



# Virtual Machine Specification

---

Java Card™ Platform, Version 3.0.1,  
Connected Edition

Copyright © 2009 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, Java Card, Javadoc, JDK, and JVM are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

The Adobe logo is a trademark or registered trademark of Adobe Systems, Incorporated.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

---

Copyright © 2009 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, États-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plus des brevets américains listés à l'adresse <http://www.sun.com/patents> et un ou les brevets supplémentaires ou les applications de brevet en attente aux États - Unis et dans les autres pays.

Droits du gouvernement des États-Unis – Logiciel Commercial. Les droits des utilisateur du gouvernement des États-Unis sont soumis aux termes de la licence standard Sun Microsystems et aux conditions appliquées de la FAR et de ces compléments.

Cette distribution peut inclure des éléments développés par des tiers.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, Java Card, Javadoc, JDK, et JVM sont des marques de fabrique ou des marques déposées enregistrées de Sun Microsystems, Inc. ou ses filiales aux États-Unis et dans d'autres pays.

UNIX est une marque déposée aux États-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Le logo Adobe est une marque déposée de Adobe Systems, Incorporated.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont regis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou reexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations de des produits ou des services qui sont régi par la législation américaine sur le contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE À LA QUALITÉ MARCHANDE, À L'APTITUDE À UNE UTILISATION PARTICULIÈRE OU À L'ABSENCE DE CONTREFAÇON.



# Contents

---

## **Preface ix**

### **1. Introduction and Background 1-1**

#### 1.1 Comparison to CLDC 1.1 Specification 1-2

### **2. Goals, Requirements, and Scope 2-1**

#### 2.1 Goals 2-1

#### 2.2 Requirements 2-2

##### 2.2.1 Hardware Requirements 2-2

##### 2.2.2 Software Requirements 2-4

##### 2.2.3 Java Card Requirements 2-4

#### 2.3 Scope 2-5

### **3. High-level Architecture and Security 3-1**

#### 3.1 Java Card VM High-Level Architecture 3-1

#### 3.2 Application Management 3-2

#### 3.3 Security 3-3

##### 3.3.1 Low-Level (Virtual Machine) Security 3-4

##### 3.3.2 Application-Level Security 3-4

##### 3.3.2.1 Sandbox Model 3-4

##### 3.3.2.2 Protecting System Classes 3-5

3.3.2.3	Additional Restrictions on Dynamic Class Loading	3-5
3.3.3	End-to-End Security	3-6
<b>4.</b>	<b>Adherence to the Java Language Specification</b>	<b>4-1</b>
4.1	No Floating Point Support	4-1
4.2	No Finalization of Class Instances	4-2
4.3	Exception and Error Handling Limitations	4-2
<b>5.</b>	<b>Adherence to Java Virtual Machine Specification</b>	<b>5-1</b>
5.1	No Floating Point Support	5-1
5.2	Features Eliminated from the Virtual Machine	5-3
5.2.1	User-Defined Class Loaders	5-3
5.2.2	Thread Groups and Daemon Threads	5-3
5.2.3	Finalization of Class Instances	5-4
5.2.4	Errors and Asynchronous Exceptions	5-4
5.3	Class File Verification	5-4
5.4	Class File Format and Class Loading	5-5
5.4.1	Supported File Formats	5-5
5.4.2	Public Representation of Java Applications and Resources	5-6
5.4.3	Class File Lookup Order	5-7
5.4.4	Implementation Optimizations and Alternative Application Representation Formats	5-8
<b>6.</b>	<b>Java Card Platform Libraries</b>	<b>6-1</b>
6.1	Overview	6-1
6.2	Classes Derived from Java SE Platform	6-2
6.2.1	System Classes	6-2
6.2.2	Data Type Classes	6-3
6.2.3	Collection Classes	6-4
6.2.4	Input-Output Classes	6-4

6.2.5	Calendar and Time Classes	6–5
6.2.6	Additional Utility Classes	6–5
6.2.7	Additional Security Classes	6–6
6.2.8	Basic Internationalization and Localization Classes	6–6
6.2.9	Exception and Error Classes	6–6
6.2.9.1	Exception Classes	6–6
6.2.9.2	Error Classes	6–8
6.2.10	Internationalization	6–8
6.2.11	Property Support	6–9
6.3	Classes Derived from Java ME Platform	6–10
6.3.1	The Generic Connection Framework	6–11
6.3.1.1	Additional Remarks	6–12
6.3.2	Public Key Infrastructure Classes	6–12

**Glossary** Glossary–1

**Index** Index–1



# Figures

---

FIGURE 3-1    High-level Architecture    3-2





# Preface

---

The *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* is based on the *Connected Limited Device Configuration Specification, Version 1.1* standard defined by the Java™ Platform, Micro Edition (Java ME platform). The technology defined in this specification for the Java Card™ 3 Platform is suitable for an advanced smart card device. Smart card devices are typically much more resource-constrained than Java ME platform devices. In this book, Java Card 3 Platform refers to both versions 3.0 and 3.0.1 to distinguish them from all earlier versions.

The technology supports class file loading from a Java™ Archive (JAR) file application distribution format and supports on-card class file verification. It supports multithreading and handles concurrent execution of applications. It also supports automatic Garbage Collection (GC) and provides a framework for end-to-end connectivity.

This document, *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*, defines the subset of the Java programming language, the subset of functionality of the platform's Java Virtual Machine<sup>1</sup>, the security and networking features, as well as the core platform libraries, all to support a wide range of vertical markets for smart card and secure token devices.

---

## Who Should Use This Specification

The audience for this document includes:

- Application developers and content providers who want to write Java Card technology applications for advanced smart card devices

---

1. The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java(TM) platform.

- Chip manufacturers who want to build small Java Card technology-enabled smart cards conforming to the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*
  - Java Card platform vendors who want to build implementations that conform to the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*.
- 

## Before You Read This Specification

Before reading this guide, you should be familiar with the Java programming language, the other Java Card technology specifications, and smart card technology. A good resource for becoming familiar with Java technology and Java Card technology is the Sun Microsystems, Inc. web site, located at

<http://java.sun.com>

You should also be familiar with the Java Card technology website at

<http://java.sun.com/products/javacard/>

---

## How This Specification Is Organized

The topics in this specification are organized as follows:

**Chapter 1** provides some background information and context for the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*. The chapter also summarizes the main differences between *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* and the *Connected Limited Device Configuration Specification, Version 1.1 (CLDC)*.

**Chapter 2** defines the goals, requirements and scope of this specification.

**Chapter 3** defines the high-level architecture of the Java Card platform and discusses its security features.

**Chapter 4** details the variances from the standard Java programming language defined by the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*.

[Chapter 5](#) details the variances from the standard Java Virtual Machine (JVM) defined by the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*.

[Chapter 6](#) defines the Java technology-based APIs (Java APIs) supported by the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*.

[Glossary](#) provides definitions of selected terms used in the entire Connected Edition.

