)

BNF Format for `Connector.open()` string

The URI must conform to the BNF syntax specified below. If the URI does not conform to this syntax, an `IllegalArgumentException` is thrown.

`<socket_connection_string>`	`::=` **`"ssl://"`**`<hostport>`
`<hostport>`	`::=` *`host ":" port`*
`<host>`	`::=` *`host name or IP address`*
`<port>`	`::=` *`numeric port number`*

In addition, the optional parameters to configure the OTA connection as specified in the `javax.microedition.io.Connector` class description are allowed.

Examples

The following examples show how a `SecureConnection` would be used to access a sample loopback program.

```
SecureConnection sc = (SecureConnection)Connector.open("ssl://host.com:79");
SecurityInfo info = sc.getSecurityInfo();
boolean isTLS = (info.getProtocolName().equals("TLS"));

sc.setSocketOption(SocketConnection.LINGER, 5);
InputStream is = sc.openInputStream();
OutputStream os = sc.openOutputStream();
os.write("\r\n".getBytes());
int ch = 0;
while(ch != -1) {
    ch = is.read();
}
```

```
is.close();
os.close();
sc.close();
```

Since: IMP-NG / MIDP 2.0

Member Summary

Methods

`SecurityInfo getSecurityInfo()`

Inherited Member Summary

Fields inherited from interface `SocketConnection`

`DELAY, KEEPALIVE, LINGER, RCVBUF, SNDBUF`

Methods inherited from interface `Connection`

`close()`

Methods inherited from interface `InputConnection`

`openDataInputStream(), openInputStream()`

Methods inherited from interface `OutputConnection`

`openDataOutputStream(), openOutputStream()`

Methods inherited from interface `SocketConnection`

`getAddress(), getLocalAddress(), getLocalPort(), getPort(), getSocketOption(byte), setSocketOption(byte, int)`

Methods

getSecurityInfo()

Declaration:
`public javax.microedition.io.SecurityInfo getSecurityInfo() throws IOException`

Description:
Return the security information associated with this connection when it was opened.

Returns: the security information associated with this open connection.

Throws:
`java.io.IOException` - if an arbitrary connection failure occurs

javax.microedition.io SecurityInfo

Declaration

```
public interface SecurityInfo
```

Description

This interface defines methods to access information about a secure network connection. Protocols that implement secure connections may use this interface to report the security parameters of the connection. It provides the certificate, protocol, version, and cipher suite, etc. in use.

Since: IMP-NG / MIDP 2.0

See Also: `javax.microedition.pki.CertificateException`, `SecureConnection`, `HttpsConnection`

Member Summary

Methods

`java.lang.String`	`getCipherSuite()`
`java.lang.String`	`getProtocolName()`
`java.lang.String`	`getProtocolVersion()`
`javax.microedition.pki.Certificate`	`getServerCertificate()`

Methods

getCipherSuite()

Declaration:
```
public String getCipherSuite()
```

Description:
Returns the name of the cipher suite in use for the connection. The name returned is from the CipherSuite column of CipherSuite definitions table in Appendix C of RFC 2246. If the cipher suite is not in Appendix C, the name returned is non-null and its contents are not specified. For non-TLS implementions the cipher suite name should be selected according actual key exchange, cipher, and hash combination used to establish the connection, so that regardless of whether the secure connection uses SSL V3 or TLS 1.0 or WTLS or WAP TLS Profile and Tunneling, equivalent cipher suites have the same name.

Returns: a String containing the name of the cipher suite in use.

getProtocolName()

Declaration:
```
public String getProtocolName()
```

Description:
Return the secure protocol name.

Returns: a String containing the secure protocol identifier; if TLS (RFC 2246) orWAP TLS Profile and Tunneling (WAP-219-TLS) is used for the connection the return value is “TLS”; if SSL V3 (The SSL Protocol Version 3.0) is used for the connection; the return value is “SSL”); if WTLS (WAP 199) is used for the connection the return value is “WTLS”.

getProtocolVersion()

Declaration:
`public String getProtocolVersion()`

Description:
Return the protocol version. If appropriate, it should contain the major and minor versions for the protocol separated with a “.” (Unicode x2E).

```
For SSL V3 it MUST return "3.0"
For TLS 1.0 it MUST return "3.1"
For WTLS (WAP-199) it MUST return "1"
For WAP TLS Profile and Tunneling Specification it MUST return "3.1"
```

Returns: a String containing the version of the protocol; the return MUST NOT be `null`.

getServerCertificate()

Declaration:
`public javax.microedition.pki.Certificate getServerCertificate()`

Description:
Get the `Certificate` used to establish the secure connection with the server.

Returns: the `Certificate` used to establish the secure connection with the server.

javax.microedition.io

ServerSocketConnection

Declaration

```
public interface ServerSocketConnection extends StreamConnectionNotifier
```

All Superinterfaces: `Connection, StreamConnectionNotifier`

Description

This interface defines the server socket stream connection.

A server socket is accessed using a generic connection string with the host omitted. For example, `socket://:79` defines an inbound server socket on port `79`. The host can be discovered using the `getLocalAddress` method.

The `acceptAndOpen()` method returns a `SocketConnection` instance. In addition to the normal `StreamConnection` behavior, the `SocketConnection` supports accessing the IP end point addresses of the live connection and access to socket options that control the buffering and timing delays associated with specific application usage of the connection.

Access to server socket connections may be restricted by the security policy of the device. `Connector.open` MUST check access for the initial server socket connection and `acceptAndOpen` MUST check before returning each new `SocketConnection`.

A server socket can be used to dynamically select an available port by omitting both the host and the port parameters in the connection URL string. For example, `socket://` defines an inbound server socket on a port, which is allocated by the system. To discover the assigned port number use the `getLocalPort` method.

BNF Format for `Connector.open()` string

The URI must conform to the BNF syntax specified below. If the URI does not conform to this syntax, an `IllegalArgumentException` is thrown.

`<socket_connection_string>`	`::=` **"socket://"** | **"socket://"**`<hostport>`
`<hostport>`	`::=` *host* ":" *port*
`<host>`	`::= omitted for inbound connections, See SocketConnection`
`<port>`	`::=` *numeric port number* `(omitted for system assigned port)`

Examples

The following examples show how a `ServerSocketConnection` would be used to access a sample loopback program.

```
// Create the server listening socket for port 1234
ServerSocketConnection scn = (ServerSocketConnection)Connector.open("socket://:1234");
// Wait for a connection.
SocketConnection sc = (SocketConnection) scn.acceptAndOpen();
// Set application specific hints on the socket.
sc.setSocketOption(DELAY, 0);
sc.setSocketOption(LINGER, 0);
sc.setSocketOption(KEEPALIVE, 0);
sc.setSocketOption(RCVBUF, 128);
sc.setSocketOption(SNDBUF, 128);
// Get the input stream of the connection.
DataInputStream is = sc.openDataInputStream();
// Get the output stream of the connection.
DataOutputStream os = sc.openDataOutputStream();
// Read the input data.
String result = is.readUTF();
// Echo the data back to the sender.
```

```
os.writeUTF(result);
// Close everything.
is.close();
os.close();
sc.close();
scn.close();
..
```

Since: IMP-NG / MIDP 2.0

Member Summary

Methods

`java.lang.String`	`getLocalAddress()`
`int`	`getLocalPort()`

Inherited Member Summary

Methods inherited from interface `Connection`

`close()`

Methods inherited from interface `StreamConnectionNotifier`

`acceptAndOpen()`

Methods

getLocalAddress()

Declaration:
`public String` **`getLocalAddress`**`() throws IOException`

Description:
Gets the local address to which the socket is bound.
The host address(IP number) that can be used to connect to this end of the socket connection from an external system. Since IP addresses may be dynamically assigned, a remote application will need to be robust in the face of IP number reasssignment.
The local hostname (if available) can be accessed from `System.getProperty("microedition.hostname")`

Returns: the local address to which the socket is bound.

Throws:
`java.io.IOException` - if the connection was closed

See Also: `SocketConnection`

getLocalPort()

Declaration:
`public int` **`getLocalPort`**`() throws IOException`

Description:
Returns the local port to which this socket is bound.

Returns: the local port number to which this socket is connected.

Throws:
`java.io.IOException` - if the connection was closed

See Also: `SocketConnection`

javax.microedition.io

SocketConnection

Declaration

`public interface` **`SocketConnection extends StreamConnection`**

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

All Known Subinterfaces: `SecureConnection`

Description

This interface defines the socket stream connection.

A socket is accessed using a generic connection string with an explicit host and port number. The host may be specified as a fully qualified host name or IPv4 number. e.g. `socket://host.com:79` defines a target socket on the `host.com` system at port `79`.

Note that RFC1900 recommends the use of names rather than IP numbers for best results in the event of IP number reassignment.

Closing Streams

Every `StreamConnection` provides a `Connection` object as well as an `InputStream` and `OutputStream` to handle the I/O associated with the connection. Each of these interfaces has its own `close()` method. For systems that support duplex communication over the socket connection, closing of the input or output stream SHOULD shutdown just that side of the connection. e.g. closing the `InputStream` will permit the `OutputStream` to continue sending data.

Once the input or output stream has been closed, it can only be reopened with a call to `Connector.open()`. The application will receive an `IOException` if an attempt is made to reopen the stream.

BNF Format for `Connector.open()` string

The URI must conform to the BNF syntax specified below. If the URI does not conform to this syntax, an `IllegalArgumentException` is thrown.

`<socket_connection_string>`	`::=` ***`"socket://"`***`<hostport>`
`<hostport>`	`::=` *`host ":" port`*
`<host>`	`::=` *`host name or IP address`* `(omitted for inbound connections, See ServerSocketConnection)`
`<port>`	`::=` *`numeric port number`*

In addition, the optional parameters to configure the OTA connection as specified in the `javax.microedition.io.Connector` class description are allowed.

Examples

The following examples show how a `SocketConnection` would be used to access a sample loopback program.

```
SocketConnection sc = (SocketConnection)Connector.open("socket://host.com:79");
sc.setSocketOption(SocketConnection.LINGER, 5);
InputStream is = sc.openInputStream();
OutputStream os = sc.openOutputStream();
os.write("\r\n".getBytes());
int ch = 0;
while(ch != -1) {
    ch = is.read();
}
is.close();
```

```
os.close();
sc.close();
```

Since: IMP-NG / MIDP 2.0

Member Summary

Fields

`static byte`	`DELAY`
`static byte`	`KEEPALIVE`
`static byte`	`LINGER`
`static byte`	`RCVBUF`
`static byte`	`SNDBUF`

Methods

`java.lang.String`	`getAddress()`
`java.lang.String`	`getLocalAddress()`
`int`	`getLocalPort()`
`int`	`getPort()`
`int`	`getSocketOption(byte option)`
`void`	`setSocketOption(byte option, int value)`

Inherited Member Summary

Methods inherited from interface `Connection`
`close()`

Methods inherited from interface `InputConnection`
`openDataInputStream(), openInputStream()`

Methods inherited from interface `OutputConnection`
`openDataOutputStream(), openOutputStream()`

Fields

DELAY

Declaration:
`public static final byte` **`DELAY`**

Description:
Socket option for the small buffer *writing delay* (0). Set to zero to disable Nagle algorithm for small buffer operations. Set to a non-zero value to enable.

KEEPALIVE

Declaration:
`public static final byte` **`KEEPALIVE`**

Description:
Socket option for the *keep alive* feature (2). Setting KEEPALIVE to zero will disable the feature. Setting KEEPALIVE to a non-zero value will enable the feature.

LINGER

Declaration:
`public static final byte` **`LINGER`**

Description:
Socket option for the *linger time* to wait in seconds before closing a connection with pending data output (1). Setting the linger time to zero disables the linger wait interval.

RCVBUF

Declaration:
`public static final byte` **`RCVBUF`**

Description:
Socket option for the size of the *receiving buffer* (3).

SNDBUF

Declaration:
`public static final byte` **`SNDBUF`**

Description:
Socket option for the size of the *sending buffer* (4).

Methods

getAddress()

Declaration:
`public String` **`getAddress`**`() throws IOException`

Description:
Gets the remote address to which the socket is bound. The address can be either the remote host name or the IP address (if available).

Returns: the remote address to which the socket is bound.

Throws:
`java.io.IOException` - if the connection was closed.

getLocalAddress()

Declaration:
`public String` **`getLocalAddress`**`() throws IOException`

Description:
Gets the local address to which the socket is bound.
The host address (IP number) that can be used to connect to this end of the socket connection from an external system. Since IP addresses may be dynamically assigned, a remote application will need to be robust in the face of IP number reasssignment.
The local hostname (if available) can be accessed from `System.getProperty("microedition.hostname")`

Returns: the local address to which the socket is bound.

Throws:
`java.io.IOException` - if the connection was closed.

See Also: `ServerSocketConnection`

getLocalPort()

Declaration:
`public int` **`getLocalPort`**`() throws IOException`

Description:
Returns the local port to which this socket is bound.

Returns: the local port number to which this socket is connected.

Throws:
`java.io.IOException` - if the connection was closed.

See Also: `ServerSocketConnection`

getPort()

Declaration:
```
public int getPort() throws IOException
```

Description:
Returns the remote port to which this socket is bound.

Returns: the remote port number to which this socket is connected.

Throws:
`java.io.IOException` - if the connection was closed.

getSocketOption(byte)

Declaration:
```
public int getSocketOption(byte option) throws IllegalArgumentException, IOException
```

Description:
Get a socket option for the connection.

Parameters:
`option` - socket option identifier (`KEEPALIVE`, `LINGER`, `SNDBUF`, `RCVBUF`, or `DELAY`)

Returns: numeric value for specified option or `-1` if the value is not available.

Throws:
`IllegalArgumentException` - if the option identifier is not valid
`java.io.IOException` - if the connection was closed

See Also: `setSocketOption(byte, int)`

setSocketOption(byte, int)

Declaration:
```
public void setSocketOption(byte option, int value)
          throws IllegalArgumentException, IOException
```

Description:
Set a socket option for the connection.
Options inform the low level networking code about intended usage patterns that the application will use in dealing with the socket connection.
Calling `setSocketOption` to assign buffer sizes is a hint to the platform of the sizes to set the underlying network I/O buffers. Calling `getSocketOption` can be used to see what sizes the system is using. The system MAY adjust the buffer sizes to account for better throughput available from Maximum Transmission Unit (MTU) and Maximum Segment Size (MSS) data available from current network information.

Parameters:
`option` - socket option identifier (`KEEPALIVE`, `LINGER`, `SNDBUF`, `RCVBUF`, or `DELAY`)
`value` - numeric value for specified option

Throws:

`IllegalArgumentException` - if the value is not valid (e.g. negative value) or if the option identifier is not valid

`java.io.IOException` - if the connection was closed

See Also: `getSocketOption(byte)`

javax.microedition.io

UDPDatagramConnection

Declaration

`public interface` **`UDPDatagramConnection extends DatagramConnection`**

All Superinterfaces: `Connection, DatagramConnection`

Description

This interface defines a datagram connection which knows it's local end point address. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the `SocketConnection`.

A `UDPDatagramConnection` is returned from `Connector.open()` in response to a request to open a `datagram://` URL connection string. If the connection string omits both the `host` and `port` fields in the URL string, then the system will allocate an available port. The local address and the local port can be discovered using the accessor methods within this interface.

The syntax described here for the datagram URL connection string is also valid for the `Datagram.setAddress()` method used to assign a destination address to a `Datagram` to be sent. e.g., `datagram://`*`host:port`*

BNF Format for `Connector.open()` string

The URI must conform to the BNF syntax specified below. If the URI does not conform to this syntax, an `IllegalArgumentException` is thrown.

`<datagram_connection_string>`	`::=` **`"datagram://"`** | **`"datagram://"`**`<hostport>`
`<hostport>`	`::=` *`host`* `":"` *`port`*
`<host>`	`::=` *`host name or IP address`* `(omitted for inbound connections)`
`<port>`	`::=` *`numeric port number`* `(omitted for system assigned port)`

In addition, the optional parameters to configure the OTA connection as specified in the `javax.microedition.io.Connector` class description are allowed.

Since: IMP-NG / MIDP 2.0

Member Summary

Methods

`java.lang.String`	`getLocalAddress()`
`int`	`getLocalPort()`

Inherited Member Summary

Methods inherited from interface `Connection`

`close()`

Methods inherited from interface `DatagramConnection`

`getMaximumLength(), getNominalLength(), newDatagram(byte[], int, String), newDatagram(byte[], int, String), newDatagram(byte[], int, String), newDatagram(byte[], int, String), receive(Datagram), send(Datagram)`

Methods

getLocalAddress()

Declaration:
`public String getLocalAddress() throws IOException`

Description:
Gets the local address to which the datagram connection is bound.
The host address(IP number) that can be used to connect to this end of the datagram connection from an external system. Since IP addresses may be dynamically assigned, a remote application will need to be robust in the face of IP number reasssignment.
The local hostname (if available) can be accessed from `System.getProperty("microedition.hostname")`

Returns: the local address to which the datagram connection is bound.

Throws:
`java.io.IOException` - if the connection was closed.

See Also: `ServerSocketConnection`

getLocalPort()

Declaration:
`public int getLocalPort() throws IOException`

Description:
Returns the local port to which this datagram connection is bound.

Returns: the local port number to which this datagram connection is connected.

Throws:
`java.io.IOException` - if the connection was closed.

See Also: `ServerSocketConnection`

10 Package javax.microedition.media

Description

The IMP-NG Media API is – like the MIDP 2.0 Media API – a directly compatible building block of the Mobile Media API (JSR-135) specification. The use of this building block is intended for J2ME™ profiles aiming to include sound support in the specification, while maintaining upwards compatibility with the full Multimedia API. Such specification example is MIDP 2.0 (JSR-118), and now also IMP-NG. The development of these two interoperable APIs enables seamless sound and multimedia content creation across the J2ME™ range of devices using the same API principles.

For IMP-NG, though, the Media API is completely optional. That means, an implementation can decide, whether it will implement the Media API or not. This takes into account, that IMs are used for very different purpose, and it makes therefore no sense to force every IMP-NG compatible device to implement the Media API, even if there is no necessity to handle any multimedia content.

Introduction

J2ME™ devices range from cell phones with simple tone generation to PDAs and Web tablets with advanced audio and video rendering capabilities. IMP-NG addresses mostly simple Information Modules (IMs) like wireless modules.