---

# Using UNIX Commands

This document might not contain information about basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following for this information:

- Software documentation that you received with your system
- Solaris™ Operating System documentation, which is at:  
<http://docs.sun.com>

---

# Shell Prompts

Shell	Prompt
C shell	<i>machine-name%</i>
C shell superuser	<i>machine-name#</i>
Bourne shell and Korn shell	\$
Bourne shell and Korn shell superuser	#

# Typographic Conventions

Typeface*	Meaning	Examples
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. % You have mail.
<b>AaBbCc123</b>	What you type, when contrasted with on-screen computer output	% <b>su</b> password:
<i>AaBbCc123</i>	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be superuser to do this. To delete a file, type <code>rm filename</code> .

\* The settings on your browser might differ from these settings.

## Related Documentation

*The Java Language Specification, Third Edition*, by James Gosling, Bill Joy, and Guy L. Steele (Addison-Wesley, 2005), ISBN 0-321-24678-0.

*The Java™ Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999).

*The Java Virtual Machine Specification (JVMS) for Java 5* at:

[http://java.sun.com/docs/books/jvms/second\\_edition/jvms-java5.html](http://java.sun.com/docs/books/jvms/second_edition/jvms-java5.html)

*The Java Development Kit(JDK), Version 1.6* at:

<http://java.sun.com/javase/6/docs/>

The Java Development Kit, Version 1.6 revisions to chapter 4 (Class File Format) of *Java Virtual Machine Specification* at:

<http://jcp.org/aboutJava/communityprocess/final/jsr202/>

*Connected, Limited Device Configuration Specification, Version 1.1*, Java Community Process, Sun Microsystems, Inc. at:

<http://jcp.org/aboutJava/communityprocess/final/jsr139/>

*Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition.*

*Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition.*

*Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition.*

---

## Documentation, Support, and Training

Sun Function	URL
Documentation	<a href="http://www.sun.com/documentation/">http://www.sun.com/documentation/</a>
Support	<a href="http://www.sun.com/support/">http://www.sun.com/support/</a>
Training	<a href="http://www.sun.com/training/">http://www.sun.com/training/</a>

---

## Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

---

## Sun Welcomes Your Comments

Sun Microsystems is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments to

[jc-bandol-spec-feedback@sun.com](mailto:jc-bandol-spec-feedback@sun.com)

Please include the title of your document with your feedback:

*Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition.*



# Introduction and Background

---

This book is targeted for the Connected Edition. The Java Card 3 Platform consists of two editions.

- The Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and is backward compatible with it, targeting resource-constrained devices that solely support applet-based applications. You may disregard the specifications for the Connected Edition if you are interested in the functionality found only in the Classic Edition.
- The Connected Edition features a significantly enhanced runtime environment and a new virtual machine. It includes new network-oriented features, such as support for web applications, including the Java™ Servlet APIs, and also support for applets with extended and advanced capabilities. An application written for or an implementation of the Connected Edition may use features found in the Classic Edition. Therefore, you will need to use the specifications for both the Classic Edition and the Connected Edition.

This document specifies the virtual machine of the Java Card 3 platform, Connected Edition. The main goal of the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* is to standardize a highly portable, minimum-footprint Java application development platform for resource-constrained, advanced smart card and secure token devices.

This specification defines the minimum required complement of Java technology components and libraries for smart card devices. Java programming language and virtual machine features, core libraries, security, input/output, and networking are the primary topics addressed by this specification.

*Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* defines the core platform that will be used as the basis to support the Web Application Model and the APDU-based Application Model on an advanced smart card device as described in the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition*.

---

## 1.1 Comparison to CLDC 1.1 Specification

The list below summarizes the main differences in *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* when compared with the *Connected Limited Device Configuration Specification, Version 1.1 (CLDC)*:

- Java Development Kit (JDK™) version 1.6 language features defined in the *The Java Language Specification (Java Series), Third Edition* are included.
- The JDK version 1.5 virtual machine extensions to the class file format are supported.
- The JDK version 1.6 virtual machine extensions to the class file format are supported.
- Floating point support has been removed.
  - All floating point byte codes have been removed.
  - Classes `Float` and `Double` have been removed.
  - Methods in library classes that handle floating point and double values have been removed.
- Weak reference support has been removed.
- Some methods in library classes have been removed.
  - `Runtime.exit()`
  - `System.exit()`
- Additional library classes that provide programming utilities have been added.
  - `java.lang.Enum`
  - `java.lang.IllegalStateException`
  - `java.lang.Iterable<T>`
  - `java.lang.StringBuilder`
  - `java.util.EventObject`
  - `java.util.EventListener`
  - `java.util.Iterator<E>`
  - `java.lang.annotation.*`
- Various library classes for additional I/O and networking support have been added.
  - `java.io.BufferedReader`
  - `java.io.BufferedWriter`
  - `java.io.PrintWriter`
  - `javax.microedition.io.HttpConnection`



- `javax.microedition.io.HttpsConnection`
- `javax.microedition.io.SecureConnection`
- `javax.microedition.io.SecurityInfo`
- `javax.microedition.io.ServerSocketConnection`
- `javax.microedition.io.SocketConnection`
- `javax.microedition.io.UDPDatagramConnection`
- `javax.microedition.pki.Certificate`
- `javax.microedition.pki.CertificateException`
- The platform includes support for fine grained access control security functionality.
- A few library classes to support the access control functionality have been added.
  - `java.security.AccessController`
  - `java.security.AccessControlException`
  - `java.security.Permission`
  - `java.security.BasicPermission`
- A minimal set of library classes to enable internationalization and localization have been added.
  - `java.util.Locale`
  - `java.util.ResourceBundle`
  - `java.util.ListResourceBundle`
  - `java.util.MissingResourceException`
- Some library classes have been enhanced with generics semantics.
  - `java.lang.Class<T>`
  - `java.util.Hashtable<K,V>`
  - `java.util.Enumeration<E>`
  - `java.util.Stack<E>`
  - `java.util.Vector<E>`



## Goals, Requirements, and Scope

---

---

### 2.1 Goals

The goal of the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* is to standardize a highly portable, minimum-footprint Java application-development platform for resource-constrained, smart card and secure token devices.

The devices targeted by the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* have the following general characteristics:

- At least 192 kB of total memory budget available for the Java platform (see [Section 2.2.1, “Hardware Requirements” on page 2-2](#)).
- A 32-bit processor
- Low power consumption, operating with an external or magnetically induced power source
- Connectivity to some kind of network, often with a wireless, intermittent connection and with limited bandwidth

Smart cards, secure tokens and secure portable storage devices are some, but not all, of the devices that might be supported by this specification.

More specifically, the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* defines a Java technology application-development platform with the following characteristics and goals:

- **Keep footprint small.**

Smart card devices are manufactured and deployed in very large quantities (hundreds of thousands, millions, or even tens of millions of units per year). To meet the needs of the cost-conscious card issuer and smart card manufacturers, the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*

defines a “lowest common denominator” standard that includes only the minimal Java platform features and APIs for a wide range of smart card vertical market segments.

- **Focus on application programming rather than systems programming.**

The Java Card platform is intended to be primarily an *application development platform*, rather than a systems programming environment. This has certain implications for the Java platform features and APIs to be included in this specification. First, the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* shall include only high-level libraries that provide sufficient programming power for the application developer. Second, we emphasize the importance of generality and portability. The *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* shall not provide any APIs that are specific to a certain device category, vertical market or system functionality.

- **Enable dynamic downloading of applications and encourage third-party application development.**

Unlike the past (when smart card devices usually came with a feature set that was hard-coded at the factory), smart card manufacturers are increasingly looking for solutions that allow them to build *extensible* products that support the dynamic downloading of interactive content from content providers and third party developers. One of the key goals of the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* is to define an environment that is well-suited for the dynamic, secure downloading of applications written in the Java programming language (Java applications) over different kinds of networks to these Java Card technology enabled devices.

The focus on dynamically delivered Java applications means that this specification is intended not only for hardware manufacturers and their system programmers, but also for *third party application developers*. In fact, we assume that once these Java Card technology enabled smart cards become more accessible to mainstream Java programmers, the vast majority of application developers for Java Card devices will be third party developers rather than smart card manufacturers themselves.

---

## 2.2 Requirements

### 2.2.1 Hardware Requirements

The Java Card platform is intended to run on a wide variety of smart card and secure token devices. The underlying hardware capabilities of these devices vary considerably and, therefore, the *Virtual Machine Specification, Java Card Platform,*

*v3.0.1, Connected Edition* does not impose any specific hardware requirements other than memory requirements. Even for memory limits, the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* defines only minimum limits for the Virtual Machine and the core platform library classes (`java.*`, `javax.*`).

The *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* assumes that:

- At least 176 kilobytes of non-volatile memory is available for the virtual machine, virtual machine core platform libraries, and the persistent object heap.
- At least 16 kilobytes of volatile memory is available for the virtual machine runtime (for example, the object heap.)

Typical Java Card platform target devices may require significantly more memory than the minimum to support the application containers, Java Card platform framework layers and industry specific extension libraries. Java Card platform smart card devices typically have as much as 512 kilobytes of non-volatile<sup>1</sup> read-only memory, 128 kilobytes of non-volatile read-write memory and 24 kilobytes of volatile<sup>2</sup> memory.

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

Smart card devices are not battery powered and have a wide range of variance with respect to timer and real time clock capabilities. The *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* assumes that the platform may only be able to support a low precision interval timer. In addition, upon power up, the platform is able to service requests for the current wall clock time only when re-synchronized with an external time source.

---

1. The term *non-volatile* is used to indicate that the memory is expected to retain its contents between the user turning the device “on” or “off”. For the purposes of the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*, it is assumed that non-volatile memory is usually accessed in read mode, and that special setup may be required to write to it. Examples of non-volatile memory include ROM, Flash and battery-packed SDRAM. The *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* does not define which memory technology a device must have, nor does it define the behavior of such memory in a power-loss scenario.

2. The term *volatile* is used to indicate that the memory is not expected to retain its contents between the user turning the device “on” or “off”. For the purposes of the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*, it is assumed that volatile memory can be read and written to directly without any special setup. The most common type of volatile memory is DRAM.

## 2.2.2 Software Requirements

Like the hardware capabilities, the system software in Java Card platform target devices varies considerably. For instance, some of the devices may have a full-featured operating system that supports multiple, concurrent operating system processes and a hierarchical file system. Many other devices may have extremely limited system software without a notion of a file system. Faced with such variety, the Java Card platform makes minimal assumptions about the underlying system software.

Generally, the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* assumes that a minimal *host operating system* or kernel is available to manage the underlying hardware. This host operating system must provide at least one schedulable entity to run the Java virtual machine. The host operating system does not need to support separate address spaces or processes, nor does it have to make any guarantees about the real-time scheduling or latency behavior.

## 2.2.3 Java Card Requirements

A Java Card platform configuration specification shall generally define a *subset* of the Java technology features and libraries provided by the Java Platform, Standard Edition (Java SE platform). Consequently, rather than providing a complete description of all the supported features, the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* shall only define the variances and differences compared to the full *Java Language Specification (JLS)* and *Java™ Virtual Machine Specification (JVMs)*. If something is not explicitly specified in the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*, then it is assumed that a virtual machine conforming to the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* MUST comply with the JLS and JVMs.

Note that the absence of optional features in the Java Card platform does not preclude the use of various *implementation-level optimizations*. For instance, at the implementation level, alternative bytecode execution techniques (such as Just-In-Time compilation) or class representation techniques can be used as long as the observable user-level semantics of the implementation remain the same as defined by the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*.

---

## 2.3 Scope

The *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* shall address the following areas:

- Java programming language and virtual machine features
- Core Java libraries (`java.lang.*`, `java.util.*`)
- Input/output (`java.io.*`)
- Security (`java.security.*`)
- Networking (`javax.microedition.io.*`)
- Internationalization (`java.util.*`)

This *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* shall not address the following areas:

- Smart Card device initialization and card lifecycle management
- Application installation and life-cycle management
- Inter-Application Communication
- Event handling
- High-level application model (the interaction between the user and the application)

These features on the Java Card platform are addressed by the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* and the *Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition*.

The remainder of this specification is organized in the following sequence. The specification starts with a discussion of the high-level architecture of a typical Java Card platform environment. Then, the specification compares the Java language and virtual machine features of a virtual machine conforming to the Java Card platform to a conventional Java environment meeting the full *Java Language Specification* and *Java™ Virtual Machine Specification*. Finally, the specification describes the Java libraries included in the Java Card platform.





# High-level Architecture and Security

---

This chapter discusses the high-level architecture of a typical Java Card platform environment. This discussion serves as a starting point for more detailed specification requirements provided in later chapters.

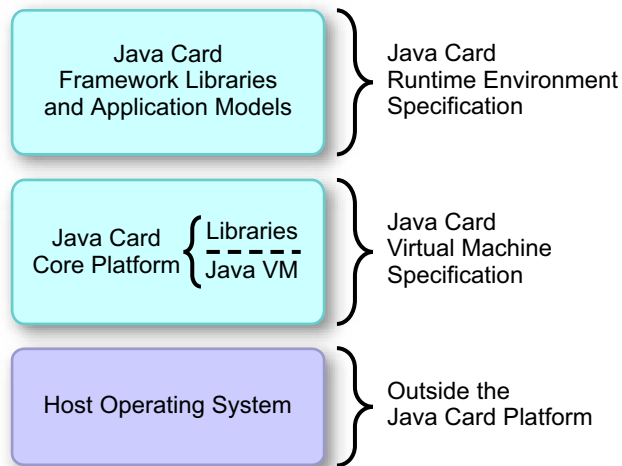
---

## 3.1 Java Card VM High-Level Architecture

The high-level architecture of a typical Java Card platform device is illustrated in [FIGURE 3-1](#). At the heart of a Java Card platform implementation is the Java virtual machine, which, apart from specific differences defined later in this specification, is compliant with the *Java Virtual Machine Specification* and *Java Language Specification*. The virtual machine typically runs on top of a Host Operating System that is outside the scope of the Java Card platform.