To accommodate diverse configurations and multimedia processing capabilities, an API with a high level of abstraction is needed. The goal of the MMAPI Expert Group work has been to address this wide range of application areas, and the result of the work is a proposal of two API sets:

- Mobile Media API (JSR 135)
- MIDP 2.0 Media API

The first API is intended for J2ME™ devices with advanced sound and multimedia capabilities, including powerful mobile phones, PDAs, and set-top boxes, for example. The latter API is a directly compatible subset of the Multimedia API, and is intended for resource-constrained devices such as mass-market mobile devices (running MIDP 2.0). Furthermore, this subset API can be adapted to other J2ME™ profiles requiring sound support. The IMP-NG expert group (IMPNGEG) decided to use this building block as an optional component of the IMP-NG. In the following, a more detailed description of the background and requirements of the building block API is given.

Background of the Media API

Some J2ME™ devices are very resource constrained. It may not be feasible for a device to support a full range of multimedia types, such as video on some cell phones. As a result, not all devices are required to support the full generality of a multimedia API, such as extensibility to support custom protocols and content types. The proposed building block subset API has been designed to meet the above constraints. This proposed building block fulfills the requirements set by the IMP-NG expert group. These requirements include:

- Low footprint audio playback
- Protocol and content format agnostic
- Supports tone generation
- Supports general media flow controls: start, stop, etc.
- Supports media-specific type controls: volume etc.
- Supports capability query

This subset differs from the full Mobile Media API in the following ways:

- It is audio-only. It excludes all Controls specific to video or graphics.
- It does not support custom protocols via custom DataSources. The `javax.microedition.media.protocol` package (DataSource) is excluded.

It is important to note that the building block subset used in IMP-NG is a proper subset of the full Mobile Media API and is fully forward compatible.

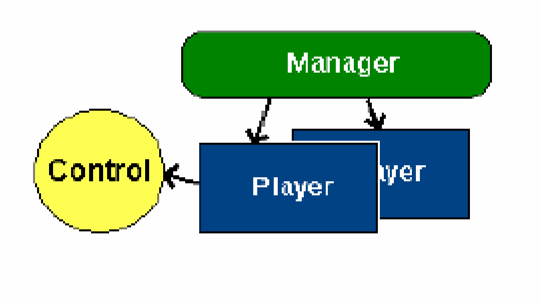
Basic Concepts

The proposed audio building block system constists of three main parts: `Manager`, `Player` and `Control`.

The `Manager` is the top level controller of audio resources. Applications use `Manager` to request `Players` and to query properties, supported content types and supported protocols. The manager also includes a method to play simple tones.

The `Player` plays the multimedia content. The application obtains a `Player` by giving the locator string to `Manager`.

A `Control` is an interface that is used to implement all different controls a `Player` might have. An application can query a `Player` of controls it supports and then ask for a specific `Control` e.g. `VolumeControl` to control volume.



API Details

The `createPlayer` method is the top-level entry point to the API:

```
Player Manager.createPlayer(String url)
```

The url fully specifies the protocol and the content of the data:

```
<protocol>:<content location>
```

The `Manager` parses the URL, recognizes the content type and creates a `Player` to handle the presentation of the data. The resulting `Player` is returned for use by the application. Connections created by `createPlayer` follow the Generic Connection framework rules and policies.

The `Player` provides general methods to control the data flow and presentation, for example:

- `Player.realize()`
- `Player.prefetch()`
- `Player.start()`

Fine-grained control is an important feature of the API; therefore, each `Player` also provides type-specific controls with the `getControls` and `getControl` methods:

- `Control[] Player.getControls()`
- `Control Player.getControl(int controlType)`

Since different types of media will yield different types of controls from its corresponding `Player`, the `getControls` `and` `getControl` methods can expose features that are unique to a particular media type.

Package javax.microedition.media

Tone Generation

Tone generation is important for every application using audio. On very small devices like IMs, it is particularly important since that is likely to be the only form of multimedia capabilities supported. In its simplest form, tone generation reduces to a single buzzer or some simple monophonic tone generation. The `Manager` class provides a top level method to handle this simple form of single tone generation:

```
Manager.playTone(int note, int duration, int volume)
```

The implementation of this method can be mapped directly to the device's hardware tone generator to provide the most responsive sound generation.

In addition, the API also provides a way to create a specific type of `Player` for synthesizing tone sequences:

```
Player p = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR)
```

The `Player` created provides a special type of `Control`, `ToneControl` which can be used for programming a tone sequence. This enables more sophisticated applications written for slightly more powerful devices.

Usage Scenarios

In this section we demonstrate how the API could be used in four common scenarios.

Scenario 1: Single-Tone Generation

```
try {
    Manager.playTone(ToneControl.C4, 5000 /* ms */, 100 /* max vol */);
} catch (MediaException e) { }
```

Scenario 2: Simple Media Playback with Looping

Notice that in IMP-NG the wav format is mandatory only in a case the device supports sampled audio.

```
try {
    Player p = Manager.createPlayer("http://webserver/music.wav");
    p.setLoopCount(5);
    p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```

Scenario 3: Playing Back from Media Stored in JAR

Notice that in IMP-NG the wav format is mandatory only in a case the device supports sampled audio.

```
try {
    InputStream is = getClass().getResourceAsStream("music.wav");
    Player p = Manager.createPlayer(is, "audio/X-wav");
    p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```

Scenario 4: Tone Sequence Generation

```
/**
* "Mary Had A Little Lamb" has "ABAC" structure.
* Use block to repeat "A" section.
*/
byte tempo = 30; // set tempo to 120 bpm
byte d = 8; // eighth-note
byte C4 = ToneControl.C4;;
byte D4 = (byte)(C4 + 2); // a whole step
byte E4 = (byte)(C4 + 4); // a major third
byte G4 = (byte)(C4 + 7); // a fifth
byte rest = ToneControl.SILENCE; // rest
byte[] mySequence = {
    ToneControl.VERSION, 1, // version 1
    ToneControl.TEMPO, tempo, // set tempo
    ToneControl.BLOCK_START, 0, // start define "A" section
    E4,d, D4,d, C4,d, E4,d, // content of "A" section
    E4,d, E4,d, E4,d, rest,d,
    ToneControl.BLOCK_END, 0, // end define "A" section
    ToneControl.PLAY_BLOCK, 0, // play "A" section
```

```
        D4,d, D4,d, D4,d, rest,d, // play "B" section
        E4,d, G4,d, G4,d, rest,d,
        ToneControl.PLAY_BLOCK, 0, // repeat "A" section
        D4,d, D4,d, E4,d, D4,d, C4,d // play "C" section
    };
    try{
        Player p = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
        p.realize();
        ToneControl c = (ToneControl)p.getControl("ToneControl");
        c.setSequence(mySequence);
        p.start();
    } catch (IOException ioe) {
    } catch (MediaException me) { }
```

Since: IMP-NG / MIDP 2.0

Class Summary

Interfaces	
`Control`	A `Control` object is used to control some media processing functions.
`Controllable`	`Controllable` provides an interface for obtaining the `Controls` from an object like a `Player`.
`Player`	`Player` controls the rendering of time based media data.
`PlayerListener`	`PlayerListener` is the interface for receiving asynchronous events generated by `Players`.
Classes	
`Manager`	`Manager` is the access point for obtaining system dependent resources such as `Players` for multimedia processing.
Exceptions	
`MediaException`	A `MediaException` indicates an unexpected error condition in a method.

javax.microedition.media
Control

Declaration

`public interface` **`Control`**

All Known Subinterfaces: `javax.microedition.media.control.ToneControl`, `javax.microedition.media.control.VolumeControl`

Description

A `Control` object is used to control some media processing functions. The set of operations are usually functionally related. Thus a `Control` object provides a logical grouping of media processing functions.

`Controls` are obtained from `Controllable`. The `Player` interface extends `Controllable`. Therefore a `Player` implementation can use the `Control` interface to extend its media processing functions. For example, a `Player` can expose a `VolumeControl` to allow the volume level to be set.

Multiple `Controls` can be implemented by the same object. For example, an object can implement both `VolumeControl` and `ToneControl`. In this case, the object can be used for controlling both the volume and tone generation.

The `javax.microedition.media.control` package specifies a set of pre-defined `Controls`.

See Also: `Controllable`, `Player`

javax.microedition.media

Controllable

Declaration

`public interface` **`Controllable`**

All Known Subinterfaces: `Player`

Description

`Controllable` provides an interface for obtaining the `Controls` from an object like a `Player`. It provides methods to query all the supported `Controls` and to obtain a particular `Control` based on its class name.

Member Summary

Methods

`Control`	`getControl(String controlType)`
`Control[]`	`getControls()`

Methods

getControl(String)

Declaration:
`public javax.microedition.media.Control` **`getControl`**`(String controlType)`

Description:
Obtain the object that implements the specified `Control` interface.
If the specified `Control` interface is not supported then `null` is returned.
If the `Controllable` supports multiple objects that implement the same specified `Control` interface, only one of them will be returned. To obtain all the `Control`'s of that type, use the `getControls` method and check the list for the requested type.

Parameters:
`controlType` - the class name of the `Control`. The class name should be given either as the fully qualified name of the class; or if the package of the class is not given, the package `javax.microedition.media.control` is assumed.

Returns: the object that implements the control, or `null`.

Throws:
`IllegalArgumentException` - Thrown if `controlType` is `null`.
`IllegalStateException` - Thrown if `getControl` is called in a wrong state. See `Player` for more details.

getControls()

Declaration:
`public javax.microedition.media.Control[]` **`getControls`**`()`

Description:
Obtain the collection of `Controls` from the object that implements this interface.
Since a single object can implement multiple `Control` interfaces, it's necessary to check each object against different `Control` types. For example:

```
Controllable controllable;
:
Control cs[];
cs = controllable.getControls();
for (int i = 0; i < cs.length; i++) {
   if (cs[i] instanceof ControlTypeA)
      doSomethingA();
   if (cs[i] instanceof ControlTypeB)
      doSomethingB();
   // etc.
}
```

The list of `Control` objects returned will not contain any duplicates. And the list will not change over time. If no `Control` is supported, a zero length array is returned.

Returns: the collection of `Control` objects.

Throws:

`IllegalStateException` - Thrown if `getControls` is called in a wrong state. See `Player` for more details.

javax.microedition.media Manager

Declaration

```
public final class Manager
```

```
Object
   |
   +--javax.microedition.media.Manager
```

Description

`Manager` is the access point for obtaining system dependent resources such as `Players` for multimedia processing.

A `Player` is an object used to control and render media that is specific to the content type of the data.

`Manager` provides access to an implementation specific mechanism for constructing `Players`.

For convenience, `Manager` also provides a simplified method to generate simple tones.

Simple Tone Generation

The `playTone` function is defined to generate tones. Given the note and duration, the function will produce the specified tone.

Creating Players

`Manager` provides two methods to create a `Player` for playing back media:

- Create from a media locator.
- Create from an `InputStream`.

The `Player` returned can be used to control the presentation of the media.

Content Types

Content types identify the type of media data. They are defined to be the registered MIME types (http://www.iana.org/assignments/media-types/); plus some user-defined types that generally follow the MIME syntax (RFC 2045, RFC 2046).

For example, here are a few common content types:

1. Wave audio files: `audio/x-wav`
2. AU audio files: `audio/basic`
3. MP3 audio files: `audio/mpeg`
4. MIDI files: `audio/midi`
5. Tone sequences: `audio/x-tone-seq`

Media Locator

Media locators are specified in URI syntax (http://www.ietf.org/rfc/rfc2396.txt), which is defined in the form:

```
<scheme>:<scheme-specific-part>
```

The “scheme” part of the locator string identifies the name of the protocol being used to deliver the data.

Package javax.microedition.media

Configuration of OTA connections

The URI string serving as a media locator is subject to the same optional parameter extensions for OTA connection configuration as described in the class description for `javax.microedition.io.Connector`.

See Also: `Player`

Member Summary

Fields

`static java.lang.String`	`TONE_DEVICE_LOCATOR`

Methods

`static Player`	`createPlayer(java.io.InputStream stream, String type)`
`static Player`	`createPlayer(String locator)`
`static java.lang.String[]`	`getSupportedContentTypes(String protocol)`
`static java.lang.String[]`	`getSupportedProtocols(String content_type)`
`static void`	`playTone(int note, int duration, int volume)`

Inherited Member Summary

Methods inherited from class `Object`

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

Fields

TONE_DEVICE_LOCATOR

Declaration:
`public static final String` **`TONE_DEVICE_LOCATOR`**

Description:
The locator to create a tone `Player` to play back tone sequences. e.g.

```
try {
   Player p = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
   p.realize();
   ToneControl tc = (ToneControl)p.getControl("ToneControl");
   tc.setSequence(mySequence);
   p.start();
} catch (IOException ioe) {
} catch (MediaException me) {}
```

If a tone sequence is not set on the tone `Player` via its `ToneControl`, the `Player` does not carry any sequence. `getDuration` returns `0` for this `Player`.
The content type of the `Player` created from this locator is `audio/x-tone-seq`.
A `Player` for this locator may not be supported for all implementations.
Value `"device://tone"` is assigned to `TONE_DEVICE_LOCATOR`.

Methods

createPlayer(InputStream, String)

Declaration:

```
public static javax.microedition.media.Player createPlayer(java.io.InputStream stream,
          String type)
          throws IOException, MediaException
```

Description:
Create a `Player` to play back media from an `InputStream`.
The `type` argument specifies the content-type of the input media. If `null` is given, `Manager` will attempt to determine the type. However, since determining the media type is non-trivial for some media types, it may not be feasible in some cases. The `Manager` may throw a `MediaException` to indicate that.

Parameters:
`stream` - The `InputStream` that delivers the input media.
`type` - The `ContentType` of the media.

Returns: A new `Player`.

Throws:
`IllegalArgumentException` - Thrown if `stream` is `null`.
`MediaException` - Thrown if a `Player` cannot be created for the given stream and type.
`java.io.IOException` - Thrown if there was a problem reading data from the `InputStream`.
`SecurityException` - Thrown if the caller does not have security permission to create the `Player`.

createPlayer(String)

Declaration:
```
public static javax.microedition.media.Player createPlayer(String locator)
                throws IOException, MediaException
```

Description:
Create a `Player` from an input locator.

Parameters:
`locator` - A locator string in URI syntax that describes the media content.

Returns: A new `Player`.

Throws:
`IllegalArgumentException` - Thrown if `locator` is `null`.
`MediaException` - Thrown if a `Player` cannot be created for the given locator.
`java.io.IOException` - Thrown if there was a problem connecting with the source pointed to by the `locator`.
`SecurityException` - Thrown if the caller does not have security permission to create the `Player`.

getSupportedContentTypes(String)

Declaration:
```
public static String[] getSupportedContentTypes(String protocol)
```

Description:
Return the list of supported content types for the given protocol.
See content types for the syntax of the content types returned. See protocol name for the syntax of the protocol used.
For example, if the given `protocol` is `"http"`, then the supported content types that can be played back with the `http` protocol will be returned.
If `null` is passed in as the `protocol`, all the supported content types for this implementation will be returned. The returned array must be non-empty.
If the given `protocol` is an invalid or unsupported protocol, then an empty array will be returned.

Parameters:
`protocol` - The input protocol for the supported content types.

Returns: The list of supported content types for the given protocol.

getSupportedProtocols(String)

Declaration:
`public static String[] getSupportedProtocols(String content_type)`

Description:
Return the list of supported protocols given the content type. The protocols are returned as strings which identify what locators can be used for creating `Players`.
See protocol name for the syntax of the protocols returned. See content types for the syntax of the content type used.
For example, if the given `content_type` is `"audio/x-wav"`, then the supported protocols that can be used to play back `audio/x-wav` will be returned.
If `null` is passed in as the `content_type`, all the supported protocols for this implementation will be returned. The returned array must be non-empty.
If the given `content_type` is an invalid or unsupported content type, then an empty array will be returned.

Parameters:
`content_type` - The content type for the supported protocols.

Returns: The list of supported protocols for the given content type.

playTone(int, int, int)

Declaration:
`public static void playTone(int note, int duration, int volume) throws MediaException`

Description:
Play back a tone as specified by a note and its duration. A note is given in the range of `0` to `127` inclusive.
The frequency of the note can be calculated from the following formula:

```
SEMITONE_CONST = 17.31234049066755 = 1/(ln(2^(1/12)))
note = ln(freq/8.176)*SEMITONE_CONST
The musical note A = MIDI note 69 (0x45) = 440 Hz.
```

This call is a non-blocking call. Notice that this method may utilize CPU resources significantly on devices that don’t have hardware support for tone generation.

Parameters:
`note` - Defines the tone of the note as specified by the above formula.
`duration` - The duration of the tone in milli-seconds. Duration must be positive.
`volume` - Audio volume range from `0` to `100`. `100` represents the maximum volume at the current hardware level. Setting the volume to a value less than `0` will set the volume to `0`. Setting the volume to greater than `100` will set the volume to `100`.

Throws:
`IllegalArgumentException` - Thrown if the given note or duration is out of range.
`MediaException` - Thrown if the tone cannot be played due to a device-related problem.

Package
javax.microedition.media

javax.microedition.media
MediaException

Declaration

```
public class MediaException extends Exception
```

```
Object
   |
   +--Throwable
         |
         +--Exception
               |
               +--javax.microedition.media.MediaException
```

Description

A `MediaException` indicates an unexpected error condition in a method.

Member Summary
Constructors
`MediaException()`
`MediaException(String reason)`

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

Constructors

MediaException()

Declaration:
`public MediaException()`

Description:
Constructs a `MediaException` with `null` as its error detail message.

Declaration:
`public MediaException(String reason)`

Description:
Constructs a `MediaException` with the specified detail message. The error message string `s` can later be retrieved by the `Throwable.getMessage()` method of class `java.lang.Throwable`.

Parameters:
`reason` - the detail message.

javax.microedition.media

Player

Declaration

`public interface` **`Player extends Controllable`**

All Superinterfaces: `Controllable`

Description

`Player` controls the rendering of time based media data. It provides the methods to manage the `Player`'s life cycle, controls the playback progress and obtains the presentation components.

Simple Playback

A `Player` can be created from one of the `Manager`'s `createPlayer` methods. After the `Player` is created, calling `start` will start the playback as soon as possible. The method will return when the playback is started. The playback will continue in the background and will stop automatically when the end of media is reached.

Simple playback example illustrates this.

Player Life Cycle

A `Player` has five states: `UNREALIZED`, `REALIZED`, `PREFETCHED`, `STARTED`, `CLOSED`.

The purpose of these life-cycle states is to provide programmatic control over potentially time-consuming operations. For example, when a `Player` is first constructed, it's in the `UNREALIZED` state. Transitioned from `UNREALIZED` to `REALIZED`, the `Player` performs the communication necessary to locate all of the resources it needs to function (such as communicating with a server or a file system). The `realize` method allows an application to initiate this potentially time-consuming process at an appropriate time.