On top of the virtual machine are the Java *libraries*. Some of these libraries are defined by the Java Card platform itself, as represented in [FIGURE 3-1](#). In addition, Java Card platform framework layers define additional libraries and features that sit on top of the core platform layer.

**FIGURE 3-1** High-level Architecture



---

## 3.2 Application Management

The Java Card platform is capable of storing applications persistently and the Card Manager on the smart card device has capabilities for managing the applications that have been stored in the device. At the high level, *application management* refers to the ability to:

- Download and install Java applications
- Inspect existing Java applications stored on the device
- Select and launch Java applications
- Delete existing Java applications (if applicable)

The Java Card platform allows multiple Java applications to execute concurrently. It is up to the particular Java Card platform implementation to decide if the execution of multiple Java applications is supported by utilizing the multitasking capabilities (if they exist) of the underlying host operating system, or by managing concurrent Java applications in the virtual machine layer.

Due to significant variations and feature differences among potential Java Card platform devices, the details of application management are highly device-specific and implementation-dependent. The responsibilities of the Card Manager and the application distribution format are described in *Runtime Environment Specification, Connected Edition*.

---

## 3.3 Security

Due to its inherent security architecture, the Java technology-development platform is particularly well-suited to security-critical environments. The security model provided by the Java SE platform provides developers with a powerful and flexible security framework that is built into the Java platform. Developers can create fine-grained security policies and articulate independent permissions for individual applications, all while appearing transparent to the end user.

Retaining all the features devoted to security in the Java SE platform would require a memory budget which the Java Card platform can ill afford. Therefore, some simplifications are necessary when defining the security model for the Java Card platform. The security model of the Java Card platform is defined at the following three levels:

- *Low-level security*

Low-level security, also known as *virtual machine security*, ensures that the applications running in the virtual machine follow the semantics of the Java programming language, and that an ill-formed or maliciously-encoded class file does not crash or in any other way harm the target device.

- *Application-level security*

Application-level security means that a Java application running on a smart card device can access only those libraries, system resources and other components that the smart card device and the Java application environment allows it to access based on the security policy defined for the application.

- *End-to-end security*

End-to-end security refers to a model that guarantees that any transaction initiated on a device is protected along the entire path from the device to/from the entity providing the services for that transaction (e.g. a server located on the Internet). Encryption or other means may be necessary to achieve this. End-to-end security is defined in the Java Card platform framework layers of the Java Card platform that provide facilities for end-to-end software development. These facilities are described in *Runtime Environment Specification, Connected Edition*.

In addition to these security levels, the Java Card platform defines the application firewall and a secure transaction models as described in the *Runtime Environment Specification, Connected Edition*.

In the following sections, we take a more detailed look at each of these levels.

### 3.3.1 Low-Level (Virtual Machine) Security

A key requirement for a Java virtual machine in a mobile information device is low-level virtual machine security. An application running in the virtual machine must not be able to harm the device in which the virtual machine is running or crash the device. In a standard Java virtual machine implementation, this constraint is guaranteed by the class file verifier, which ensures that the bytecodes and other items stored in class files cannot contain illegal instructions, cannot be executed in an illegal order, and cannot contain references to invalid memory locations or memory areas that are outside the Java platform-object memory (the object heap). In general, the role of the class file verifier is to ensure that class files loaded into the virtual machine do not execute in any way that is not allowed by the *Java Virtual Machine Specification*.

As will be explained in more detail in [Section 5.3, “Class File Verification” on page 5-4](#) the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* requires that a Java virtual machine conforming to the Java Card platform standard must be able to reject invalid class files. This is guaranteed by the class file verification technology presented in [Section 5.3, “Class File Verification” on page 5-4](#).

### 3.3.2 Application-Level Security

Even though class file verification plays a critical role in ensuring the security of the Java platform, the security provided by the class file verifier is insufficient by itself. The class file verifier can only guarantee that the given program is a valid Java application and nothing more. There are still several other potential security threats that will go unnoticed by the verifier. For instance, access to external resources such as the file system, native libraries or the network is beyond the scope of the class file verifier. By *application-level security*, we mean that a *Java application can access only those libraries, system resources and other components that the device and the Java application environment allows it to access based on the security policy defined for the application*. The details of application-level security are discussed below.

#### 3.3.2.1 Sandbox Model

In the Java Card platform, application-level security is accomplished by using a metaphor of a closed “sandbox.” An application must run in a closed environment in which the application can access only those libraries that have been defined by the Java Card platform and Java Card platform framework, extension libraries and other classes supported by the device. Java applications cannot escape from this sandbox or access any libraries or resources that are not part of the predefined functionality. The sandbox ensures that a malicious or possibly erroneous application cannot gain access to system resources.

More specifically, the Java Card platform sandbox model requires that:

- Class files must be properly verified and guaranteed to be valid Java applications. (Class file verification is discussed in more detail in [Section 5.3, “Class File Verification” on page 5-4](#)).
- The downloading, installation, and management of Java applications on the smart card device takes place in such a way that the application programmer cannot modify or bypass the standard class loading mechanisms of the virtual machine.
- A closed, predefined set of Java APIs is available to the application programmer, as defined by the Java Card platform, framework layers, market-specific extension layers and manufacturer-specific classes.
- The set of native functions accessible to the virtual machine is closed, meaning that the application programmer cannot download any new libraries containing native functionality or access any native functions that are not part of the Java libraries provided by the Java Card platform, framework layers, market-specific extension layers or manufacturer-specific classes. Services performed by native functions **MUST** be performed only after verifying that the method parameters are within bounds and all objects passed in as parameters are accessible according to the context isolation rules described in the *Runtime Environment Specification, Connected Edition*.

Java Card platform framework layers may provide additional security solutions. Java Card platform framework layers also define which additional APIs are available to the application programmer.

### 3.3.2.2 Protecting System Classes

A central requirement for the Java Card platform is the ability to support dynamic downloading of Java applications to the virtual machine. A possible application-level security hole in the Java virtual machine would be exposed if the downloaded applications could override or extend the set of the system classes provided in packages `java.*`, `javax.*` or other Java Card platform framework, or market-specific extension packages, or manufacturer-specific packages. A Java Card platform implementation **MUST** ensure that the application programmer cannot override, modify, or add any classes to these protected system packages.

For security reasons, it is also required that the application programmer is not allowed to manipulate the class file lookup order in any way. Class file lookup order is discussed in more detail in [Section 5.4.3, “Class File Lookup Order” on page 5-7](#).

### 3.3.2.3 Additional Restrictions on Dynamic Class Loading

Dynamic loading of Java applications is a key feature of the Java Card platform. However, the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* defines the class loading mechanism of a virtual machine conforming to the

Java Card platform to be implementation-dependent, with one important restriction: by default, a Java application can load application classes only from its own JAR file. This restriction ensures that Java applications on a device cannot interfere with each other or steal data from each other. Additionally, this ensures that a third-party application cannot gain access to the private or protected components of the Java class files that the device manufacturer or a service provider may have provided as part of the system applications. JAR files and applications representation formats are discussed in more detail in [Section 5.4, “Class File Format and Class Loading” on page 5-5](#).

### 3.3.3 End-to-End Security

A device conforming to the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* is typically a part of an end-to-end solution such as a wireless network or a payment terminal network. These networks commonly require a number of advanced security solutions (such as encryption) to ensure safe delivery of data and code between server machines and client devices. Given the broad diversity of network infrastructure in the world, the Java Card platform does not mandate any specific end-to-end security mechanism. Specific end-to-end security solutions are assumed to be implementation-dependent. The Java Card platform framework layer includes infrastructure for implementing end-to-end security solutions and is described in the *Runtime Environment Specification, Connected Edition*.

# Adherence to the Java Language Specification

---

The general goal for a virtual machine conforming to the Java Card platform is to be as compliant with the *Java Language Specification* as is feasible within the strict memory limits of Java Card platform target devices. This chapter summarizes the differences between a virtual machine conforming to Java Card platform and the Java virtual machine of the Java SE platform. Except for the differences indicated herein, a virtual machine conforming to the Java Card platform **MUST** be compatible with *The Java Language Specification, Third Edition*, by James Gosling, Bill Joy, and Guy L. Steele (Addison-Wesley, 2005), ISBN 0-321-24678-0.

---

**Note** – For the remainder of this specification, the *Java Language Specification* will be referred to as JLS. Sections within the *Java Language Specification* will be referred to using the § symbol. For example, (JLS §4.2.4).

---

---

## 4.1 No Floating Point Support

The main language-level difference between the full *Java Language Specification* and this *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* is that a JVM implementation supporting the Java Card platform does not have floating point support. Floating point support was removed because the majority of Java Card platform target devices do not have hardware floating point support and the cost of supporting floating point in software was considered too high.

This means that a JVM implementation supporting the Java Card platform shall not allow the use of floating point literals (JLS §3.10.2), floating point types and values (JLS §4.2.3) and floating point operations (JLS §4.2.4). For further information, refer to [Section 5.1, “No Floating Point Support” on page 5-1](#) in this *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*.

---

## 4.2 No Finalization of Class Instances

Java Card platform libraries do not include the `methodObject.finalize()`. Therefore, a virtual machine conforming to Java Card platform does not support finalization of class instances (JLS §12.6). No application built to conform to the Java Card platform shall require that finalization is available.

---

## 4.3 Exception and Error Handling Limitations

A virtual machine conforming to the Java Card platform MUST generally support *exception* handling as defined in JLS Chapter 11, with the exception that *asynchronous exceptions* (JLS §11.3.2) are not supported.

In contrast, the set of *error* classes included in the Java Card platform libraries is limited, and consequently the error handling capabilities of the Java Card platform are considerably more limited. This is because of the following two reasons:

- In embedded systems, recovery from error conditions is usually highly device-specific.  
Application programmers cannot be expected to worry about device-specific error handling mechanisms and conventions.
- As specified in JLS §11.5, class `java.lang.Error` and its subclasses are exceptions from which programs are not ordinarily expected to recover.

Implementing the error handling capabilities fully according to the *Java Language Specification* is rather expensive, and mandating the presence and handling of all the error classes would impose an overhead on the virtual machine implementation.



A virtual machine conforming to the Java Card platform MUST support the set of Error classes defined in [Section 6.2, “Classes Derived from Java SE Platform”](#) on [page 6-2](#). When encountering any other error, the implementation MUST behave in one of the following ways:

- The virtual machine halts in an implementation-specific manner.
- The virtual machine throws an Error that is the nearest Java Card platform-supported superclass of the Error class representing the error condition that must be thrown according to the *Java Language Specification*

If the virtual machine conforming to the Java Card platform implements additional error checks that are part of the *Java Language Specification* but that are not required by the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*, the implementation MUST throw the nearest Java Card platform-supported superclass of the Error class representing the error condition that is defined by the *Java Language Specification*.



# Adherence to Java Virtual Machine Specification

---

The general goal for a virtual machine conforming to the Java Card platform is to be as compliant with the *Java Virtual Machine Specification* as possible within the strict memory constraints of Java Card platform target devices. This chapter summarizes the differences between a virtual machine conforming to the Java Card platform and the Java virtual machine of the Java SE platform. Except for the differences indicated herein, a virtual machine conforming to the Java Card platform **MUST** be compatible with the Java virtual machine as specified in the *Java Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999), ISBN 0-201-43294-3 and as amended by *Java Virtual Machine Specification (JVMS) for Java 5* and by the Java Development Kit, Version 1.6 revisions to chapter 4 (Class File Format) of *Java Virtual Machine Specification*.

---

**Note** – For the remainder of this specification, the *Java Virtual Machine Specification* and the JDK version 1.5 and JDK version 1.6 amendments mentioned above are referred to as JVMS. Sections within the *Java Virtual Machine Specification* are referred to using the § symbol. For example, (JVMS §2.4.3).

---

---

## 5.1 No Floating Point Support

A Java virtual machine supporting the Java Card Platform does not have floating point support. Floating point support was removed because the majority of Java Card Platform target devices do not have hardware floating point support, and since the cost of supporting floating point in software was considered too high. Consequently, a Java virtual machine supporting the Java Card Platform shall not support the bytecodes listed in [TABLE 5-1](#):

**TABLE 5-1** Bytecodes Not Supported by Java Virtual Machine

Type	Bytecode
Constants	fconst_0, fconst_1, fconst_2, dconst_0, dconst_1
Loads	fload, fload_x, dload, dload_x
Stores	fstore, fstore_x, dstore, dstore_x
Arrays	faload, daload, fastore, dastore, newarray T_DOUBLE, newarray T_FLOAT
Arithmetic	fadd, dadd, fsub, dsub, fmul, dmul, fdiv, ddiv, frem, drem, fneg, dneg, fcmpl, fcmpg, dcmpl, dcmpg
Conversion	i2f, f2i, i2d, d2i, l2f, l2d, f2l, d2l, f2d, d2f
Returns	freturn, dreturn

All user-supplied classes and methods running on top of a Java virtual machine supporting the Java Card Platform must satisfy the following constraints:

- No method shall use any of the above forbidden bytecodes.
- No field can have as its type `float` or `double`, or an array of one of those types.
- No method can have an argument or return type of type `float` or `double` or an array whose component type is `float` or `double`.
- No constant pool entry can be of type `CONSTANT_Float` or `CONSTANT_Double`.
- No constant pool entry can be of type `CONSTANT_Class` in which the class is an array of type `float` or `double`.

Due to the lack of floating point support, the following sections and subsections of the *Java Virtual Machine Specification* (JVMS) are not applicable to a Java virtual machine supporting the Java Card Platform: §2.4.3, §2.4.4, §2.18, §3.3.2 and §3.8. In addition, all the other parts of the JVMS that refer to floating point data types (`float` or `double`) or operations are beyond the scope of this *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition*.