Typically, a `Player` moves from the `UNREALIZED` state to the `REALIZED` state, then to the `PREFETCHED` state, and finally on to the `STARTED` state.

A `Player` stops when it reaches the end of media; or when the `stop` method is invoked. When that happens, the `Player` moves from the `STARTED` state back to the `PREFETCHED` state. It is then ready to repeat the cycle.

To use a `Player`, one must set up parameters to manage its movement through these life-cycle states and then move it through the states using the `Player`'s state transition methods.

Player States

This section describes the semantics of each of the `Player` states.

`UNREALIZED` State

A `Player` starts in the *`UNREALIZED`* state. An unrealized `Player` does not have enough information to acquire all the resources it needs to function.

The following methods must not be used when the `Player` is in the *`UNREALIZED`* state.

- `getContentType`
- `setMediaTime`
- `getControls`
- `getControl`

An `IllegalStateException` will be thrown.

The realize method transitions the Player from the *`UNREALIZED`* state to the *`REALIZED`* state.

`REALIZED` State

A `Player` is in the ***REALIZED*** state when it has obtained the information required to acquire the media resources. Realizing a `Player` can be a resource and time consuming process. The `Player` may have to communicate with a server, read a file, or interact with a set of objects.

Although a realized `Player` does not have to acquire any resources, it is likely to have acquired all of the resources it needs except those that imply exclusive use of a scarce system resource, such as an audio device.

Normally, a `Player` moves from the ***UNREALIZED*** state to the ***REALIZED*** state. After `realize` has been invoked on a `Player`, the only way it can return to the ***UNREALIZED*** state is if `deallocate` is invoked before `realize` is completed. Once a `Player` reaches the ***REALIZED*** state, it never returns to the ***UNREALIZED*** state. It remains in one of four states: ***REALIZED***, ***PREFETCHED***, ***STARTED*** or ***CLOSED***.

`PREFETCHED` State

Once realized, a `Player` may still need to perform a number of time-consuming tasks before it is ready to be started. For example, it may need to acquire scarce or exclusive resources, fill buffers with media data, or perform other start-up processing. Calling `prefetch` on the `Player` carries out these tasks.

Once a `Player` is in the ***PREFETCHED*** state, it may be started. Prefetching reduces the startup latency of a `Player` to the minimum possible value.

When a started `Player` stops, it returns to the ***PREFETCHED*** state.

`STARTED` State

Once prefetched, a `Player` can enter the ***STARTED*** state by calling the `start` method. A ***STARTED*** `Player` means the `Player` is running and processing data. A `Player` returns to the ***PREFETCHED*** state when it stops, because the `stop` method was invoked, or it has reached the end of the media.

When the `Player` moves from the ***PREFETCHED*** to the ***STARTED*** state, it posts a `STARTED` event. When it moves from the ***STARTED*** state to the ***PREFETCHED*** state, it posts a `STOPPED`, or `END_OF_MEDIA` event depending on the reason it stopped.

The following method must not be used when the `Player` is in the ***STARTED*** state:

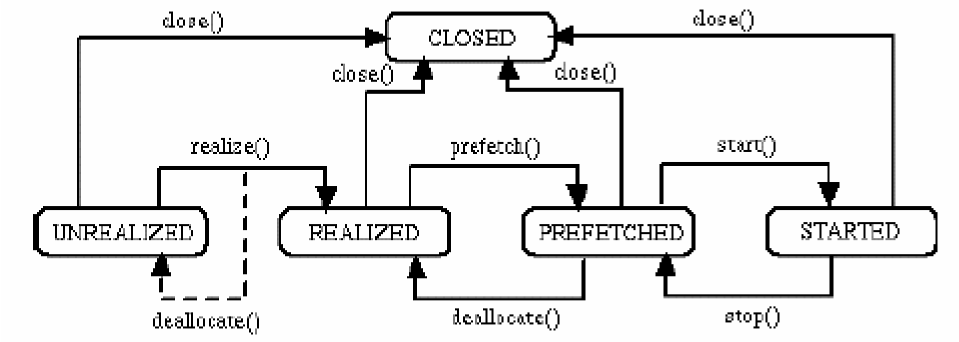
- `setLoopCount`

An `IllegalStateException` will be thrown.

`CLOSED` state

Calling `close` on the `Player` puts it in the ***CLOSED*** state. In the ***CLOSED*** state, the `Player` has released most of its resources and must not be used again.

The `Player`'s five states and the state transition methods are summarized in the following diagram:



Package javax.microedition.media

Player Events

`Player` events asynchronously deliver information about the `Player`'s state changes and other relevant information from the `Player's Controls`.

To receive events, an object must implement the `PlayerListener` interface and use the `addPlayerListener` method to register its interest in a `Player`'s events. All `Player` events are posted to each registered listener.

The events are guaranteed to be delivered in the order that the actions representing the events occur. For example, if a `Player` stops shortly after it starts because it is playing back a very short media file, the `STARTED` event must always preceed the `END_OF_MEDIA` event.

An `ERROR` event may be sent any time an irrecoverable error has occured. When that happens, the `Player` is in the ***`CLOSED`*** state.

The `Player` event mechanism is extensible and some `Players` define events other than the ones described here. For a list of pre-defined player events, check the `PlayerListener` interface.

Managing the Resources Used by a `Player`

The `prefetch` method is used to acquire scarce or exclusive resources such as the audio device. Conversely, the `deallocate` method is used to release the scarce or exclusive resources. By using these two methods, an application can programmatically manage the `Player`'s resources.

For example, in an implementation with an exclusive audio device, to alternate the audio playback of multiple `Players`, an application can selectively deallocate and prefetch individual `Players`.

Player's Controls

`Player` implements `Controllable` which provides extra controls via some type-specific `Control` interfaces. `getControl` and `getControls` cannot be called when the `Player` is in the ***`UNREALIZED`*** or ***`CLOSED`*** state. An `IllegalStateException` will be thrown.

Simple Playback Example

```
try {
    Player p = Manager.createPlayer("http://abc.wav");
    p.start();
} catch (MediaException pe) {
} catch (IOException ioe) {
}
```

Member Summary

Fields

static int	CLOSED
static int	PREFETCHED
static int	REALIZED
static int	STARTED
static long	TIME_UNKNOWN
static int	UNREALIZED

Methods

void	addPlayerListener(PlayerListener playerListener)
void	close()
void	deallocate()
java.lang.String	getContentType()
long	getDuration()
long	getMediaTime()
int	getState()
void	prefetch()
void	realize()
void	removePlayerListener(PlayerListener playerListener)
void	setLoopCount(int count)
long	setMediaTime(long now)
void	start()
void	stop()

Inherited Member Summary

Methods inherited from interface `Controllable`

`getControl(String), getControls()`

Fields

CLOSED

Declaration:
`public static final int` **`CLOSED`**

Description:
The state of the `Player` indicating that the `Player` is closed.
Value `0` is assigned to `CLOSED`.

PREFETCHED

Declaration:
`public static final int` **`PREFETCHED`**

Description:
The state of the `Player` indicating that it has acquired all the resources to begin playing.
Value `300` is assigned to `PREFETCHED`.

REALIZED

Declaration:
`public static final int` **`REALIZED`**

Description:
The state of the `Player` indicating that it has acquired the required information but not the resources to function.
Value `200` is assigned to `REALIZED`.

STARTED

Declaration:
`public static final int` **`STARTED`**

Description:
The state of the `Player` indicating that the `Player` has already started.
Value `400` is assigned to `STARTED`.

TIME_UNKNOWN

Declaration:
`public static final long` **`TIME_UNKNOWN`**

Description:
The returned value indicating that the requested time is unknown.
Value `-1` is assigned to `TIME_UNKNOWN`.

UNREALIZED

Declaration:
`public static final int` **`UNREALIZED`**

Description:
The state of the `Player` indicating that it has not acquired the required information and resources to function.
Value `100` is assigned to `UNREALIZED`.

Methods

addPlayerListener(PlayerListener)

Declaration:
`public void` **`addPlayerListener`**`(javax.microedition.media.PlayerListener playerListener)`

Description:
Add a player listener for this player.

Parameters:
`playerListener` - the listener to add. If `null` is used, the request will be ignored.

Throws:
`IllegalStateException` - Thrown if the `Player` is in the ***`CLOSED`*** state.

See Also: `removePlayerListener(PlayerListener)`

close()

Declaration:
`public void` **`close`**`()`

Description:
Close the `Player` and release its resources.
When the method returns, the `Player` is in the ***`CLOSED`*** state and can no longer be used. A `CLOSED` event will be delivered to the registered `PlayerListeners`.
If `close` is called on a closed `Player` the request is ignored.

deallocate()

Declaration:
`public void deallocate()`

Description:
Release the scarce or exclusive resources like the audio device acquired by the `Player`.
When `deallocate` returns, the `Player` is in the *`UNREALIZED`* or *`REALIZED`* state.
If the `Player` is blocked at the `realize` call while realizing, calling `deallocate` unblocks the `realize` call and returns the `Player` to the *`UNREALIZED`* state. Otherwise, calling `deallocate` returns the `Player` to the *`REALIZED`* state.
If `deallocate` is called when the `Player` is in the *`UNREALIZED`* or *`REALIZED`* state, the request is ignored.
If the `Player` is `STARTED` when `deallocate` is called, `deallocate` will implicitly call `stop` on the `Player`.

Throws:
`IllegalStateException` - Thrown if the `Player` is in the *`CLOSED`* state.

getContentType()

Declaration:
`public String getContentType()`

Description:
Get the content type of the media that's being played back by this `Player`.
See content type for the syntax of the content type returned.

Returns: The content type being played back by this `Player`.

Throws:
`IllegalStateException` - Thrown if the `Player` is in the *`UNREALIZED`* or *`CLOSED`* state.

getDuration()

Declaration:
`public long getDuration()`

Description:
Get the duration of the media. The value returned is the media's duration when played at the default rate. If the duration cannot be determined (for example, the `Player` is presenting live media) `getDuration` returns `TIME_UNKNOWN`.

Returns: The duration in microseconds or `TIME_UNKNOWN`.

Throws:
`IllegalStateException` - Thrown if the `Player` is in the *`CLOSED`* state.

getMediaTime()

Declaration:
`public long getMediaTime()`

Description:
Gets this `Player`'s current *media time*. If the *media time* cannot be determined, `getMediaTime` returns `TIME_UNKNOWN`.

Returns: The current *media time* in microseconds or `TIME_UNKNOWN`.

Throws:

`IllegalStateException` - Thrown if the `Player` is in the *`CLOSED`* state.

See Also: `setMediaTime(long)`

getState()

Declaration:

`public int getState()`

Description:

Gets the current state of this `Player`. The possible states are: *`UNREALIZED`*, *`REALIZED`*, *`PREFETCHED`*, *`STARTED`*, *`CLOSED`*.

Returns: The `Player`'s current state.

prefetch()

Declaration:

`public void prefetch() throws MediaException`

Description:

Acquires the scarce and exclusive resources and processes as much data as necessary to reduce the start latency.
When `prefetch` completes successfully, the `Player` is in the *`PREFETCHED`* state.
If `prefetch` is called when the `Player` is in the *`UNREALIZED`* state, it will implicitly call `realize`.
If `prefetch` is called when the `Player` is already in the *`PREFETCHED`* state, the `Player` may still process data necessary to reduce the start latency. This is to guarantee that start latency can be maintained at a minimum.
If `prefetch` is called when the `Player` is in the *`STARTED`* state, the request will be ignored.
If the `Player` cannot obtain all of the resources it needs, it throws a `MediaException`. When that happens, the `Player` will not be able to start. However, `prefetch` may be called again when the needed resource is later released perhaps by another `Player` or application.

Throws:

`IllegalStateException` - Thrown if the `Player` is in the *`CLOSED`* state.
`MediaException` - Thrown if the `Player` cannot be prefetched.
`SecurityException` - Thrown if the caller does not have security permission to prefetch the `Player`.

realize()

Declaration:

`public void realize() throws MediaException`

Description:

Constructs portions of the `Player` without acquiring the scarce and exclusive resources. This may include examining media data and may take some time to complete.
When `realize` completes successfully, the `Player` is in the *`REALIZED`* state.
If `realize` is called when the `Player` is in the *`REALIZED`*, *`PREFETCHTED`* or *`STARTED`* state, the request will be ignored.

Throws:

`IllegalStateException` - Thrown if the `Player` is in the *`CLOSED`* state.
`MediaException` - Thrown if the `Player` cannot be realized.
`SecurityException` - Thrown if the caller does not have security permission to realize the `Player`.

removePlayerListener(PlayerListener)

Declaration:

`public void removePlayerListener(javax.microedition.media.PlayerListener playerListener)`

Description:
Remove a player listener for this player.

Parameters:
`playerListener` - the listener to remove. If `null` is used or the given `playerListener` is not a listener for this `Player`, the request will be ignored.

Throws:
`IllegalStateException` - Thrown if the `Player` is in the *`CLOSED`* state.

See Also: `addPlayerListener(PlayerListener)`

setLoopCount(int)

Declaration:
`public void` **`setLoopCount`**`(int count)`

Description:
Set the number of times the `Player` will loop and play the content.
By default, the loop count is one. That is, once started, the `Player` will start playing from the current media time to the end of media once.
If the loop count is set to `N` where `N` is bigger than one, starting the `Player` will start playing the content from the current media time to the end of media. It will then loop back to the beginning of the content (media time zero) and play till the end of the media. The number of times it will loop to the beginning and play to the end of media will be `N-1`.
Setting the loop count to `0` is invalid. An `IllegalArgumentException` will be thrown.
Setting the loop count to `-1` will loop and play the content indefinitely.
If the `Player` is stopped before the preset loop count is reached either because `stop` is called, calling `start` again will resume the looping playback from where it was stopped until it fully reaches the preset loop count.
An *`END_OF_MEDIA`* event will be posted every time the `Player` reaches the end of media. If the `Player` loops back to the beginning and starts playing again because it has not completed the loop count, a *`STARTED`* event will be posted.

Parameters:
`count` - indicates the number of times the content will be played. `1` is the default. `0` is invalid. `-1` indicates looping indefintely.

Throws:
`IllegalArgumentException` - Thrown if the given count is invalid.
`IllegalStateException` - Thrown if the `Player` is in the *`STARTED`* or *`CLOSED`* state.

setMediaTime(long)

Declaration:
`public long` **`setMediaTime`**`(long now) throws MediaException`

Description:
Sets the `Player`'s *media time*.
For some media types, setting the media time may not be very accurate. The returned value will indicate the actual media time set.
Setting the media time to negative values will effectively set the media time to zero. Setting the media time to beyond the duration of the media will set the time to the end of media.
There are some media types that cannot support the setting of media time. Calling `setMediaTime` will throw a `MediaException` in those cases.

Parameters:
`now` - The new media time in microseconds.

Returns: The actual media time set in microseconds.

Throws:

`IllegalStateException` - Thrown if the `Player` is in the *`UNREALIZED`* or *`CLOSED`* state.
`MediaException` - Thrown if the media time cannot be set.

See Also: `getMediaTime()`

start()

Declaration:
`public void start() throws MediaException`

Description:
Starts the `Player` as soon as possible. If the `Player` was previously stopped by calling `stop`, it will resume playback from where it was previously stopped. If the `Player` has reached the end of media, calling `start` will automatically start the playback from the start of the media.
When `start` returns successfully, the `Player` must have been started and a `STARTED` event will be delivered to the registered `PlayerListeners`. However, the `Player` is not guaranteed to be in the *`STARTED`* state. The `Player` may have already stopped (in the *`PREFETCHED`* state) because the media has `0` or a very short duration.
If `start` is called when the `Player` is in the *`UNREALIZED`* or *`REALIZED`* state, it will implicitly call `prefetch`.
If `start` is called when the `Player` is in the *`STARTED`* state, the request will be ignored.

Throws:

`IllegalStateException` - Thrown if the `Player` is in the *`CLOSED`* state.
`MediaException` - Thrown if the `Player` cannot be started.
`SecurityException` - Thrown if the caller does not have security permission to start the `Player`.

stop()

Declaration:
`public void stop() throws MediaException`

Description:
Stops the `Player`. It will pause the playback at the current media time.
When `stop` returns, the `Player` is in the *`PREFETCHED`* state. A `STOPPED` event will be delivered to the registered `PlayerListeners`.
If `stop` is called on a stopped `Player`, the request is ignored.

Throws:

`IllegalStateException` - Thrown if the `Player` is in the *`CLOSED`* state.
`MediaException` - Thrown if the `Player` cannot be stopped.

javax.microedition.media
PlayerListener

Declaration

```
public interface PlayerListener
```

Description

`PlayerListener` is the interface for receiving asynchronous events generated by `Players`. Applications may implement this interface and register their implementations with the `addPlayerListener` method in `Player`.

A number of standard `Player` events are defined here in this interface. Event types are defined as strings to support extensibility as different implementations may introduce proprietary events by adding new event types. To avoid name conflicts, proprietary events should be named with the “reverse-domainname” convention. For example, a company named “mycompany” should name its proprietary event names with strings like `"com.mycompany.myEvent"` etc.

Applications that rely on proprietary events may not function properly across different implementations. In order to make the applications that use those events to behave well in environments that don’t implement them, `String.equals()` should be used to check the event.

Code fragment for catching standard events in `playerUpdate()`

```
if (eventType == PlayerListener.STARTED) {...}
```

Code fragment for catching proprietary events in `playerUpdate()`

```
if (eventType.equals("com.company.myEvent")) {...}
```

See Also: `Player`

Member Summary

Fields

static java.lang.String	CLOSED
static java.lang.String	DEVICE_AVAILABLE
static java.lang.String	DEVICE_UNAVAILABLE
static java.lang.String	DURATION_UPDATED
static java.lang.String	END_OF_MEDIA
static java.lang.String	ERROR
static java.lang.String	STARTED
static java.lang.String	STOPPED
static java.lang.String	VOLUME_CHANGED

Methods

void	playerUpdate(Player player, String event, Object eventData)

Fields

CLOSED

Declaration:
```
public static final String CLOSED
```

Description:
Posted when a `Player` is closed. When this event is received, the `eventData` parameter is null.
Value `"closed"` is assigned to `CLOSED`.

DEVICE_AVAILABLE

Declaration:
`public static final String` **`DEVICE_AVAILABLE`**

Description:
Posted when the system or another higher priority application has released an exclusive device which is now available to the `Player`.
The `Player` will be in the *`REALIZED`* state when this event is received. The application may acquire the device with the `prefetch` or `start` method.
A `DEVICE_UNAVAILABLE` event must preceed this event.
The `eventData` parameter is a `String` specifying the name of the device.
Value `"deviceAvailable"` is assigned to `DEVICE_AVAILABLE`.

DEVICE_UNAVAILABLE

Declaration:
`public static final String` **`DEVICE_UNAVAILABLE`**

Description:
Posted when the system or another higher priority application has temporarily taken control of an exclusive device which was previously available to the `Player`.
The `Player` will be in the *`REALIZED`* state when this event is received.
This event must be followed by either a `DEVICE_AVAILABLE` event when the device becomes available again, or an `ERROR` event if the device becomes permanently unavailable.
The `eventData` parameter is a `String` specifying the name of the device.
Value `"deviceUnavailable"` is assigned to `DEVICE_UNAVAILABLE`.

DURATION_UPDATED

Declaration:
`public static final String` **`DURATION_UPDATED`**

Description:
Posted when the duration of a `Player` is updated. This happens for some media types where the duration cannot be derived ahead of time. It can only be derived after the media is played for a period of time —- for example, when it reaches a key frame with duration info; or when it reaches the end of media.
When this event is received, the `eventData` parameter will be a `Long` object designating the duration of the media.
Value `"durationUpdated"` is assigned to `DURATION_UPDATED`.

END_OF_MEDIA

Declaration:
`public static final String` **`END_OF_MEDIA`**

Description:
Posted when a `Player` has reached the end of the media. When this event is received, the `eventData` parameter will be a `Long` object designating the media time when the `Player` reached end of media and stopped.
Value `"endOfMedia"` is assigned to `END_OF_MEDIA`.

ERROR

Declaration:
`public static final String` **`ERROR`**

Description:
Posted when an error had occurred. When this event is received, the `eventData` parameter will be a `String` object specifying the error message.

Value `"error"` is assigned to `ERROR`.

STARTED

Declaration:
`public static final String` **`STARTED`**

Description:
Posted when a `Player` is started. When this event is received, the `eventData` parameter will be a `Long` object designating the media time when the `Player` is started.
Value `"started"` is assigned to `STARTED`.

STOPPED

Declaration:
`public static final String` **`STOPPED`**

Description:
Posted when a `Player` stops in response to the `stop` method call. When this event is received, the `eventData` parameter will be a `Long` object designating the media time when the `Player` stopped.
Value `"stopped"` is assigned to `STOPPED`.

VOLUME_CHANGED

Declaration:
`public static final String` **`VOLUME_CHANGED`**

Description:
Posted when the volume of an audio device is changed. When this event is received, the `eventData` parameter will be a `VolumeControl` object. The new volume can be queried from the `VolumeControl`.
Value `"volumeChanged"` is assigned to `VOLUME_CHANGED`.

Methods

playerUpdate(Player, String, Object)

Declaration:
```
public void playerUpdate(javax.microedition.media.Player player, String event,
                  Object eventData)
```

Description:
This method is called to deliver an event to a registered listener when a `Player` event is observed.

Parameters:
`player` - The player which generated the event.
`event` - The event generated as defined by the enumerated types.
`eventData` - The associated event data.

11 Package javax.microedition.media.control

Description

This package defines the specific `Control` types that can be used with a `Player`. Like the `media` package, it is completely optional for IMP-NG.

Since: IMP-NG / MIDP 2.0

Class Summary	
Interfaces	
`ToneControl`	`ToneControl` is the interface to enable playback of a user-defined monotonic tone sequence.
`VolumeControl`	`VolumeControl` is an interface for manipulating the audio volume of a `Player`.

javax.microedition.media.control

ToneControl

Declaration

`public interface` **`ToneControl extends javax.microedition.media.Control`**

All Superinterfaces: `javax.microedition.media.Control`

Description

`ToneControl` is the interface to enable playback of a user-defined monotonic tone sequence.

A tone sequence is specified as a list of tone-duration pairs and user-defined sequence blocks. The list is packaged as an array of bytes. The `setSequence` method is used to input the sequence to the `ToneControl`.

The syntax of a tone sequence is described in Augmented BNF (http://www.ietf.org/rfc/rfc2234) notations:

```
sequence                     = version *1tempo_definition *1resolution_definition
                               *block_definition 1*sequence_event
version                      = VERSION version_number
VERSION                      = byte-value
version_number               = 1 ; version # 1
tempo_definition             = TEMPO tempo_modifier
TEMPO                        = byte-value
tempo_modifier               = byte-value
                     ; multiply by 4 to get the tempo (in bpm) used
                     ; in the sequence.
resolution_definition        = RESOLUTION resolution_unit
RESOLUTION = byte-value
resolution_unit              = byte-value
block_definition             = BLOCK_START block_number
                                 1*sequence_event
                               BLOCK_END block_number
BLOCK_START                  = byte-value
BLOCK_END                    = byte-value
block_number                 = byte-value
                     ; block_number specified in BLOCK_END has to be the
                     ; same as the one in BLOCK_START
sequence_event               = tone_event / block_event /
                               volume_event / repeat_event
tone_event                   = note duration
note                         = byte-value ; note to be played
duration                     = byte-value ; duration of the note
block_event                  = PLAY_BLOCK block_number
PLAY_BLOCK                   = byte-value
block_number                 = byte-value
                     ; block_number must be previously defined
                     ; by a full block_definition
volume_event                 = SET_VOLUME volume
SET_VOLUME                   = byte-value
Volume                       = byte-value ; new volume
repeat_event                 = REPEAT multiplier tone_event
REPEAT                       = byte-value
Multiplier                   = byte-value
                     ; number of times to repeat a tone
byte-value                   = -128 - 127
                     ; the value of each constant and additional
                     ; constraints on each parameter are specified below.
```

`VERSION`, `TEMPO`, `RESOLUTION`, `BLOCK_START`, `BLOCK_END`, `PLAY_BLOCK`, `SET_VOLUME`, `REPEAT` are pre-defined constants.

Following table shows the valid range of the parameters:

Parameter	Valid Range	Effective Range	Default
`tempo_modifier`	`5<= tempo_modifier <= 127`	20bpm to 508bpm	120bpm
`resolution_unit`	`1<= resolution_unit <= 127`	1/1 note to 1/127 note	1/64 note
`block_number`	`0<= block_number <= 127`	-	-
`note`	`0<= note <= 127 or SILENCE`	C-1 to G9 or rest	-
`duration`	`1<= duration <= 127`	-	-
`volume`	`0<= volume <= 100`	0% to 100% volume	100%
`multiplier`	`2<= multiplier <= 127`	-	-

The frequency of the note can be calculated from the following formula:

```
SEMITONE_CONST = 17.31234049066755 = 1/(ln(2^(1/12)))
note = ln(freq/8.176)*SEMITONE_CONST
```

The musical note A = note 69 (0x45) = 440 Hz.

Middle C (C4) and SILENCE are defined as constants.

The duration of each tone is measured in units of 1/resolution notes and tempo is specified in beats/minute, where 1 beat = 1/4 note. Because the range of positive values of `byte` is only `1 - 127`, the tempo is formed by multiplying the tempo modifier by 4. Very slow tempos are excluded so range of tempo modifiers is `5 - 127` providing an effective range of 20 - 508 bpm.

To compute the effective duration in milliseconds for a tone, the following formula can be used:

```
duration * 60 * 1000 * 4 / (resolution * tempo)
```

The following table lists some common durations in musical notes:

Note Length	Duration, Resolution=64	Duration, Resolution=96
`1/1`	`64`	`96`
`1/4`	`16`	`24`
`1/4 dotted`	`24`	`36`
`1/8`	`8`	`12`
`1/8 triplets`	`-`	`8`
`4/1`	`REPEAT 4 <note> 64`	`REPEAT 4 <note> 96`

Example:

```
// "Mary Had A Little Lamb" has "ABAC" structure.
// Use block to repeat "A" section.
byte tempo = 30; // set tempo to 120 bpm
byte d = 8; // eighth-note
byte C4 = ToneControl.C4;;
byte D4 = (byte)(C4 + 2); // a whole step
byte E4 = (byte)(C4 + 4); // a major third
byte G4 = (byte)(C4 + 7); // a fifth
byte rest = ToneControl.SILENCE; // rest
byte[] mySequence = {
    ToneControl.VERSION, 1, // version 1
    ToneControl.TEMPO, tempo, // set tempo
    ToneControl.BLOCK_START, 0, // start define "A" section
    E4,d, D4,d, C4,d, E4,d, // content of "A" section
    E4,d, E4,d, E4,d, rest,d,
    ToneControl.BLOCK_END, 0, // end define "A" section
    ToneControl.PLAY_BLOCK, 0, // play "A" section
    D4,d, D4,d, D4,d, rest,d, // play "B" section
    E4,d, G4,d, G4,d, rest,d,
    ToneControl.PLAY_BLOCK, 0, // repeat "A" section
    D4,d, D4,d, E4,d, D4,d, C4,d // play "C" section
};
```

```
try{
    Player p = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
    p.realize();
    ToneControl c = (ToneControl)p.getControl("ToneControl");
    c.setSequence(mySequence);
    p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```

Member Summary

Fields

static byte	BLOCK_END
static byte	BLOCK_START
static byte	C4
static byte	PLAY_BLOCK
static byte	REPEAT
static byte	RESOLUTION
static byte	SET_VOLUME
static byte	SILENCE
static byte	TEMPO
static byte	VERSION

Methods

void	setSequence(byte[] sequence)

Fields

BLOCK_END

Declaration:
`public static final byte` **`BLOCK_END`**

Description:
Defines an ending point for a block.
Value `-6` is assigned to `BLOCK_END`.

BLOCK_START

Declaration:
`public static final byte` **`BLOCK_START`**

Description:
Defines a starting point for a block.
Value `-5` is assigned to `BLOCK_START`.

C4

Declaration:
`public static final byte` **`C4`**

Description:
Middle C.
Value `60` is assigned to `C4`.

PLAY_BLOCK

Declaration:
`public static final byte` **`PLAY_BLOCK`**

Description:
Play a defined block.
Value `-7` is assigned to `PLAY_BLOCK`.

REPEAT

Declaration:
`public static final byte` **`REPEAT`**

Description:
The `REPEAT` event tag.
Value `-9` is assigned to `REPEAT`.

RESOLUTION

Declaration:
`public static final byte` **`RESOLUTION`**

Description:
The `RESOLUTION` event tag.
Value `-4` is assigned to `RESOLUTION`.

SET_VOLUME

Declaration:
`public static final byte` **`SET_VOLUME`**

Description:
The `SET_VOLUME` event tag.
Value `-8` is assigned to `SET_VOLUME`.

SILENCE

Declaration:
`public static final byte` **`SILENCE`**

Description:
Silence.
Value `-1` is assigned to `SILENCE`.

TEMPO

Declaration:
`public static final byte` **`TEMPO`**

Description:
The `TEMPO` event tag.
Value `-3` is assigned to `TEMPO`.

VERSION

Declaration:
`public static final byte` **`VERSION`**

Description:
The `VERSION` attribute tag.
Value `-2` is assigned to `VERSION`.

Methods

setSequence(byte[])

Declaration:
`public void` **`setSequence`**`(byte[] sequence)`

Description:
Sets the tone sequence.

Parameters:

`sequence` - The sequence to set.

Throws:

`IllegalArgumentException` - Thrown if the sequence is `null` or invalid.
`IllegalStateException` - Thrown if the `Player` that this control belongs to is in the *`PREFETCHED`* or *`STARTED`* state.

Package javax.microedition.media.control

javax.microedition.media.control VolumeControl

Declaration

`public interface` **`VolumeControl extends`** **`javax.microedition.media.Control`**

All Superinterfaces: `javax.microedition.media.Control`

Description

`VolumeControl` is an interface for manipulating the audio volume of a `Player`.

Volume Settings

This interface allows the output volume to be specified using an integer value that varies between `0` and `100`.

Specifying Volume in the Level Scale

The level scale specifies volume in a linear scale. It ranges from `0` to `100`, where `0` represents silence and `100` represents the highest volume. The mapping for producing a linear multiplicative value is implementation dependent.

Mute

Setting mute on or off doesn't change the volume level returned by `getLevel`. If mute is `true`, no audio signal is produced by this `Player`; if mute is `false` an audio signal is produced and the volume is restored.

Volume Change Events

When the state of the `VolumeControl` changes, a `VOLUME_CHANGED` event is delivered through the `PlayerListener`.

See Also: `javax.microedition.media.Control`, `javax.microedition.media.Player`, `javax.microedition.media.PlayerListener`

Member Summary

Methods

`int`	`getLevel()`
`Boolean`	`isMuted()`
`int`	`setLevel(int level)`
`void`	`setMute(boolean mute)`

Methods

getLevel()

Declaration:
`public int` **`getLevel`**`()`

Description:
Get the current volume level set.
`getLevel` may return `-1` if and only if the `Player` is in the *`REALIZED`* state (the audio device has not been initialized) and `setLevel` has not yet been called.

Returns: The current volume level or `-1`.

See Also: `setLevel(int)`

isMuted()

Declaration:
`public boolean isMuted()`

Description:
Get the mute state of the signal associated with this `VolumeControl`.

Returns: The mute state.

See Also: `setMute(boolean)`

setLevel(int)

Declaration:
`public int setLevel(int level)`

Description:
Set the volume using a linear point scale with values between `0` and `100`.
`0` is silence; `100` is the loudest useful level that this `VolumeControl` supports. If the given level is less than `0` or greater than `100`, the level will be set to `0` or `100` respectively.
When `setLevel` results in a change in the volume level, a `VOLUME_CHANGED` event will be delivered through the `PlayerListener`.

Parameters:
`level` - The new volume specified in the level scale.

Returns: The level that was actually set.

See Also: `getLevel()`

setMute(boolean)

Declaration:
`public void setMute(boolean mute)`

Description:
Mute or unmute the `Player` associated with this `VolumeControl`.
Calling `setMute(true)` on the `Player` that is already muted is ignored, as is calling `setMute(false)` on the `Player` that is not currently muted. Setting mute on or off doesn't change the volume level returned by getLevel.
When `setMute` results in a change in the muted state, a `VOLUME_CHANGED` event will be delivered through the `PlayerListener`.

Parameters:
`mute` - Specify `true` to mute the signal, `false` to unmute the signal.

See Also: `isMuted()`

12 Package javax.microedition.midlet

Description

The `MIDlet` package defines IMP-NG applications and the interactions between the application and the environment in which the application runs. An application of the IMP-NG is called an IMlet, as has been for the Information Module Profile (JSR-195). Technically, applications of IMP as well as IMP-NG are MIDlets, as defined in the class `javax.microedition.midlet.MIDlet` in this specification. The reason is – as it was for IMP – that this part of the specification has been taken unchanged from the Mobile Information Device Profile Specification, version 2.0 (JSR-118).

Applications

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. This application model is also valid for IMP and IMP-NG, because the conditions of limited resources etc. are very similar.

The application model defines how multiple IMlets 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.

IMP-NG IMlet Suite

An IMP-NG application MUST use only functionality specified by the IMP-NG specification as it is developed, tested, deployed, and run.

The elements of an IMlet suite are:

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

Each device is presumed to implement the functions required by its users to install, select, run, and remove IMlets. The term application management software is used to refer collectively to these device specific functions. The application management software provides an environment in which the IMlet is installed, started, stopped, and uninstalled. It is responsible for handling errors during the installation, execution, and removal of IMlet suites and interacting with the user as needed. It provides to the IMlet(s) the Java runtime environment required by the IMP-NG Specification.

One or more IMlets MAY be packaged in a single JAR file. Each IMlet consists of a class that extends the `javax.microedition.midlet.MIDlet` class and other classes as may be needed by the IMlet. The manifest in the JAR file contains attributes that are used during installation and execution of IMlets. The IMlet is the entity that is launched by the application management software. When an IMlet suite is invoked, a Java Virtual Machine is needed on which the classes can be executed. A new instance of the IMlet is created by the application management software and used to direct the IMlet to start, pause, and destroy itself.

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

Package javax.microedition.midlet

IMlet Suite Security

The MIDP 1.0 specification constrained each MIDlet suite to operate in a sandbox wherein all of the APIs available to the MIDlets would prevent access to sensitive functions of the device. IMP used the same sandbox model for its IMlet suites. The sandbox concept is also used in this specification (as it has been used in the MIDP 2.0 specification) and all untrusted IMlet suites are subject to its limitations. Every implementation of this specification MUST support running untrusted IMlet suites.

MIDP 2.0 introduces the concept of trusted applications that may be permitted to use APIs that are considered sensitive and are restricted. As described above, the trusted application concept is used also in IMP-NG – with some simplifications originated in the character of information modules. If and when a device determines that an IMlet suite can be trusted the device allows access as indicated by the policy. The chapter **Error! Reference source not found.**. **Error! Reference source not found.** describes the concepts and capabilities of untrusted and trusted applications.

IMP-NG Execution Environment

The IMP-NG defines the execution environment provided to IMlets. The execution environment is shared by all IMlets within an IMlet suite, and any IMlet can interact with other IMlets packaged together. The application management software initiates the applications and makes the following available to the IMlet:

- Classes and native code that implement the CLDC, including a Java Virtual Machine
- Classes and native code that implement the IMP-NG 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, when it is present
- Any other APIs available on the device such as implementations of additional JSRs, Licensee Open Classes, Optional Packages, etc.

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. IMlet Suites MUST NOT contain classes that are in packages defined by the CLDC or IMP-NG.

IMP-NG provides the classes that implement the IMP-NG APIs. The implementation MUST ensure that the application programmer cannot override, modify, or add any classes to these protected system packages.

A single JAR file contains all of the IMlet's classes. The IMlet may load and invoke methods from any class in the JAR file, in the IMP-NG, or in the CLDC. All of the classes within these three scopes are shared in the execution environment of the IMlets from the JAR file. All states accessible via those classes are available to any Java class running on behalf of the IMlet. There is a single space containing the objects of all IMlets, MIDP, and CLDC in use by the IMlet 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 IMlet should use it to run safely in a multi-threaded environment.

The class files of the IMlet 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 `java.lang.Class.getResourceAsStream`. For example, the manifest would be available in this manner.

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

Package javax.microedition.midlet

IMlet Suite Packaging

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

- A manifest describing the contents
- Java classes for the IMlet(s) and classes shared by the IMlets
- Resource files used by the IMlet(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 IMlet suite and as defaults for attributes not found in the application descriptor. The attributes are defined for use in both the manifest and the optional application descriptor.

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

IMlet Attributes

Attribute Name	**Attribute Description**
`MIDlet-Name`	The name of the IMlet suite that identifies the IMlets to the user.
`MIDlet-Version`	The version number of the IMlet 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.
`MIDlet-Vendor`	The organization that provides the IMlet suite.
`MIDlet-Icon`[2]	The case-sensitive absolute name of a PNG file within the JAR used to represent the IMlet suite. Due to the different UI capabilities of IMs it MAY be empty and MAY be ignored. If it is not ignored, the information behind the attribute can naturally not be used to display the icon. This is because IMs by definition do not have the capabilities to do this.
`MIDlet-Description`	The description of the IMlet suite.
`MIDlet-Info-URL`	A URL for information further describing the IMlet suite. The syntax and meaning MUST conform to RFC2396 and RFCs that define each scheme.
`MIDlet-<n>`	The name, icon, and class of the nth IMlet in the JAR file separated by a comma. The lowest value of `<n>` MUST be `1` and consecutive ordinals MUST be used. Leading and trailing spaces in name, icon and class are ignored. Name is used to identify this IMlet to the user. The name must be present and be non-null. Icon MAY be empty and MAY be ignored. Class is the name of the class extending the `javax.microedition.midlet.MIDlet` class for the nth IMlet. The classname MUST be non-null and contain only characters for Java class names. The class MUST have a public no-args constructor. The class name IS case sensitive.
`MIDlet-Jar-URL`	The URL from which the JAR file can be loaded. The syntax and meaning MUST conform to RFC2396 and RFCs that define each scheme. Both absolute and relative URLs MUST be supported. The context for a relative URL is the URL from which this application descriptor was 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 IMlet. The device may provide additional storage according to its own policy. The default is zero.
`MicroEdition-Profile`	The J2ME profiles required, using the same format and value as the System

[1] The Java™ Product Versioning Specification: http://java.sun.com/products/jdk/1.2/docs/guide/versioning/spec/VersioningSpecification.html
[2] This attribute exists mainly for compatibility reasons with MIDP 2.0.

	property `microedition.profiles` (for example `"IMP-NG"`). The device must implement *all* of the profiles listed. If any of the profiles are not implemented the installation MUST fail. Multiple profiles are separated with a blank (Unicode x20).
`MicroEdition-Configuration`	The J2ME Configuration required using the same format and value as the System property `microedition.configuration` (for example `"CLDC-1.0"`).
`MIDlet-Permissions`	Zero or more permissions that are critical to the function of the IMlet suite. See chapter **Error! Reference source not found.**. **Error! Reference source not found.** for details of usage.
`MIDlet-Permissions-Opt`	Zero or more permissions that are non-critical to the function of the IMlet suite. See chapter **Error! Reference source not found.**. **Error! Reference source not found.** for details of usage.
`MIDlet-Push-<n>`	Register an IMlet to handle inbound connections. Refer to `javax.microedition.io.PushRegistry` for details.
`MIDlet-Install-Notify`	Refer to chapter **Error! Reference source not found.**. **Error! Reference source not found.** for details.
`MIDlet-Delete-Notify`	Refer to chapter **Error! Reference source not found.**. **Error! Reference source not found.** for details.
`MIDlet-Delete-Confirm`	Refer to chapter **Error! Reference source not found.**. **Error! Reference source not found.** for details.

Some attributes use multiple values, for those attributes the values are separated by a comma (Unicode U+002C) except where noted. Leading and trailing whitespace (Unicode U+0020) and tab (Unicode U+0009) are ignored on each value.

Version Numbering

Version numbers have 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). Version numbers are described in the the Java(TM) Product Versioning Specification http://java.sun.com/products/jdk/1.2/docs/guide/versioning/spec/VersioningSpecification.html.

For example, 1.0.0 can be used to specify the first version of an IMlet 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 IMlet suite.

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. IMP-NG implementations MUST implement handling of lines longer than 72 bytes as defined in the manifest specification. Manifest attributes are passed to the IMlet when requested using the `MIDlet.getAppProperty` method, unless the attribute is duplicated in the application descriptor, for handling of duplicate attributes see the “Application Descriptor” section.

The manifest MUST contain the following attributes:

- `MIDlet-Name`
- `MIDlet-Version`
- `MIDlet-Vendor`

The manifest or the application descriptor MUST contain the following attributes:

- `MIDlet-<n>` for each IMlet
- `MicroEdition-Profile`

- `MicroEdition-Configuration`

The manifest MAY contain the following:

- `MIDlet-Description`
- `MIDlet-Icon` (MAY be ignored by IMP-NG)
- `MIDlet-Info-URL`
- `MIDlet-Data-Size`
- `MIDlet-Permissions`
- `MIDlet-Permissions-Opt`
- `MIDlet-Push-<n>`
- `MIDlet-Install-Notify`
- `MIDlet-Delete-Notify`
- `MIDlet-Delete-Confirm`
- Any application-specific attributes that do not begin with `MIDlet-` or `MicroEdition-`

For example, a manifest for a hypothetical suite of Weather station applications would look like the following example:

```
MIDlet-Name: WeatherStation
MIDlet-Version: 1.1.9
MIDlet-Vendor: WeatherCorp
MIDlet-1: Wind, , com.weathercorp.Wind
MIDlet-2: Temperature, , com.weathercorp.Temp
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.0
```

MIDlet Classes

All Java classes needed by the IMlet 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`.

Application Descriptor

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

To allow devices to dispatch an application descriptor to the IMP-NG application management software, a file extension and MIME type (http://www.iana.org/assignments/media-types/text/vnd.sun.j2me.app-descriptor) are registered with the IANA:

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

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

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-<n>` for each IMlet
- `MicroEdition-Profile`
- `MicroEdition-Configuration`
- `MIDlet-Description`
- `MIDlet-Icon` (MAY be ignored by IMP-NG)
- `MIDlet-Info-URL`
- `MIDlet-Data-Size`
- `MIDlet-Permissions`
- `MIDlet-Permissions-Opt`
- `MIDlet-Push-<n>`
- `MIDlet-Install-Notify`
- `MIDlet-Delete-Notify`
- `MIDlet-Delete-Confirm`
- Any application-specific attributes that do not begin with `MIDlet-` or `MicroEdition-`

The mandatory attributes `MIDlet-Name`, `MIDlet-Version`, and `MIDlet-Vendor` MUST be duplicated in the descriptor and manifest files since they uniquely identify the application. If they are not identical (not from the same application), then the JAR MUST NOT be installed.

Duplication of other manifest attributes in the application descriptor is not required and their values MAY differ even though both the manifest and descriptor files contain the same attribute for untrusted IMlet suites. If the IMlet suite is not trusted the value from the descriptor file will override the value from the manifest file. If the IMlet suite is trusted then the values in the application descriptor MUST be identical to the corresponding attribute values in the Manifest.

IMlets MUST NOT add any attributes to the manifest or the Application Descriptor that start with `MIDlet-` or `MicroEdition-` other than those defined in the relevant Configuration and Profiles (e.g. CLDC and IMP-NG) specifications. Unrecognized attributes MUST be ignored by the AMS.

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 converted to Unicode before parsing, using the rules below. For example, an ISO-8859-1 encoded file would need to be read through the equivalent of `java.io.InputStreamReader` with the appropriate encoding. The default character encoding for transporting a descriptor is UTF-8. 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 convert the stream to UCS-2.

BNF for Parsing Application Descriptors

```
appldesc:       *attrline
attrline:       attrname ":" [WSP] attrvalue [WSP] newlines
attrname:       1*<any Unicode char except CTLs or separators>
attrvalue:      *valuechar | valuechar *(valuechar | WSP) valuechar
valuechar:      <any valid Unicode character, excluding CTLS and WSP>
newlines        = 1*newline ; allow blank lines to be ignored
```

```
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 Weather Station applications would look like the following example:

```
MIDlet-Name: WeatherStation
MIDlet-Version: 1.1.9
MIDlet-Vendor: WeatherCorp
MIDlet-1: Wind, , com.weathercorp.Wind
MIDlet-2: Temperature, , com.weathercorp.Temp
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.0
MIDlet-Description: Really cool weather station
MIDlet-Jar-URL: http://www.weathercorp.com/appls/weatherStation.jar
MIDlet-Jar-Size: 6512
MIDlet-Data-Size: 512
```

Application Lifecycle

Each IMlet MUST extend `the javax.microedition.midlet.MIDlet` class. The `MIDlet` class allows for the orderly starting, stopping, and cleanup of the IMlet. The IMlet can request the arguments from the application descriptor to communicate with the application management software. An IMlet suite SHOULD 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 an IMlet.

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

When the IMlet is run, an instance of the IMlet's primary class is created using its public no-argument constructor, and the methods of the `MIDlet` class are called to sequence the IMlet through its various states. The IMlet can either request changes in state or notify the application management software of state changes via the `MIDlet` methods. When the IMlet 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 IMlet MUST NOT call `System.exit`, which will throw a `SecurityException` when called by an IMlet.

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.

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

IMlet lifecycle

The IMlet lifecycle defines the protocol between an IMlet and its environment through the following:

- A simple well-defined state machine
- A concise definition of the IMlet's states
- APIs to signal changes between the states

Package javax.microedition.midlet

IMlet Lifecycle Definitions

The following definitions are used in the IMlet lifecycle:

- **application management software -** a part of the device's software operating environment that manages IMlets. It maintains the IMlet state and directs the IMlet through state changes.
- **IMlet** – an IMP-NG application on the device. The IMlet can signal the application management software about whether is it wants to run or has completed. An IMlet has no knowledge of other IMlets through the IMlet API.
- **IMlet States** - the states an IMlet can have are defined by the transitions allowable through the `MIDlet` class interface. More specific application states are known only to the application.

IMlet States

The IMlet 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.

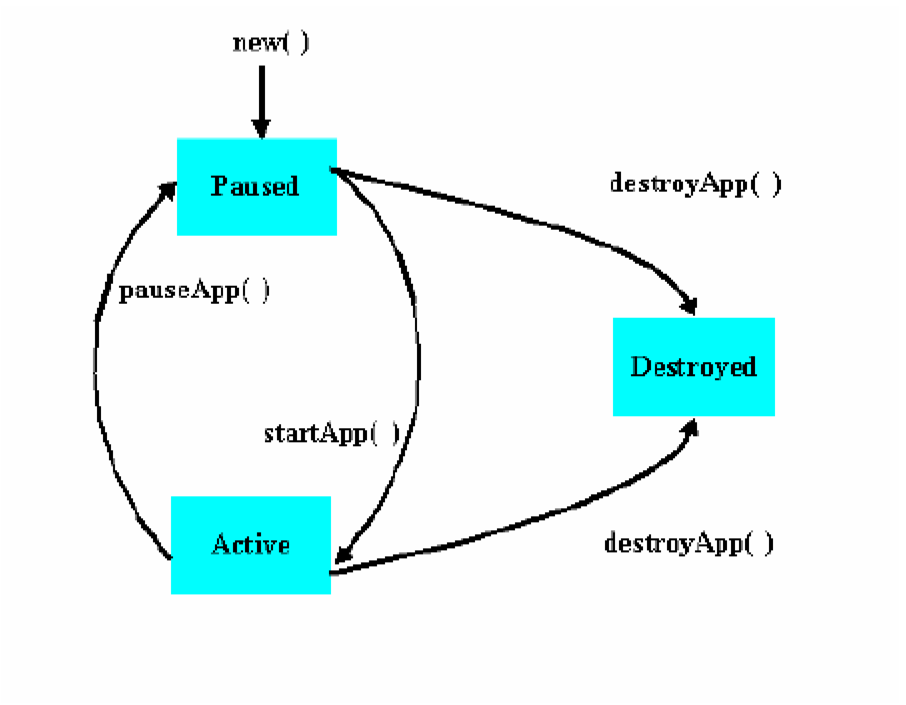
The valid states for IMlets are:

State Name	Description
`Paused`	The IMlet is initialized and is quiescent. It SHOULD not be holding or using any shared resources. This state is entered: • After the IMlet has been created using `new`. The public no-argument constructor for the IMlet 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 is called from the AMS and returns successfully. • From the `Active` state when the `MIDlet.notifyPaused()` method returns successfully to the IMlet. • From the `Active` state if `startApp` throws an `MIDletStateChangeException`.
`Active`	The IMlet is functioning normally. This state is entered: • Just prior to the AMS calling the `MIDlet.startApp()` method.
`Destroyed`	The IMlet has released all of its resources and terminated. This state is entered: • When the AMS called the `MIDlet.destroyApp()` method and returns successfully, 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 IMlet must have performed the equivalent of the `MIDlet.destroyApp()` method before calling `MIDlet.notifyDestroyed()`. **Note:** This state is only entered once

.

Package javax.microedition.midlet

The states and transitions for an IMlet are:



MIDlet Lifecycle Model

A typical sequence of IMlet execution is:

Application Management Software	IMlet
The application management software creates a new instance of an IMlet.	The default (no argument) constructor for the IMlet is called; it is in the `Paused` state.
The application management software has decided that it is an appropriate time for the IMlet to run, so it calls the `MIDlet.startApp` method for it to enter the `Active` state.	The IMlet acquires any resources it needs and begins to perform its service.
The application management software wants the IMlet to significantly reduce the amount of resources it is consuming, so that they may temporarily be used by other functions on the device such as a phone call or running another IMlet. The AMS will signal this request to the IMlet by calling the `MIDlet.pauseApp` method. The IMlet should then reduce its resource consumption as much as possible.	The IMlet stops performing its service and might choose to release some resources it currently holds.
The application management software has determined that the IMlet is no longer needed, or perhaps needs to make room for a higher priority application in memory, so it signals the IMlet that it is a candidate to be destroyed by calling the `MIDlet.destroyApp` method.	If it has been designed to do so, the IMlet saves state or user preferences and performs clean up.

MIDlet Interface

- **`pauseApp`** - the IMlet SHOULD release any temporary resources and become passive
- **`startApp`** - the IMlet SHOULD acquire any resources it needs and resume
- **`destroyApp`** - the IMlet SHOULD save any state and release all resources

- **`notifyDestroyed`** - the IMlet notifies the application management software that it has cleaned up and is done
- **`notifyPaused`** - the IMlet notifies the application management software that it has paused
- **`resumeRequest`** - the IMlet asks application management software to be started again
- **`getAppProperty`** - gets a named property from the IMlet

Application Implementation Notes

The application SHOULD take measures to avoid race conditions in the execution of the IMlet's methods. Each method may need to synchronize itself with the other methods avoid concurrency problems during state changes.

Example IMlet Application

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

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

/**
* An example IMlet 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
    * running.
    */
    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() {
        }
}
```

Since: IMP / MIDP 1.0

Class Summary

Classes	
`MIDlet`	An IMlet (an IMP-NG application) has to be an instance of a class derived from `MIDlet`.
Exceptions	
`MIDletStateChangeException`	Signals that a requested IMlet state change failed.

javax.microedition.midlet
MIDlet

Declaration

```
public abstract class MIDlet

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

Description

An IMlet is an IMP / IMP-NG application. In order to define IMP-NG as a subset of MIDP 2.0 (as IMP has been defined as a subset of MIDP 1.0), the application must extend this class `MIDlet` to allow the application management software to control the IMlet and to be able to retrieve properties from the application descriptor and notify and request state changes. The methods of the class `MIDlet` allow the application management software to create, start, pause, and destroy an IMlet. An IMlet 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 IMlets within a runtime environment. It can select which IMlets are active at a given time by starting and pausing them individually. The application management software maintains the state of the IMlet and invokes methods on the IMlet to notify the IMlet of change states.

The IMlet implements these methods to update its internal activities and resource usage as directed by the application management software. The IMlet 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	
int	checkPermission(String permission)
protected abstract void	destroyApp(boolean unconditional)
java.lang.String	getAppProperty(String key)
void	notifyDestroyed()
void	notifyPaused()
protected abstract void	pauseApp()
boolean	platformRequest(String URL)
void	resumeRequest()
protected abstract void	startApp()

Inherited Member Summary
Methods inherited from class `Object`
equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()

Constructors

MIDlet()

Declaration:
`protected MIDlet()`

Description:
Protected constructor for subclasses. The application management software is responsible for creating IMlets and creation of IMlets is restricted. IMlets should not attempt to create other IMlets.

Throws:
`SecurityException` – unless the application management software is creating the IMlet

---

Methods

checkPermission(String)

Declaration:
```
public final int checkPermission(String permission)
```

Description:
Get the status of the specified permission. If no API on the device defines the specific permission requested then it must be reported as denied. If the status of the permission is not known because it might require a user interaction then it should be reported as unknown.

Parameters:
`permission` - to check if denied, allowed, or unknown.

Returns: `0` if the permission is denied; `1` if the permission is allowed; `-1` if the status is unknown

destroyApp(boolean)

Declaration:
```
protected abstract void destroyApp(boolean unconditional)
                  throws MIDletStateChangeException
```

Description:
Signals the IMlet to terminate and enter the *`Destroyed`* state. In the *`destroyed`* state the IMlet must release all resources and save any persistent state. This method may be called from the *`Paused`* or *`Active`* states.
IMlets should perform any operations required before being terminated, such as releasing resources or saving preferences or state.
Note: The IMlet can request that it not enter the *`Destroyed`* state by throwing an `MIDletStateChangeException`. This is only a valid response if the `unconditional` flag is set to `false`. If it is `true` the IMlet is assumed to be in the *`Destroyed`* state regardless of how this method terminates. If it is not an unconditional request, the IMlet 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 IMlet is put into the *`Destroyed`* state.

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

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

getAppProperty(String)

Declaration:
```
public final String getAppProperty(String key)
```

Description:
Provides an IMlet 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. For trusted applications the values in the manifest MUST NOT be overridden by those in the application descriptor. If they differ, the IMlet will not be installed on the device. For untrusted applications, if an attribute 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:
`NullPointerException` - is thrown if key is `null`.

notifyDestroyed()

Declaration:
`public final void` **`notifyDestroyed`**`()`

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

notifyPaused()

Declaration:
`public final void` **`notifyPaused`**`()`

Description:
Notifies the application management software that the IMlet does not want to be active and has entered the *`Paused`* state. Invoking this method will have no effect if the IMlet is destroyed, or if it has not yet been started.
It may be invoked by the IMlet when it is in the *`Active`* state.
If an IMlet 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.
If the application pauses itself it will need to call `resumeRequest` to request to reenter the *`active`* state.

pauseApp()

Declaration:
`protected abstract void` **`pauseApp`**`()`

Description:
Signals the IMlet to enter the *`Paused`* state. In the *`Paused`* state the IMlet must release shared resources and become quiescent. This method will only be called called when the IMlet is in the *`Active`* state.
If a Runtime exception occurs during `pauseApp` the IMlet will be destroyed immediately. Its `destroyApp` will be called allowing the IMlet to cleanup.

platformRequest(String)

Declaration:
`public final boolean` **`platformRequest`**`(String URL) throws ConnectionNotFoundException`

Description:
Requests that the device handle (for example, install or dial) the indicated URL.