---

## 5.2 Features Eliminated from the Virtual Machine

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

- User-defined class loaders (JVMS §5.3.2)
- Thread groups and daemon threads (JVMS §2.19, §8.12)
- Finalization of class instances (JVMS §2.17.7)
- Asynchronous exceptions (JVMS §2.16.1)

In addition, a virtual machine conforming to the Java Card platform has a significantly more limited set of error classes than a full Java SE platform virtual machine.

Applications written for the Java Card platform **MUST** not rely on any of the features above. Each of the features in this list is discussed in more detail below.

### 5.2.1 User-Defined Class Loaders

A virtual machine conforming to the Java Card platform does not support user-defined, Java platform-level class loaders (JVMS §5.3, §2.17.2). A virtual machine conforming to the Java Card platform **MUST** use platform defined class loaders that cannot be overridden, replaced, or reconfigured. The platform managed class loader hierarchy used by the Java Card platform is described in *Runtime Environment Specification, Connected Edition*. The elimination of user-defined class loaders is part of the security restrictions presented in [Section 3.3.2.1, “Sandbox Model” on page 3-4](#).

### 5.2.2 Thread Groups and Daemon Threads

A virtual machine conforming to the Java Card platform implements multithreading, but does not have support for thread groups or daemon threads (JVMS §2.19, §8.12). Thread operations such as starting threads can be applied only to individual thread objects. If application programmers want to perform thread operations for groups of threads, explicit collection objects must be used at the application level to store the thread objects.

## 5.2.3 Finalization of Class Instances

Java Card platform libraries do not include the method `Object.finalize()`. Therefore, a virtual machine conforming to the Java Card platform does not support finalization of class instances (JVMS §2.17.7). No application running on top of a virtual machine conforming to the Java Card platform shall require that finalization be available.

## 5.2.4 Errors and Asynchronous Exceptions

As discussed earlier in [Section 4.3, “Exception and Error Handling Limitations”](#) on [page 4-2](#), the error handling capabilities of a virtual machine conforming to the Java Card platform are limited.

A virtual machine conforming to the Java Card platform **MUST** support the set of Error classes defined in [Section 6.2, “Classes Derived from Java SE Platform”](#) on [page 6-2](#). When encountering any other error, the implementation **MUST** behave in one of the following ways:

- The virtual machine halts in an implementation-specific manner.
- The virtual machine throws an Error that is the nearest Java Card platform-supported superclass of the Error class representing the error condition that must be thrown according to the *Java Virtual Machine Specification*.

If the virtual machine conforming to the Java Card platform implements additional error checks that are part of the *Java Virtual Machine Specification* but that are not required by the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*, the implementation **MUST** throw the nearest Java Card platform-supported superclass of the Error class representing the error condition that is defined by the *Java Virtual Machine Specification*.

A virtual machine conforming to the Java Card platform **MUST** generally support exception handling as defined by *Java Virtual Machine Specification*. However, a virtual machine conforming to the Java Card platform does not support *asynchronous exceptions* (JVMS §2.16.1).

---

## 5.3 Class File Verification

Like the Java virtual machine of the Java SE platform, a virtual machine conforming to the Java Card platform must be able to reject invalid class files. This means that a Java Card platform implementation must support class file verification.

Class file verification in the Java Card platform **MUST** be implemented using the type checking approach based on the `StackMapTable` attribute defined in the *The Java Development Kit, Version 1.6 revisions to chapter 4 (Class File Format) of Java Virtual Machine Specification* (JVMS §4.11). But, unlike JDK version 1.6 on the Java SE platform, if the type checking fails on the Java Card platform, the class file **MUST** be rejected.

Runtime class file verification guarantees type safety. Classes that pass the runtime verifier cannot, for example, violate the type system of the Java virtual machine or corrupt the memory. Unlike approaches based on code signing, such a guarantee does not rely on the verification attribute to be authentic or trusted. A missing, incorrect or corrupted verification attribute causes the class to be rejected by the runtime verifier.

---

## 5.4 Class File Format and Class Loading

An essential requirement for the Java Card platform is the ability to support dynamic downloading of Java applications and third party content. The dynamic class loading mechanism of the Java platform plays a central role in enabling this. This section discusses the application representation formats and class loading practices required of a virtual machine conforming to the Java Card platform.

### 5.4.1 Supported File Formats

A Java Card platform implementation must be able to read standard Java class files (defined in JVMS Chapter 4). In addition, a Java Card platform implementation must support compressed JAR files. Detailed information about the JAR file format is provided at

<http://java.sun.com/javase/6/docs/technotes/guides/jar/>. The Java Card application distribution units using JAR files are described in *Runtime Environment Specification, Connected Edition*.

Network bandwidth conservation is very important in low-bandwidth wireless networks. The compressed JAR file format provides 30 to 50 percent compression over regular class files without loss of any symbolic information or compatibility problems with existing Java technology systems.

A Java Card platform implementation must be able to read Java class files in the format supported by Java SE platform, JDK version 1.6.

---

**Note** – The class file format numbers used by JDK version 1. 6 are as follows:  
- The 50.\* version number identifies JDK version 1.6 class files.

---

A virtual machine conforming to the Java Card platform should be able to read Java class files in the formats listed above. However, the virtual machine is allowed to ignore certain class file attributes, which the Java Card platform implementation does not need. More specifically, the following attributes can be ignored by a Java Card platform implementation:

- The Synthetic attribute (JVMS §4.7.6)
- The SourceFile attribute (JVMS §4.7.7)
- The LineNumberTable attribute (JVMS §4.7.8)
- The LocalVariableTable attribute (JVMS §4.7.9)
- The Deprecated attribute (JVMS §4.7.10)
- The EnclosingMethod attribute (JVMS §4.8.6)
- The Signature attribute (JVMS §4.4.4)
- The LocalVariableTypeTable attribute (JVMS §4.8.13)
- The RuntimeVisibleAnnotations attribute (JVMS §4.8.15)
- The RuntimeInvisibleAnnotations attribute (JVMS §4.8.16) for annotations not defined by the *Runtime Environment Specification, Connected Edition*
- The RuntimeVisibleParameterAnnotations attribute (JVMS §4.8.17)
- The AnnotationDefault attribute (JVMS 4.8.19)
- The RuntimeInvisibleParameterAnnotations attribute (JVMS §4.8.18) for annotations not defined by the *Runtime Environment Specification, Connected Edition*

For historical reasons (JVMS p. 127), a virtual machine conforming to the Java Card platform is not required to check that the InnerClasses attribute (JVMS §4.7.5) is well-formed.

## 5.4.2 Public Representation of Java Applications and Resources

A Java application is considered to be “*represented publicly*” or “*distributed publicly*” when the system it is stored on is open to the public, and the transport layers and protocols that can access it are open standards. In contrast, a device can be part of a *closed network environment* where the vendor (such as the operator of a wireless network) controls all communication. In this case, the application is no longer represented publicly once it enters and is distributed via the closed network system.