If the platform has the appropriate capabilities and resources available, it SHOULD handle the plarform request, while keeping the IMlet suite running. If the platform does not have appropriate capabilities or resources available, it MAY wait to handle the URL request until after the IMlet suite exits.
This is a non-blocking method. In addition, this method does NOT queue multiple requests. On platforms where the IMlet suite must exit before the request is handled, the platform MUST handle only the last request made. On platforms where the IMlet suite and the request can be handled concurrently, each request that the IMlet suite makes MUST be passed to the platform software for handling in a timely fashion.
Devices MAY choose to support several URL schemes, there are no schemes to be supported mandatory, though. In this sense, the following two paragraphs should be interpreted as examples:
If the URL specified refers to an IMlet suite (either an Application Descriptor or a JAR file), the application handling the request MAY interpret it as a request to install the named package. In case of an interpretation as install request, the platform's normal IMlet suite installation process SHOULD be used, handling the request as an "internal trigger". If the IMlet suite being installed is an update of the currently running IMlet suite, the platform MUST first stop the currently running IMlet suite before performing the update. On some platforms, the currently running IMlet suite MAY need to be stopped before any installations can occur.
If the URL specified is of the form tel:<number>, as specified in RFC2806 (http://rfc.net/rfc2806.html), then the platform MAY interpret this as a request to initiate a voice call. Such an interpretation is only useful, if a "phone" application exists on the device. If so, the request MUST be passed to the "phone" application to handle. The "phone" application, if present, MUST be able to set up local and global phone calls and also perform DTMF post dialing. Not all elements of RFC2806 need be implemented, especially the area-specifier or any other requirement on the terminal to know its context. The isdn-subaddress, service-provider and future-extension may also be ignored. Pauses during dialing are not relevant in some telephony services.
Many of the ways this method will be used could have a financial impact to the user (e.g. transferring data through a wireless network, or initiating a voice call). Whatever action the content of the URL parameter implies, before performing it, necessary permissions for the caller MUST be checked. Note that even actions with financial impact take place without user confirmation if the security policy allows doing so. This is an important difference towards MIDP 2.0, where such actions always get an interactive confirmation from the user

Parameters:
`URL` - The URL for the platform to load. An empty string (not null) cancels any pending requests.

Returns: true if the IMlet suite MUST first exit before the content can be fetched.

Throws:
`java.microedition.io.ConnectionNotFoundException` - if the platform cannot handle the URL requested.

Since: IMP-NG / MIDP 2.0

resumeRequest()

Declaration:
```
public final void resumeRequest()
```

Description:
Provides an IMlet 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()

Declaration:
```
protected abstract void startApp() throws MIDletStateChangeException
```

Description:
Signals the IMlet that it has entered the *`Active`* state. In the *`Active`* state the IMlet may hold resources. The method will only be called when the IMlet 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 `RuntimeException` occurs during `startApp` the IMlet will be destroyed immediately. Its `destroyApp` will be called allowing the IMlet to cleanup.

Throws:
`MIDletStateChangeException` - is thrown if the IMlet cannot start now but might be able to start at a later time.

javax.microedition.midlet MIDletStateChangeException

Declaration

```
public class MIDletStateChangeException extends Exception

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

Description

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

Since: IMP / MIDP 1.0

See Also: `MIDlet`

Member Summary

Constructors

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

Inherited Member Summary

Methods inherited from class `Object`

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

Methods inherited from class `Throwable`

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

Constructors

MIDletStateChangeException()

Declaration:
`public MIDletStateChangeException()`

Description:
Constructs an exception with no specified detail message.

MIDletStateChangeException(String)

Declaration:
`public MIDletStateChangeException(String s)`

Description:
Constructs an exception with the specified detail message.

Parameters:
`s` - the detail message

13 Package javax.microedition.pki

Description

Certificates are used to authenticate information for secure Connections. The `Certificate` interface provides to the application information about the origin and type of the certificate. The `CertificateException` provides information about failures that may occur while verifying or using certificates.

The chapter **Error! Reference source not found.**. **Error! Reference source not found.** defines the format and usage of certificates. X.509 Certificates MUST be supported. Other certificate formats MAY be supported. The implementation MAY store only the essential information from certificates. Internally, the fields of the certificate MAY be stored in any format that is suitable for the implementation.

Requirements

Like MIDP 2.0 devices, IMP-NG devices are expected to operate using standard Internet and wireless protocols and techniques for transport and security. The current mechanisms for securing Internet content is based on existing Internet standards for public key cryptography.

IMP-NG X.509 Certificate Profile

Devices MUST conform to all mandatory requirements in and SHOULD conform to all optional requirements except those requirements in excluded sections listed below. Mandatory and optional requirements are listed in Appendix C of [WAPCert]. Additional requirements, ON TOP of those listed in [WAPCert] are given below.

- Excluding [WAPCert] Section 6.2, User Certificates for Authentication
- Excluding [WAPCert] Section 6.3, User Certificates for Digital Signatures RFC2459 contains sections which are not relevant to implementations of this specification. TheWAP Certificate Profile does not mention these functions. The sections to be excluded are:
- Exclude the requirements from Paragraphs 4 of Section 4.2 - Standard Certificate Extensions. A conforming implementation of this specification does not need to recognize extensions that must or may be critical including certificate policies, name constraints, and policy constraints.
- Exclude RFC2459 Section 6.2 Extending Path Validation. Support for Policy Certificate Authority or policy attributes is not required.

Certificate Extensions

A version 1 X.509 certificate MUST be considered equivalent to a version 3 certificate with no extensions. At a minimum, a device conforming to this profile MUST recognize key usage (see RFC2459 sec. 4.2.1.3), basic constraints (see RFC2459 sec. 4.2.1.10).

Although a conforming device may not recognize the authority and subject key identifier (see RFC2459 sec. 4.2.1.1 and 4.2.1.2) extensions it MUST support certificate authorities that sign certificates using the same distinguished name but using multiple public keys.

Implementations MUST be able to process certificates with unknown distinguished name attributes.

Implementations MUST be able to process certificates with unknown, non-critical certificate extensions.

The `serialNumber` attribute defined by [WAPCert] must be recognized in distinguished names for Issuer and Subject.

Certificate Size

Devices must be able to process certificates that are not self-signed root CA certificates of size up to at least 1500 bytes.

Package javax.microedition.pki

Algorithm Support

A device MUST support the RSA signature algorithm with the SHA-1 hash function `sha1WithRSAEncryption` as defined by PKCS #1 [RFC2437]. Devices that support these algorithms MUST be capable of verifying signatures made with RSA keys of length up to and including 2048 bits.

Devices SHOULD support signature algorithms `md2WithRSAEncryption` and `md5WithRSAEncryption` as defined in [RFC2437]. Devices that support these algorithms MUST be capable of verifying signatures made with RSA keys of length up to and including 2048 bits.

Certificate Processing for HTTPS

Devices MUST recognize the extended key usage extension defined of RFC2818 if it is present and is marked critical and when present MUST verify that the extension contains the `id-kp-serverAuth` object identifier (see RFC2459 sec. 4.2.1.13).

SSL and TLS allow the web server to include the redundant root certificate in the server certificate message. In practice this certificate may not have the basic constraint extension (it is most likely a version 1 certificate), a device MUST ignore the redundant certificate in this case. Web servers SHOULD NOT include a self-signed root CA in a certificate chain.

Since: IMP-NG / MIDP 2.0

Class Summary

Interfaces	
`Certificate`	Interface common to certificates.
Exceptions	
`CertificateException`	The `CertificateException` encapsulates an error that occurred while a `Certificate` is being used.

javax.microedition.pki Certificate

Declaration

`public interface` **`Certificate`**

Description

Interface common to certificates. The features abstracted of `Certificates` include subject, issuer, type, version, serial number, signing algorithm, dates of valid use, and serial number.

Printable Representation for Binary Values

A non-string values in a certificate are represented as strings with each byte as two hex digits (capital letters for A-F) separated by `":"` (Unicode `0x3A`).

For example: `0C:56:FA:80`

Printable Representation for X.509 Distinguished Names

For a X.509 certificate the value returned is the printable verision of the distingished name (DN) from the certificate.

An X.509 distinguished name of is set of attributes, each attribute is a sequence of an object ID and a value. For string comparison purposes, the following rules define a strict printable representation.

1. There is no added white space around separators.
2. Attributes are not reordered.
3. If an object ID is in the table below, the label from the table will be substituted for the object ID, else the ID is formatted as a string using the binary printable representation above.
4. Each object ID or label and value within an attribute will be separated by a `"="` (Unicode `0x3D`), even if the value is empty.
5. If value is not a string, then it is formatted as a string using the binary printable representation above.
6. Attributes will be separated by a `";"` (Unicode `0x3B`)

Labels for X.500 Distinguished Name Attributes

Object ID	Binary	Label
`id-at-commonName`	`55:04:03`	CN
`id-at-surname`	`55:04:04`	SN
`id-at-countryName`	`55:04:06`	C
`id-at-localityName`	`55:04:07`	L
`id-at-stateOrProvinceName`	`55:04:08`	ST
`id-at-streetAddress`	`55:04:09`	STREET
`id-at-organizationName`	`55:04:0A`	O
`id-at-organizationUnitName`	`55:04:0B`	OU
`emailAddress`	`2A:86:48:86:F7:0D:01:09:01`	EmailAddress

Example of a printable distinguished name:

```
C=US;O=Any Company, Inc.;CN=www.anycompany.com
```

Since: IMP-NG / MIDP 2.0

Member Summary

Methods

`java.lang.String`	`getIssuer()`
`long`	`getNotAfter()`
`long`	`getNotBefore()`
`java.lang.String`	`getSerialNumber()`
`java.lang.String`	`getSigAlgName()`
`java.lang.String`	`getSubject()`
`java.lang.String`	`getType()`
`java.lang.String`	`getVersion()`

Methods

getIssuer()

Declaration:
`public String getIssuer()`

Description:
Gets the name of this certificate's issuer.

Returns: The issuer of the `Certificate`; the value MUST NOT be `null`.

getNotAfter()

Declaration:
`public long getNotAfter()`

Description:
Gets the time after which this `Certificate` may not be used from the validity period.

Returns: The time in milliseconds after which the `Certificate` is not valid (expiration date); it MUST be positive; `Long.MAX_VALUE` is returned if the certificate does not have its validity restricted based on the time.

getNotBefore()

Declaration:
`public long getNotBefore()`

Description:
Gets the time before which this `Certificate` may not be used from the validity period.

Returns: The time in milliseconds before which the `Certificate` is not valid; it MUST be positive, `0` is returned if the certificate does not have its validity restricted based on the time.

getSerialNumber()

Declaration:
`public String getSerialNumber()`

Description:
Gets the printable form of the serial number of this `Certificate`. If the serial number within the certificate is binary it should be formatted as a string using the binary printable representation in class description. For example, `0C:56:FA:80`.

Returns: A string containing the serial number in user-friendly form; `null` is returned if there is no serial number.

getSigAlgName()

Declaration:
`public String getSigAlgName()`

Description:
Gets the name of the algorithm used to sign the `Certificate`. The algorithm names returned should be the labels defined in RFC2459 Section 7.2.

Returns: The name of signature algorithm; the value MUST NOT be `null`.

getSubject()

Declaration:
`public String getSubject()`

Description:
Gets the name of this certificate's subject.

Returns: The subject of this `Certificate`; the value MUST NOT be `null`.

getType()

Declaration:
`public String getType()`

Description:
Get the type of the `Certificate`. For X.509 Certificates the value returned is `"X.509"`.

Returns: The type of the `Certificate`; the value MUST NOT be `null`.

getVersion()

Declaration:
`public String getVersion()`

Description:
Gets the version number of this `Certificate`. The format of the version number depends on the specific type and specification. For a X.509 certificate per RFC 2459 it would be `"3"`.

Returns: The version number of the `Certificate`; the value MUST NOT be `null`.

javax.microedition.pki
CertificateException

Declaration

```
public class CertificateException extends java.io.IOException

Object
    |
    +--Throwable
            |
            +--Exception
            |
            +--java.io.IOException
                    |
                    +--javax.microedition.pki.CertificateException
```

Description

The `CertificateException` encapsulates an error that occurred while a `Certificate` is being used. If multiple errors are found within a `Certificate` the more significant error should be reported in the exception.

Since: IMP-NG / MIDP 2.0

Member Summary

Fields

static byte	BAD_EXTENSIONS
static byte	BROKEN_CHAIN
static byte	CERTIFICATE_CHAIN_TOO_LONG
static byte	EXPIRED
static byte	INAPPROPRIATE_KEY_USAGE
static byte	MISSING_SIGNATURE
static byte	NOT_YET_VALID
static byte	ROOT_CA_EXPIRED
static byte	SITENAME_MISMATCH
static byte	UNAUTHORIZED_INTERMEDIATE_CA
static byte	UNRECOGNIZED_ISSUER
static byte	UNSUPPORTED_PUBLIC_KEY_TYPE
static byte	UNSUPPORTED_SIGALG
static byte	VERIFICATION_FAILED

Constructors

```
CertificateException(Certificate certificate, byte status)
CertificateException(String message, Certificate certificate,
                     byte status)
```

Methods

Certificate	getCertificate()
byte	getReason()

Inherited Member Summary

Methods inherited from class `Object`

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

Methods inherited from class `Throwable`

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

Package
javax.microedition.pki

Fields

BAD_EXTENSIONS

Declaration:
`public static final byte` **`BAD_EXTENSIONS`**

Description:
Indicates a certificate has unrecognized critical extensions. The value is `1`.

BROKEN_CHAIN

Declaration:
`public static final byte` **`BROKEN_CHAIN`**

Description:
Indicates a certificate in a chain was not issued by the next authority in the chain. The value is `11`.

CERTIFICATE_CHAIN_TOO_LONG

Declaration:
`public static final byte` **`CERTIFICATE_CHAIN_TOO_LONG`**

Description:
Indicates the server certificate chain exceeds the length allowed by an issuer’s policy. The value is `2`.

EXPIRED

Declaration:
`public static final byte` **`EXPIRED`**

Description:
Indicates a certificate is expired. The value is `3`.

INAPPROPRIATE_KEY_USAGE

Declaration:
`public static final byte` **`INAPPROPRIATE_KEY_USAGE`**

Description:
Indicates a certificate public key has been used in way deemed inappropriate by the issuer. The value is `10`.

MISSING_SIGNATURE

Declaration:
`public static final byte` **`MISSING_SIGNATURE`**

Description:
Indicates a certificate object does not contain a signature. The value is `5`.

NOT_YET_VALID

Declaration:
`public static final byte` **`NOT_YET_VALID`**

Description:
Indicates a certificate is not yet valid. The value is `6`.

ROOT_CA_EXPIRED

Declaration:
`public static final byte` **`ROOT_CA_EXPIRED`**

Description:
Indicates the root CA’s public key is expired. The value is `12`.

SITENAME_MISMATCH

Declaration:
```
public static final byte SITENAME_MISMATCH
```

Description:
Indicates a certificate does not contain the correct site name. The value is `7`.

UNAUTHORIZED_INTERMEDIATE_CA

Declaration:
```
public static final byte UNAUTHORIZED_INTERMEDIATE_CA
```

Description:
Indicates an intermediate certificate in the chain does not have the authority to be a intermediate CA. The value is `4`.

UNRECOGNIZED_ISSUER

Declaration:
```
public static final byte UNRECOGNIZED_ISSUER
```

Description:
Indicates a certificate was issued by an unrecognized entity. The value is `8`.

UNSUPPORTED_PUBLIC_KEY_TYPE

Declaration:
```
public static final byte UNSUPPORTED_PUBLIC_KEY_TYPE
```

Description:
Indicates that type of the public key in a certificate is not supported by the device. The value is `13`.

UNSUPPORTED_SIGALG

Declaration:
```
public static final byte UNSUPPORTED_SIGALG
```

Description:
Indicates a certificate was signed using an unsupported algorithm. The value is `9`.

VERIFICATION_FAILED

Declaration:
```
public static final byte VERIFICATION_FAILED
```

Description:
Indicates a certificate failed verification. The value is `14`.

Constructors

CertificateException(Certificate, byte)

Declaration:
```
public CertificateException(javax.microedition.pki.Certificate certificate,
                 byte status)
```

Description:
Create a new exception with a `Certificate` and specific error reason. The descriptive message for the new exception will be automatically provided, based on the reason.

Parameters:
`certificate` - the certificate that caused the exception
`status` - the reason for the exception; the status MUST be between `BAD_EXTENSIONS` and `VERIFICATION_FAILED` inclusive.

CertificateException(String, Certificate, byte)

Declaration:
```
public CertificateException(String message,
        javax.microedition.pki.Certificate certificate, byte status)
```

Description:
Create a new exception with a message, `Certificate`, and specific error reason.

Parameters:
`message` - a descriptive message
`certificate` - the certificate that caused the exception
`status` - the reason for the exception; the status MUST be between `BAD_EXTENSIONS` and `VERIFICATION_FAILED` inclusive.

Methods

getCertificate()

Declaration:
```
public javax.microedition.pki.Certificate getCertificate()
```

Description:
Get the `Certificate` that caused the exception.

Returns: the `Certificate` that included the failure.

getReason()

Declaration:
```
public byte getReason()
```

Description:
Get the reason code.

Returns: the reason code

14 Package javax.microedition.rms

Description

Like the MIDP for MIDlets – also IMP-NG provides a mechanism for IMlets 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.

Persistent Storage

The IMP-NG provides a mechanism for IMlets 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.

Record Store

A record store consists of a collection of records that will remain persistent across multiple invocations of an IMlet. The platform is responsible for making its best effort to maintain the integrity of the IMlet'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 IMlets. The naming space for record stores is controlled at the IMlet suite granularity. IMlets within an IMlet suite are allowed to create multiple record stores, as long as they are each given different names. When an IMlet suite is removed from a platform, all record stores associated with its IMlets MUST also be removed. IMlets within an IMlet suite can access one another's record stores directly. New APIs in IMP-NG (derived from those in MIDP 2.0) allow for the explicit sharing of record stores if the IMlet creating the `RecordStore` chooses to give such permission.

Sharing is accomplished through the ability to name a RecordStore in another IMlet suite and by defining the accessibilty rules related to the Authentication of the two IMlet suites.

RecordStores are uniquely named using the unique name of the IMlet suite plus the name of the RecordStore. IMlet suites are identified by the `MIDlet-Vendor` and `MIDlet-Name` attributes from the application descriptor.

Access controls are defined when RecordStores to be shared are created. Access controls are enforced when RecordStores are opened. The access modes allow private use or shareable with any other IMlet suite.

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 IMlet suite. In other words, IMlets within an IMlet suite are not allowed to create more than one record store with the same name; however, an IMlet in one IMlet suite is allowed to have a record store with the same name as an IMlet in another IMlet 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 an IMlet uses multiple threads to access a record store, it is the IMlet's responsibility to coordinate this access, or unintended consequences may result. For example, if two threads in an IMlet 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 IMlet. 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 IMlets 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.

Package javax.microedition.rms

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)`. IMlets can create other indices by using the `RecordEnumeration` class.

Example:

The example uses the Record Management System to store and retrieve temperature values for several measurement points. In the example, temperature values 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;
import javax.microedition.midlet.* ;
/**
 * A class used for storing and sending temperature values.
 */
public class RMSTemperatures extends MIDlet
   implements RecordFilter, RecordComparator
{
    /*
     * The RecordStore used for storing the temperature values.
     */
    private RecordStore recordStore = null;
    /*
     * The measurement point to use when filtering.
     */
    public static String measurementPointFilter = null;
    /*
     * Part of the RecordFilter interface.
     */
    public boolean matches(byte[] candidate) throws IllegalArgumentException
    {
            // If no filter set, nothing can match it.
            if (this. measurementPointFilter == null) {
                    return false;
            }
            ByteArrayInputStream bais = new ByteArrayInputStream(candidate);
            DataInputStream inputStream = new DataInputStream(bais);
            String point = null;
            try {
                    int temp = inputStream.readInt();
                    point = inputStream.readUTF();
            }
            catch (EOFException eofe) {
                    … // some appropriate error handling
            }
            catch (IOException eofe) {
                    … // some appropriate error handling
            }
            return (this.measurementPointFilter.equals(point));
    }
    /*
     * Part of the RecordComparator interface.
     */
    public int compare(byte[] rec1, byte[] rec2)
    {
            // Construct DataInputStreams for extracting the temperature
            // values from the records.
            ByteArrayInputStream bais1 = new ByteArrayInputStream(rec1);
            DataInputStream inputStream1 = new DataInputStream(bais1);
            ByteArrayInputStream bais2 = new ByteArrayInputStream(rec2);
            DataInputStream inputStream2 = new DataInputStream(bais2);
            int temp1 = 0;
            int temp2 = 0;
            try {
                    // Extract the temperature values.
                    temp1 = inputStream1.readInt();
                    temp2 = inputStream2.readInt();
            }
            catch (EOFException eofe) {
                    … // some appropriate error handling
            }
            catch (IOException eofe) {
                    … // some appropriate error handling
            }
            // Sort by temperature value
            if (temp1 < temp2) {
                    return RecordComparator.PRECEDES;
            }
            else if (temp1 > temp2) {
                    return RecordComparator.FOLLOWS;
            }
            else {
                    return RecordComparator.EQUIVALENT;
            }
    }
    /**
     * The constructor opens the underlying record store,
     * creating it if necessary.
     */
    public RMSTemperatures()
    {
            //
            // Create a new record store for this example
            //
            try {
                    recordStore = RecordStore.openRecordStore("temperatures", true);
            }
            catch (RecordStoreException rse) {
                    … // some appropriate error handling
            }
    }
    /**
     * Add a new temperature value to the storage.
     *
     * @param temp the temperature to store.
     * @param point the name of the point where this value has been measured.
     */
    public void addValue(int temp, String point)
    {
            //
            // Each value is stored in a separate record, formatted with
            // the temperature value, followed by the name of the measurement point.
            //
            int recId; // returned by addRecord but not used
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            DataOutputStream outputStream = new DataOutputStream(baos);
            try {
                    // Push the value into a byte array.
                    outputStream.writeInt(temp);
```

```
                // Then push the measurement point’s name.
                outputStream.writeUTF(point);
        }
        catch (IOException ioe) {
                … // some appropriate error handling
        }
        // Extract the byte array
        byte[] b = baos.toByteArray();
        // Add it to the record store
        try {
                recId = recordStore.addRecord(b, 0, b.length);
        }
        catch (RecordStoreException rse) {
                … // some appropriate error handling
        }
}
/**
 * A helper method for the sendValue methods.
 */
private void sendValueHelper(RecordEnumeration re)
{
        try {
                while(re.hasNextElement()) {
                        int id = re.nextRecordId();
                        ByteArrayInputStream bais =
                                new ByteArrayInputStream(recordStore.getRecord(id));
                        DataInputStream inputStream = new DataInputStream(bais);
                        try {
                                int temp = inputStream.readInt();
                                String point = inputStream.readUTF();
                                … // send values to receiver
                        }
                        catch (EOFException eofe) {
                                … // some appropriate error handling
                        }
                        … // handle exceptions occurring while sending the
                          // values to receiver
                }
        }
        catch (RecordStoreException rse) {
                … // some appropriate error handling
        }
        catch (IOException ioe) {
                … // some appropriate error handling
        }
}
/**
 * This method sends all measured temperatures sorted by values.
 */
public void sendValues()
{
        try {
                // Enumerate the records using the comparator implemented
                // above to sort by temperature height.
                RecordEnumeration re = recordStore.enumerateRecords(null, this, true);
                sendValueHelper(re);
        }
        catch (RecordStoreException rse) {
                … // some appropriate error handling
        }
}

/**
 * This method sends all temperature values for a given measurement point,
 * sorted by temperature height.
 */
public void sendValues(String point)
{
        try {
```

```
                    // Enumerate the records using the comparator and filter
                    // implemented above to sort by temperature height.
                    RecordEnumeration re = recordStore.enumerateRecords(this, this, true);
                    sendValueHelper(re);
            }
            catch (RecordStoreException rse) {
                    … // some appropriate error handling
            }
    }

    protected void pauseApp()
    {
            // do something if necessary …
    }

    protected void startApp() throws MIDletStateChangeException
    {
            addValue(22, "Living Room");
            addValue(15, "Bedroom 1");
            addValue(17, "Bedroom 2");
            addValue(5, "Cellar");
            addValue(-11, "Front Door");
            addValue(-12, "Back Door");
            addValue(-15, "Roof");
            sendValues();
            RMSTemperatures.measurementPointFilter="Front Door";
            sendValues("Front Door ");
    }

    protected void destroyApp(Boolean uncon) throws MIDletStateChangeException
    {
            // do something if necessary …
    }
}
```

Since: IMP / MIDP 1.0

Class Summary	
Interfaces	
`RecordComparator`	An interface defining a comparator which compares two records (in an implementationdefined manner) to see if they match or what their relative sort order is.
`RecordEnumeration`	An interface 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	
`RecordStore`	A class representing a record store.
Exceptions	
`InvalidRecordIDException`	Thrown to indicate an operation could not be completed because the record ID was invalid.
`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

Declaration

```
public class InvalidRecordIDException extends RecordStoreException
```

```
Object
    |
    +--Throwable
            |
            +--Exception
                    |
                    +--javax.microedition.rms.RecordStoreException494
                            |
                            +--javax.microedition.rms.InvalidRecordIDException
```

Description

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

Since: IMP / MIDP 1.0

Member Summary
Constructors
`InvalidRecordIDException()`
`InvalidRecordIDException(String message)`

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

Constructors

InvalidRecordIDException()

Declaration:
`public InvalidRecordIDException()`

Description:
Constructs a new `InvalidRecordIDException` with no detail message.

InvalidRecordIDException(String)

Declaration:
`public InvalidRecordIDException(String message)`

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

Parameters:
`message` - the detail message

javax.microedition.rms

RecordComparator

Declaration

```
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();
if (c.compare(recordStore.getRecord(rec1), recordStore.getRecord(rec2))
            == RecordComparator.PRECEDES)
    return rec1;
```

Since: IMP / MIDP 1.0

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

Fields

EQUIVALENT

Declaration:
`public static final int EQUIVALENT`

Description:
`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

Declaration:
`public static final int FOLLOWS`

Description:
`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

Declaration:
`public static final int PRECEDES`

Description:
`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[])

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

Description:
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

Package
javax.microedition.rms

javax.microedition.rms RecordEnumeration

Declaration

`public interface` **`RecordEnumeration`**

Description

An interface 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 recordIds 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.

If the `RecordStore` used by this `RecordEnumeration` is closed, this `RecordEnumeration` becomes invalid and all subsequent operations performed on it will give invalid results or throw a `RecordStoreNotOpenException`, even if the same `RecordStore` is later opened again. In addition, calls to `hasNextElement()` and `hasPreviousElement()` will return `false`.

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 until the beginning is reached.

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

Since: IMP / MIDP 1.0

Member Summary

Methods

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

Methods

destroy()

Declaration:
`public void destroy()`

Description:
Frees internal resources used by this `RecordEnumeration`. IMlets should call this method when they are done using a `RecordEnumeration`. If an IMlet tries to use a `RecordEnumeration` after this method has been called, it will throw a `IllegalStateException`. Note that this method is used for manually aiding in the minimization of immediate resource requirements when this enumeration is no longer needed.

hasNextElement()

Declaration:
`public boolean hasNextElement()`

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

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

hasPreviousElement()

Declaration:
`public boolean hasPreviousElement()`

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

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

isKeptUpdated()

Declaration:
`public boolean isKeptUpdated()`

Description:
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)

Declaration:
`public void keepUpdated(boolean keepUpdated)`

Description:
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. Calling `keepUpdated(true)` has the same effect as calling `RecordEnumeration.rebuild`: the enumeration will be updated to reflect the current record set.

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: `rebuild()`

nextRecord()

Declaration:
```
public byte[] nextRecord() throws InvalidRecordIDException,
                                  RecordStoreNotOpenException, RecordStoreException
```

Description:
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()

Declaration:
```
public int nextRecordId() throws InvalidRecordIDException
```

Description:
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()

Declaration:
```
public int numRecords()
```

Description:
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()

Declaration:
```
public byte[] previousRecord() throws InvalidRecordIDException,
                               RecordStoreNotOpenException, RecordStoreException
```

Description:
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()

Declaration:
```
public int previousRecordId() throws InvalidRecordIDException
```

Description:
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()

Declaration:
```
public void rebuild()
```

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

See Also: `keepUpdated(boolean)`

reset()

Declaration:
```
public void reset()
```

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

Package
javax.microedition.rms

javax.microedition.rms

RecordFilter

Declaration

```
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);
```

Since: IMP / MIDP 1.0

Member Summary	
Methods	
`Boolean`	`matches(byte[] candidate)`

Methods

matches(byte[])

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

Description:
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

Declaration

`public interface` **`RecordListener`**

Description

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

Since: IMP / MIDP 1.0

See Also: `RecordStore.addRecordListener(RecordListener)`

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

Methods

recordAdded(RecordStore, int)

Declaration:
`public void recordAdded(javax.microedition.rms.RecordStore recordStore, int recordId)`

Description:
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)

Declaration:
`public void recordChanged(javax.microedition.rms.RecordStore recordStore, int recordId)`

Description:
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)

Declaration:

```
public void recordDeleted(javax.microedition.rms.RecordStore recordStore, int recordId)
```

Description:

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

Declaration

```
public class RecordStore
```

```
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 IMlet. The platform is responsible for making its best effort to maintain the integrity of the IMlet'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 IMlets. The naming space for record stores is controlled at the IMlet suite granularity. IMlets within an IMlet suite are allowed to create multiple record stores, as long as they are each given different names. When an IMlet suite is removed from a platform all the record stores associated with its IMlets will also be removed. IMlets within an IMlet suite can access each other's record stores directly. New APIs in IMP-NG (derived from those in MIDP 2.0) allow for the explicit sharing of record stores if the IMlet creating the `RecordStore` chooses to give such permission.

Sharing is accomplished through the ability to name a `RecordStore` created by another IMlet suite.

`RecordStores` are uniquely named using the unique name of the IMlet suite plus the name of the RecordStore. IMlet suites are identified by the `MIDlet-Vendor` and `MIDlet-Name` attributes from the application descriptor.

Access controls are defined when `RecordStores` to be shared are created. Access controls are enforced when `RecordStores` are opened. The access modes allow private use or shareable with any other IMlet suite.

Record store names are case sensitive and may consist of any combination of between one and 32 Unicode characters inclusive. Record store names must be unique within the scope of a given IMlet suite. In other words, IMlets within an IMlet suite are not allowed to create more than one record store with the same name, however an IMlet in one IMlet suite is allowed to have a record store with the same name as an IMlet in another IMlet 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 an IMlet uses multiple threads to access a record store, it is the IMlet'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 IMlet 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 recorded 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'`. IMlets can create other sequences of the records in the `RecordStore` 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* number, 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.

Since: IMP / MIDP 1.0

Package javax.microedition.rms

Member Summary

Fields

`static int`	`AUTHMODE_ANY`
`static int`	`AUTHMODE_PRIVATE`

Methods

`int`	`addRecord(byte[] data, int offset, int numBytes)`
`void`	`addRecordListener(RecordListener listener)`
`void`	`closeRecordStore()`
`void`	`deleteRecord(int recordId)`
`static void`	`deleteRecordStore(String recordStoreName)`
`RecordEnumeration`	`enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean keepUpdated)`
`long`	`getLastModified()`
`java.lang.String`	`getName()`
`int`	`getNextRecordID()`
`int`	`getNumRecords()`
`byte[]`	`getRecord(int recordId)`
`int`	`getRecord(int recordId, byte[] buffer, int offset)`
`int`	`getRecordSize(int recordId)`
`int`	`getSize()`
`int`	`getSizeAvailable()`
`int`	`getVersion()`
`static java.lang.String[]`	`listRecordStores()`
`static RecordStore`	`openRecordStore(String recordStoreName, Boolean createIfNecessary)`
`static RecordStore`	`openRecordStore(String recordStoreName, Boolean createIfNecessary, int authmode, boolean writable)`
`static RecordStore`	`openRecordStore(String recordStoreName, String vendorName, String suiteName)`
`void`	`removeRecordListener(RecordListener listener)`
`void`	`setMode(byte authmode, boolean writable)`
`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(), wait()`

Fields

AUTHMODE_ANY

Declaration:
`public static final int` **`AUTHMODE_ANY`**

Description:
Authorization to allow access to any IMlet suites. `AUTHMODE_ANY` has a value of `1`.

AUTHMODE_PRIVATE

Declaration:
`public static final int` **`AUTHMODE_PRIVATE`**

Description:
Authorization to allow access only to the current IMlet suite. `AUTHMODE_PRIVATE` has a value of `0`.

Methods

addRecord(byte[], int, int)

Declaration:

```
public int addRecord(byte[] data, int offset, int numBytes)
         throws RecordStoreNotOpenException, RecordStoreException,
                RecordStoreFullException
```

Description:
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
`SecurityException` - if the IMlet has read-only access to the `RecordStore`

addRecordListener(RecordListener)

Declaration:

```
public void addRecordListener(javax.microedition.rms.RecordListener listener)
```

Description:
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

See Also: `removeRecordListener(RecordListener)`

closeRecordStore()

Declaration:

```
public void closeRecordStore() throws RecordStoreNotOpenException, RecordStoreException
```

Description:
This method is called when the IMlet 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 IMlet 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 and all `RecordEnumerations` associated with it become invalid. If the IMlet 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)

Declaration:

```
public void deleteRecord(int recordId)
                 throws RecordStoreNotOpenException, InvalidRecordIDException,
                        RecordStoreException
```

Description:
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
`SecurityException` - if the IMlet has read-only access to the `RecordStore`

deleteRecordStore(String)

Declaration:

```
public static void deleteRecordStore(String recordStoreName)
                 throws RecordStoreException, RecordStoreNotFoundException
```

Description:
Deletes the named record store. IMlet suites are only allowed to delete their own record stores. If the named record store is open (by an IMlet in this suite or an IMlet in a different IMlet suite) when this method is called, a `RecordStoreException` will be thrown. If the named record store does not exist a `RecordStoreNotFoundException` will be thrown. Calling this method does NOT result in `recordDeleted` calls to any registered listeners of this `RecordStore`.

Parameters:

`recordStoreName` - the IMlet 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)

Declaration:

```
public javax.microedition.rms.RecordEnumeration enumerateRecords
                 (javax.microedition.rms.RecordFilter filter,
                  javax.microedition.rms.RecordComparator comparator,
                  boolean keepUpdated)
                        throws RecordStoreNotOpenException
```

Description:
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` is 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. If a filter is used with a null comparator, the enumeration will traverse the filtered records in an undefined order. 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: `RecordEnumeration.rebuild()`

getLastModified()

Declaration:
`public long `**`getLastModified`**`() throws RecordStoreNotOpenException`

Description:
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()

Declaration:
`public String `**`getName`**`() throws RecordStoreNotOpenException`

Description:
Returns the name of this `RecordStore`.

Returns: the name of this `RecordStore`

Throws:

`RecordStoreNotOpenException` - if the record store is not open

getNextRecordID()

Declaration:
`public int `**`getNextRecordID`**`() throws RecordStoreNotOpenException, RecordStoreException`

Description:
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 `recorded` 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()

Declaration:
```
public int getNumRecords() throws RecordStoreNotOpenException
```

Description:
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)

Declaration:
```
public byte[] getRecord(int recordId)
                 throws RecordStoreNotOpenException, InvalidRecordIDException,
                        RecordStoreException
```

Description:
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

See Also: `setRecord(int, byte[], int, int)`

getRecord(int, byte[], int)

Declaration:
```
public int getRecord(int recordId, byte[] buffer, int offset)
                 throws RecordStoreNotOpenException, InvalidRecordIDException,
                        RecordStoreException
```

Description:
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

See Also: `setRecord(int, byte[], int, int)`

getRecordSize(int)

Declaration:

```
public int getRecordSize(int recordId)
                 throws RecordStoreNotOpenException, InvalidRecordIDException,
                        RecordStoreException
```

Description:
Returns the size (in bytes) of the IMlet 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 IMlet 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()

Declaration:

```
public int getSize() throws RecordStoreNotOpenException
```

Description:
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()

Declaration:

```
public int getSizeAvailable() throws RecordStoreNotOpenException
```

Description:
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 IMlet-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()

Declaration:
```
public int getVersion() throws RecordStoreNotOpenException
```

Description:
Each time a record store is modified (by `addRecord`, `setRecord`, or `deleteRecord` methods) its *version* is incremented. This can be used by IMlets 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 increases only when the `RecordStore` is updated. The increment value need not be constant and may vary with each update.

Returns: the current record store version

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

listRecordStores()

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

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

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

openRecordStore(String, boolean)

Declaration:
```
public static javax.microedition.rms.RecordStore openRecordStore
                   (String recordStoreName, boolean createIfNecessary)
                           throws RecordStoreException, RecordStoreFullException,
                                   RecordStoreNotFoundException
```

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

Parameters:
`recordStoreName` - the IMlet suite unique name for the record store, consisting of between one and 32 Unicode characters inclusive.
`createIfNecessary` - if `true`, the record store will be created if necessary

Returns: `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
`IllegalArgumentException` - if `recordStoreName` is invalid

openRecordStore(String, boolean, int, boolean)

Declaration:

```
public static javax.microedition.rms.RecordStore openRecordStore
                (String recordStoreName, boolean createIfNecessary, int authmode,
                 boolean writable)
              throws RecordStoreException, RecordStoreFullException,
                   RecordStoreNotFoundException
```

Description:

Open (and possibly create) a record store that can be shared with other IMlet suites. The `RecordStore` is owned by the current IMlet suite. The authorization mode is set when the record store is created, as follows:

- `AUTHMODE_PRIVATE` - Only allows the IMlet suite that created the `RecordStore` to access it. This case behaves identically to `openRecordStore(recordStoreName,createIfNecessary)`.
- `AUTHMODE_ANY` - Allows any IMlet to access the `RecordStore`. Note that this makes your record store accessible by any other IMlet on the device. This could have privacy and security issues depending on the data being shared. Please use carefully.

The owning IMlet suite may always access the `RecordStore` and always has access to write and update the store.

If this method is called by an IMlet when the record store is already open by an IMlet in the IMlet suite, this method returns a reference to the same `RecordStore` object.

Parameters:

`recordStoreName` - the IMlet suite unique name for the record store, consisting of between one and 32 Unicode characters inclusive.
`createIfNecessary` - if `true`, the record store will be created if necessary
`authmode` - the mode under which to check or create access. Must be one of `AUTHMODE_PRIVATE` or `AUTHMODE_ANY`. This argument is ignored if the `RecordStore` exists.
`writable` - `true` if the `RecordStore` is to be writable by other IMlet suites that are granted access. This argument is ignored if the `RecordStore` exists.

Returns: `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
`SecurityException` - if this IMlet Suite is not allowed to open or create the specified `RecordStore`
`IllegalArgumentException` - if authmode or `recordStoreName` is invalid

Since: IMP-NG / MIDP 2.0

openRecordStore(String, String, String)

Declaration:

```
public static javax.microedition.rms.RecordStore openRecordStore
                (String recordStoreName, String vendorName, String suiteName)
          throws RecordStoreException, RecordStoreNotFoundException
```

Description:

Open a record store associated with the named IMlet suite. The IMlet suite is identified by `MIDlet-vendor` and `MIDlet-name`. Access is granted only if the authorization mode of the `RecordStore` allows access by the current IMlet suite. Access is limited by the authorization mode set when the record store was created:

- `AUTHMODE_PRIVATE` - Succeeds only if `vendorName` and `suiteName` identify the current IMlet suite; this case behaves identically to `openRecordStore(recordStoreName,createIfNecessary)`.
- AUTHMODE_ANY - Always succeeds. Note that this makes your record store accessible by any other IMlet on the device. This could have privacy and security issues depending on the data being shared. Please use carefully. Untrusted IMlet suites are allowed to share data but this is not recommended. The

authenticity of the origin of untrusted IMlet suites cannot be verified so shared data may be used unscrupulously.

If this method is called by an IMlet when the record store is already open by an IMlet in the IMlet suite, this method returns a reference to the same `RecordStore` object.

If an IMlet calls this method to open a record store from its own suite, the behavior is identical to calling: `openRecordStore(recordStoreName, false).`

Parameters:

`recordStoreName` - the IMlet suite unique name for the record store, consisting of between one and 32 Unicode characters inclusive.

`vendorName` - the vendor of the owning IMlet suite

`suiteName` - the name of the IMlet suite

Returns: `RecordStore` object for the record store

Throws:

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

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

`SecurityException` - if this IMlet Suite is not allowed to open the specified `RecordStore`.

`IllegalArgumentException` - if `recordStoreName` is invalid

Since: IMP-NG / MIDP 2.0

removeRecordListener(RecordListener)

Declaration:

`public void removeRecordListener(javax.microedition.rms.RecordListener listener)`

Description:

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

Parameters:

`listener` - the RecordChangedListener

See Also: `addRecordListener(RecordListener)`

setMode(byte, boolean)

Declaration:

`public void setMode(byte authmode, boolean writable) throws RecordStoreException`

Description:

Changes the access mode for this `RecordStore`. The authorization mode choices are:

- `AUTHMODE_PRIVATE` - Only allows the IMlet suite that created the `RecordStore` to access it. This case behaves identically to `openRecordStore(recordStoreName,createIfNecessary).`
- AUTHMODE_ANY - Allows any IMlet to access the RecordStore. Note that this makes your record store accessible by any other IMlet on the device. This could have privacy and security issues depending on the data being shared. Please use carefully.

The owning IMlet suite may always access the `RecordStore` and always has access to write and update the store. Only the owning IMlet suite can change the mode of a `RecordStore`.

Parameters:

`authmode` - the mode under which to check or create access. Must be one of `AUTHMODE_PRIVATE` or `AUTHMODE_ANY`.

`writable` - `true` if the `RecordStore` is to be writable by other IMlet suites that are granted access

Throws:

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

`SecurityException` - if this IMlet Suite is not allowed to change the mode of the `RecordStore`
`IllegalArgumentException` - if `authmode` is invalid

Since: IMP-NG / MIDP 2.0

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

Declaration:

```
public void setRecord(int recordId, byte[] newData, int offset, int numBytes)
                 throws RecordStoreNotOpenException, InvalidRecordIDException,
                        RecordStoreException, RecordStoreFullException
```

Description:
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
`SecurityException` - if the IMlet has read-only access to the `RecordStore`

See Also: `getRecord(int, byte[], int)`

javax.microedition.rms RecordStoreException

Declaration

```
public class RecordStoreException extends Exception

Object
    |
    +--Throwable
            |
            +--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.

Since: IMP / MIDP 1.0

Member Summary

Constructors

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

Inherited Member Summary

Methods inherited from class `Object`

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

Methods inherited from class `Throwable`

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

Constructors

RecordStoreException()

Declaration:
`public RecordStoreException()`

Description:
Constructs a new `RecordStoreException` with no detail message.

RecordStoreException(String)

Declaration:
`public RecordStoreException(String message)`

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

Parameters:
`message` - the detail message

javax.microedition.rms RecordStoreFullException

Declaration

```
public class RecordStoreFullException extends RecordStoreException
```

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

Description

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

Since: IMP / MIDP 1.0

Member Summary
Constructors
`RecordStoreFullException()`
`RecordStoreFullException(String message)`

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

Constructors

RecordStoreFullException()

Declaration:
`public RecordStoreFullException()`

Description:
Constructs a new `RecordStoreFullException` with no detail message.

RecordStoreFullException(String)

Declaration:
`public RecordStoreFullException(String message)`

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

Parameters:
`message` - the detail message

javax.microedition.rms RecordStoreNotFoundException

Declaration

`public class` **`RecordStoreNotFoundException`** `extends RecordStoreException`

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

Description

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

Since: IMP / MIDP 1.0

Member Summary
Constructors
`RecordStoreNotFoundException()`
`RecordStoreNotFoundException(String message)`

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

Constructors

RecordStoreNotFoundException()

Declaration:
`public` **`RecordStoreNotFoundException`**`()`

Description:
Constructs a new `RecordStoreNotFoundException` with no detail message.

RecordStoreNotFoundException(String)

Declaration:
`public` **`RecordStoreNotFoundException`**`(String message)`

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

Parameters:
`message` - the detail message

Package
javax.microedition.rms

javax.microedition.rms RecordStoreNotOpenException

Declaration

```
public class RecordStoreNotOpenException extends RecordStoreException
```

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

Description

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

Since: IMP / MIDP 1.0

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

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

Constructors

RecordStoreNotOpenException()

Declaration:
`public RecordStoreNotOpenException()`

Description:
Constructs a new `RecordStoreNotOpenException` with no detail message.

RecordStoreNotOpenException(String)

Declaration:
`public RecordStoreNotOpenException(String message)`

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

Parameters:
`message` - the detail message

15 The Recommended Security Policy for GSM/UMTS Compliant Devices

15.1 Scope

This addendum is informative. However, all implementations of IMP-NG on GSM/UMTS compliant devices are expected to comply with this addendum.

IMP-NG defines the framework for authenticating the source of an IMlet suite and authorizing the IMlet suite to perform protected functions by granting permissions it may have requested based on the security policy on the device. It also identifies functions that are deemed security vulnerable and defines permissions for those protected functions. Additionally, IMP-NG specifies the common rules for APIs that can be used together with IMP-NG but are specified outside the profile. IMP-NG specification does not mandate a single trust model but rather allows the model to accord with the device trust policy.

The purpose of this addendum is to extend the base IMlet suite security framework defined in the chapters above and to define the following areas:

- The required trust model for GSM/UMTS compliant devices
- The domain number and structure, as reflected in the device security policy
- The mechanism of reading root keys from sources external to the device
- Capabilities of IMlets based on permissions defined by IMP-NG and other JSRs
- IMlet behavior in the roaming network
- IMlet behavior when SIM/USIM is changed

15.2 General

GSM/UMTS compliant devices implementing this Recommended Security Policy MUST follow the security framework specified in IMP-NG. Additionally, devices that support trusted IMlets MUST follow the PKI-based authentication scheme as defined in the IMP-NG specification.

15.3 Protection Domains in the Device Security Policy

A protection domain is a way to differentiate between downloaded IMlet suites based on the entity that signed the IMlet suite (there is no such entity for untrusted IMlet suites, of course), and to grant or make available to an IMlet suite a set of permissions. A domain binds a Protection Domain Root Certificate to a set of permissions. The permissions are specified in the protection domain security policy, a policy has as many entries as there are protection domains available on the device (this is at least one for the untrusted IMlet suites). A domain can exist only for a Protection Domain Root Certificate that contains the `id-kp-codeSigning` extended key usage extension. IMlet suites that authenticate to a trusted Protection Domain Root Certificate are treated as trusted, and assigned to the corresponding protection domain. An IMlet suite cannot belong to more than one protection domain. The representation of a domain and its security policy is implementation specific.

15.4 Protection Domains and the Permissions Framework

This chapter specifies two different requirements as to how the IMP-NG permissions framework should be used, depending on the protection domain an application executes.

Untrusted Domain Protection Domain defining which (security vulnerable) APIs and functions an untrusted application may access when running on an IM.

User Domains Protection Domains IMlet suites are assigned to after having authenticated to a corresponding trusted Protection Domain Root Certificate. Those domains define which (security vulnerable) APIs and functions a trusted application may access when running on an IM, depending on to which Protection Domain Root Certificate it authenticated.

15.4.1 User Domains

A User Protection Domain Root Certificate is used to verify user IMlet suites. There is no explicit limitation on the number of User Protection Domain Root Certificates available either on the device or at the specified location in the SIM, USIM or WIM. A User Protection Domain Root Certificates MUST be mapped on to the security policy for the corresponding user domain on the device. A device MUST support the security policy for the user domain. If there are no User Protection Domain Root Certificates available either on the device or at the specified location in the SIM, USIM or WIM; the user domain MUST be disabled.

Contrary to MIDP 2.0, User Protection Domain Root Certificates downloaded after device manufacturer are allowed to be used for authentication of IMlet suites. This is due to another procedure of certificate delivery for IMs than for MIDs.

The user MUST be able to delete or disable User Protection Domain Root CertificatesThe user MUST be able to enable a disabled User Protection Domain Root Certificate. A disabled User Protection Domain Root Certificate MUST NOT be used to verify downloaded IMlet suites. Furthermore, if a User Protection Domain Root Certificate is deleted or disabled (for example, revoked, deleted, or disabled by the user) the user domain MUST no longer be associated with this Protection Domain Root Certificate. If the user chooses to delete or disable the Protection Domain Root Certificate, implementation MAY delete the IMlet suites authenticated to it.

15.5 Remotely Located Security Policy

The IMP-NG specification defines the generic format for a policy file that can be read from removable media. GSM/UMTS compliant devices are not expected to use it in the first phase, but rather to use security policy resident on the device. The possibility of remotely located security policy files is left for further consideration.

15.6 Permissions for Downloaded IMlet Suites

15.6.1 Mapping IMP-NG Permissions onto Function Groups in Protected Domains

In MIDP 2.0 specification, the problem of an appropriate presentation of permissions on devices with small displays has been discussed. Though this problem naturally does not exist for IMs due to the lack of a display, it seems useful to use the MIDP 2.0 proposed mapping of functionality into high-level function groups also for IMP-NG. Using these high-level function groups makes it much easier to set the permissions for a particular protection domain policy. The function groups are as follows:

Network/cost-related groups:

Phone Call: the group represents permissions to any function that results in a voice call.

Net Access: the group represents permissions to any function that results in an active network data connection (for example GSM, GPRS, UMTS, etc.); such functions must be mapped to this group.

Messaging: the group represents permissions to any function that allows sending or receiving messages (for example, SMS, etc.)

Application Auto Invocation: the group represents permissions to any function that allows an IMlet suite to be invoked automatically (for example, push, timed IMlets, etc.)

Application Management: the group represents permissions to any function that allows to install, start, pause, stop or remove an IMlet (suite) (for example, platformRequest if used to install an application (suite))

Local Connectivity: the group represents permissions to any function that activates a local port for further connection (for example, COMM port, IrDA, Bluetooth, etc.)

User-privacy-related groups:

Multimedia recording: the group represents permissions to any function that gives an IMlet suite the ability to record audio clips.

Read User Data Access: the group represents permissions to any function that gives an IMlet suite the ability to read any data in a file or directory.

Write User Data Access: the group represents permissions to any function that gives an IMlet suite the ability to add or modify any data in a file or directory.

Whenever new features are added to IMP-NG they should be assigned to the appropriate function group. In addition, APIs that are specified elsewhere (that is, in other JSRs) but rely on the IMP-NG security framework should also be assigned to an appropriate function group. If none of the function groups defined in this section is able to capture the new feature, however, then a new function group MUST be defined in this document.

If a new function group is to be added, the following should be taken into consideration: the group to be added MUST not introduce any redundancy to the existing groups, the new group MUST be capable of protecting a wide range of similar features. The latter requirement is to prevent introducing narrowly scoped groups.

Also, as IMP-NG is a strict subset of MIDP 2.0, before adding a new function group, a synchronization between the IMP-NG and MIDP 2.0 maintenance leads should take place – according to the practice as agreed in the "Maintenance Note" at the begin of this document.

Tables 15-1 through 15-3 present individual permissions defined in IMP-NG and other JSRs, and map to the function groups specified in this section. An individual permission MUST occur in only one function group.

TABLE 15-1: Assigning permissions specified in IMP-NG to function groups

IMP-NG JSR 228		
Permission	**Protocol**	**Function group**
javax.microedition.io.Connector.http	http	Net Access
javax.microedition.io.Connector.https	https	Net Access
javax.microedition.io.Connector.datagram	datagram	Net Access
javax.microedition.io.Connector.datagramreceiver	datagram server (without host)	Net Access
javax.microedition.io.Connector.socket	socket	Net Access
javax.microedition.io.Connector.serversocket	server socket (without host)	Net Access
javax.microedition.io.Connector.ssl	ssl	Net Access
javax.microedition.io.Connector.comm	comm.	Local Connectivity
javax.microedition.io.PushRegistry	All	Application Auto Invocation
javax.microedition.midlet.MIDlet.install	All	Application Management

TABLE 15-2: Assigning proposed permissions and API calls specified in the Bluetooth API to function groups

Bluetooth API JSR 82		
Security Policy Identifier (Proposed Permission)	**Permitted API calls**	**Function group**

javax.microedition.io.Connector.bluetooth.client	Connector.open("btspp://<server BD_ADDR>"")
	Connector.open("btl2cap://<server BD_ADDR>"")
		Local Connectivity

javax.microedition.io.Connector.obex.client	Connector.open("btgoep://<server BD_ADDR>"")
	Connector.open("irdaobex://discover"")
	Connector.open("irdaobex://addr"")
	Connector.open("irdaobex://conn"")
	Connector.open("irdaobex://name"")
		Local Connectivity

javax.microedition.io.Connector.obex.client.tcp	Connector.open("tcpobex://<server IP_ADDR>"")	Net Access

javax.microedition.io.Connector.bluetooth.server	Connector.open("btspp://localhost:"")
	Connector.open("btl2cap://localhost:"")
		Local Connectivity

javax.microedition.io.Connector.obex.server	Connector.open("btgoep://localhost:"")
	Connector.open("irdaobex://localhost:"")
		Local Connectivity

javax.microedition.io.Connector.obex.server.tcp	Connector.open("tcpobex://:<PORT>")
	Connector.open("tcpobex://")
		Net Access

TABLE 15-3: Assigning proposed permissions and API calls specified in the Wireless Messaging API to function groups

Wireless Messaging API: JSR 120		
Security Policy Identifier (Proposed Permission)	**Permitted API calls**	**Function group**

javax.microedition.io.Connector.sms.send	Connector.open("sms://"", WRITE)
	Connector.open("sms://"", WRITE, Bool)
		Messaging

javax.microedition.io.Connector.sms.receive	Connector.open("sms://"", READ)
	Connector.open("sms://"", READ, Bool)
		Messaging

javax.microedition.io.Connector.sms	Connector.open("sms://"")
	Connector.open("sms://"", READ)
	Connector.open("sms://"", READ, Bool)
	Connector.open("sms://"", WRITE)
	Connector.open("sms://"", WRITE, Bool)
	Connector.open("sms://"", READ_WRITE)
	Connector.open("sms://"", READ_WRITE, Bool)
		Messaging

javax.microedition.io.Connector.cbs.receive	Connector.open("cbs://"")
	Connector.open("cbs://"", READ)
	Connector.open("cbs://"", READ, Bool)
		Messaging

15.6.2 Implementation notes:

When the user grants permission to a function group, this action effectively grants access to all individual permissions under this function group.

An implementation MUST guarantee that a `SecurityException` is thrown when the caller does not have the appropriate security permissions.

If an IMlet uses the capabilities defined in IMP-NG and other APIs, the following rules MUST apply:

- All the external API functions that need to be protected by IMP-NG security framework MUST have permissions defined in the subsequent JSRs, and follow the naming rules identified in the IMP-NG Specification, chapter **Error! Reference source not found.**. **Error! Reference source not found.**.

The Recommended Security Policy for GSM/UMTS Compliant Devices

- The functions that are not deemed security-protected by specification can be accessed explicitly by all IMlet suites, as per general IMP-NG security rules.
- If an external API does not define permissions for security-protected functions because the API specification is released earlier than IMP-NG (or MIDP 2.0, where these requirements were postulated first), any functions that relate to network access are recommended being rejected for untrusted applications.
- All licensee open classes MUST adhere to the permission framework as defined in this document.