Whenever Java applications intended for a Java Card platform device are represented publicly, the compressed JAR file representation format must be used. The JAR file must contain JDK version 1. 6 format class files, as defined in *The Java Development Kit, Version 1.6 revisions to chapter 4 (Class File Format) of Java Virtual Machine Specification*.

Additionally, the JAR file may contain *application-specific resource* files that can be loaded into the virtual machine by calling method `Class.getResourceAsStream(String name)`. See the Java Card platform library documentation for details.

### 5.4.3 Class File Lookup Order

The *Java Language Specification* and *Java Virtual Machine Specification* do not specify the order in which class files are searched when new class files are loaded into the virtual machine. At the implementation level, a typical Java virtual machine implementation utilizes a special environment variable `classpath` to define the lookup order.

This *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* assumes class file lookup order to be based on the standard manifest file attribute, *Class-Path* with the restrictions described in the next paragraph. The lookup strategy is typically defined as part of the application management implementation (see [Section 3.2, “Application Management” on page 3-2.](#))

Two restrictions apply to class file lookup order. Both of the following restrictions are important for security reasons:

- As explained in [Section 3.3.2.2, “Protecting System Classes” on page 3-5](#), a virtual machine conforming to the Java Card platform must guarantee that the application programmer cannot override, modify, or add new system classes (classes belonging to the Java Card platform, core platform, Java Card framework or market-specific extension libraries or manufacturer-specific classes) in any way.
- The application programmer must not be able to manipulate the class file lookup order of system classes or other previously installed extension library classes in any way.

## 5.4.4 Implementation Optimizations and Alternative Application Representation Formats

**Preloading/prelinking (“ROMizing”)** - A virtual machine conforming to the Java Card platform may choose to preload/prelink some classes. This technology is referred to informally as ROMizing.<sup>1</sup> Typically, small virtual machine implementations choose to preload all the system classes and perform application loading dynamically.

The actual mechanisms for preloading are implementation-dependent and beyond the scope of the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*. In all cases, the runtime effect and semantics of preloading/prelinking must be the same as if the actual class had been loaded in at that point. There must be no visible effects from preloading other than the possible speed-up in application launching. In particular, any class initialization that has a user-visible effect must be performed at the time the class would have first been loaded if it had not been preloaded into the system.

**Other implementation-level optimizations** - Java class files are not optimized for network transport in bandwidth-limited environments. Each Java class file is an independent unit that contains its own constant pool (symbol table), method, field and exception tables, bytecodes, exception handlers, and some other information. The self-contained nature of class files is one of the virtues of Java technology, allowing applications to be composed of multiple pieces that do not necessarily have to reside in the same location, making it possible to extend applications dynamically at runtime. However, this flexibility has its price. If Java applications were treated as a sealed unit, a lot of space could be saved by removing the redundancies in multiple constant pools and other structures, especially if full symbolic information was left out. Also, one of the desirable features of an application transport format in a limited-power computing environment is the ability to execute applications “in-place,” without any special loading or conversion process between the static representation and runtime representation. Standard Java class files are not designed for such execution.

The *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* mandates the use of compressed JAR files for Java applications that are represented and distributed publicly. However, in closed network environments (see the discussion in [Section 5.4.2, “Public Representation of Java Applications and Resources”](#) on page 5-6) and inside the virtual machine at runtime, alternative formats can be used. For instance, in low-bandwidth wireless networks it is often reasonable to use alternative, more compact transport formats at the network transport level to conserve network bandwidth. Similarly, when storing downloaded applications in Java Card platform compliant smart card devices, more compact representations can be used, as long as the observable user-level semantics of the

---

1. The term *ROMizing* is somewhat misleading, since this technology can be used independently of any specific memory technology. ROMized class files do not necessarily have to be stored in read-only- memory (ROM).

applications remain the same as the original representation. The alternative formats may not be used for representing or distributing Java Card applications publicly (i.e., the public representation format of Java Card applications must always be as defined in [Section 5.4.2, “Public Representation of Java Applications and Resources”](#) on page 5-6.)

The definition of alternative application representations is assumed to be implementation-dependent and outside the scope of the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*.



# Java Card Platform Libraries

---

---

## 6.1 Overview

The Java SE platform and the Java Platform, Enterprise Edition (Java EE) provide a very rich set of libraries for the development of applications for desktop computers and server machines. Unfortunately, these libraries require multiple megabytes of memory to run, and are therefore unsuitable for small devices with limited resources.

A general goal for designing the libraries for the Java Card platform is to provide a minimum useful set of libraries for practical application development for the various market segments for smart card devices. As explained in [Section 2.1, “Goals” on page 2-1](#), the Java Card platform is a “lowest common denominator” standard that includes only the minimal Java platform features and APIs for a wide range of smart card devices. Given the strict memory constraints and differing features of smart card devices, it is virtually impossible to create a set of libraries that would be ideal for everyone. No matter where the bar for feature inclusion is set, the bar is inevitably going to be too low for some smart card devices and users, and possibly too high for others.

To ensure upward compatibility with larger editions of the Java platform, the majority of the libraries included in the Java Card platform are a subset of the Java SE platform, the Java EE platform, and Java Platform, Micro Edition (Java ME platform). While upward compatibility is a very desirable goal, Java SE and Java EE class libraries have strong internal dependencies that make subsetting them difficult in important areas such as security, input-output, networking, and storage. These dependencies are a natural consequence of the design evolution and reuse that has taken place over time during the development of Java libraries. Unfortunately, these dependencies make it difficult to take just one part of the libraries without including several others. For this reason, some Java ME class libraries, especially in the area of networking are included.

The Java Card platform libraries defined by the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* can be divided into the following two categories:

- Classes that are a subset of standard Java SE class libraries
- Classes that are a subset of standard Java ME class libraries (but which can be mapped onto the Java SE platform)

Classes belonging to the former category are located in packages `java.lang.*`, `java.util`, `java.io`, and `java.security`. A detailed list of these classes is presented in [Section 6.2, “Classes Derived from Java SE Platform”](#) on page 6-2.

Classes belonging to the latter category are located in package `javax.microedition.io`, and `javax.microedition.pki`. These classes are discussed in [Section 6.3, “Classes Derived from Java ME Platform”](#) on page 6-10.

---

## 6.2 Classes Derived from Java SE Platform

The Java Card platform supports a number of classes that have been derived from the Java SE 6 platform. Each class that has the same name and package name as a Java SE class is identical to or a subset of the corresponding Java SE class. The semantics of the classes and their included methods in the subset are not changed. The classes do not include any public or protected methods or fields that are not available in the corresponding Java SE classes.

For a definitive reference on the classes listed in this section, refer to the *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition* for all the Java Card platform and framework libraries in Javadoc™ tool format.

### 6.2.1 System Classes

Java SE class libraries include several classes that are intimately coupled with the Java virtual machine. Similarly, several standard Java technology tools assume the presence of certain classes in the system. For instance, the standard Java technology compiler (`javac`) generates code that requires that certain functions of classes `String` and `StringBuffer` and `StringBuilder` be available. The system classes included in the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* are provided in the following list. Each of these classes is a subset of the corresponding class in the Java SE platform.

`java.lang.Object`

```
java.lang.Class<T>
java.lang.Enum
java.lang.Runtime
java.lang.System
java.lang.Thread
java.lang.Runnable (interface)
java.lang.String
java.lang.StringBuffer
java.lang.StringBuilder
java.lang.Throwable
java.lang.Iterable<T> (interface)
java.lang.annotation.Annotation
java.lang.annotation.AnnotationFormatError
java.lang.annotation.Documented
java.lang.annotation.ElementType
java.lang.annotation.Inherited
java.lang.annotation.Retention
java.lang.annotation.RetentionPolicy
java.lang.annotation.Target
java.lang.Override
java.lang.SuppressWarnings
```

## 6.2.2 Data Type Classes

The following basic data type classes from package `java.lang` are supported. Each of these classes is a subset of the corresponding class in the Java SE platform.

```
java.lang.Boolean
java.lang.Byte
```

```
java.lang.Short  
java.lang.Integer  
java.lang.Long  
java.lang.Character  
java.lang.Void
```

## 6.2.3 Collection Classes

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

```
java.util.Vector<E>  
java.util.Stack<E>  
java.util.Hashtable<K,V>  
java.util.Enumeration<E> (interface)
```

## 6.2.4 Input-Output Classes

The following classes from package `java.io` are supported.

```
java.io.InputStream  
java.io.OutputStream  
java.io.ByteArrayInputStream  
java.io.ByteArrayOutputStream  
java.io.DataInput (interface)  
java.io.DataOutput (interface)  
java.io.DataInputStream  
java.io.DataOutputStream  
java.io.Reader  
java.io.BufferedReader
```



```
java.io.BufferedWriter
java.io.Writer
java.io.PrintWriter
java.io.InputStreamReader
java.io.OutputStreamWriter
java.io.PrintStream
```

## 6.2.5 Calendar and Time Classes

The Java Card platform includes a small subset of the standard Java SE classes `java.util.Calendar`, `java.util.Date`, and `java.util.TimeZone`. To conserve space, the *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* requires only one time zone to be supported. By default, this time zone is GMT. Additional time zones may be provided by manufacturer-specific implementations of the Java Card platform, as long as the time zones are compatible with those provided by the Java SE platform.

```
java.util.Calendar
java.util.Date
java.util.TimeZone
```

## 6.2.6 Additional Utility Classes

Two additional utility classes are provided. Class `java.util.Random` provides a simple pseudo-random number generator that is useful for implementing security applications. Class `java.lang.Math` provides methods `min`, `max` and `abs` that are frequently used by other Java library classes. The `java.util.EventObject` class and the `java.util.EventListener` class provide the basic event functionality required for web applications.

```
java.util.Random
java.lang.Math
java.util.EventObject
java.util.EventListener
```

## 6.2.7 Additional Security Classes

The Java Card platform includes a small subset of Java SE platform security classes which support Access Control security functions.

```
java.security.AccessController  
java.security.Permission  
java.security.BasicPermission
```

## 6.2.8 Basic Internationalization and Localization Classes

The Java Card platform also includes a small subset of Java SE platform internationalization framework to enable internationalization and localization functionality.

```
java.util.Locale  
java.util.ResourceBundle  
java.util.ListResourceBundle
```

## 6.2.9 Exception and Error Classes

Since the libraries included in the Java Card platform are generally intended to be highly compatible with Java SE class libraries, the library classes included in the Java Card platform MUST throw precisely the same exceptions as regular Java SE classes. Consequently, a fairly comprehensive set of exception classes has been included.

In contrast, as explained in [Section 4.3, “Exception and Error Handling Limitations” on page 4-2](#), the error handling capabilities of the Java Card platform are limited. By default, a virtual machine conforming to the Java Card platform is required to support only the error classes listed below in [Section 6.2.9.2, “Error Classes” on page 6-8](#).

### 6.2.9.1 Exception Classes

```
java.lang.Exception
java.lang.ArithmeticException
java.lang.ArrayIndexOutOfBoundsException
java.lang.ArrayStoreException
java.lang.AssertionError
java.lang.ClassCastException
java.lang.ClassNotFoundException
java.lang.IllegalAccessException
java.lang.IllegalArgumentException
java.lang.IllegalStateException
java.lang.IllegalMonitorStateException
java.lang.IllegalThreadStateException
java.lang.IndexOutOfBoundsException
java.lang.InstantiationException
java.lang.InterruptedIOException
java.lang.NegativeArraySizeException
java.lang.NoSuchFieldError
java.lang.NullPointerException
java.lang.NumberFormatException
java.lang.RuntimeException
java.lang.SecurityException
java.lang.StringIndexOutOfBoundsException

java.util.EmptyStackException
java.util.NoSuchElementException
java.util.MissingResourceException

java.security.AccessControlException
```

```
java.io.EOFException
java.io.InterruptedIOException
java.io.IOException
java.io.UnsupportedEncodingException
java.io.UTFDataFormatException
```

## 6.2.9.2 Error Classes

```
java.lang.Error
java.lang.NoClassDefFoundError
java.lang.OutOfMemoryError
java.lang.VirtualMachineError
```

## 6.2.10 Internationalization

**Character sets and character case conversion support** - A Java Card platform implementation is required to support Unicode characters. Character information is based on the *Unicode Standard*, version 3.0. However, since the full character tables required for Unicode support can be excessively large for devices with tight memory budgets, by default the character property and case conversion facilities in the Java Card platform assume the presence of ISO Latin-1 range of characters only. More specifically, implementations must provide support for character properties and case conversions for characters in the “Basic Latin” and “Latin-1 Supplement” blocks of Unicode 3.0. Other Unicode character blocks may be supported as necessary.

**Character encodings** - The Java Card platform includes limited support for the translation of Unicode characters to and from a sequence of bytes. In the Java SE platform, this is done using objects called Readers and Writers, and this same mechanism is utilized in the Java Card platform using the `InputStreamReader` and `OutputStreamWriter` classes with identical constructors.

```
new InputStreamReader(InputStream is);
new InputStreamReader(InputStream is, String enc);
new OutputStreamWriter(OutputStream os);
```

```
new OutputStreamWriter(OutputStream os, String enc);
```

If the `enc` parameter is present, it is the name of the encoding to be used. Where it is not, a default encoding (defined by the system property `javacard.encoding`) is used.

A virtual machine conforming to *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* must support the following character encodings:

**TABLE 6-1** Character Encodings

Name	Canonical Name for java.io and java.lang API	Description
ISO-8859-1	ISO8859_1	ISO-8859-1, Latin Alphabet No. 1
UTF8	UTF8	Eight-bit Unicode (or UCS) Transformation Format

Additional converters may be provided by particular implementations. If a converter for a certain encoding is not available, an `UnsupportedEncodingException` will be thrown. For official information on character encodings in the Java SE platform, refer to

<http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html>

**Localization support** - The Java Card platform includes limited support for localization based on a small subset of Java SE platform internationalization framework, see [Section 6.2.8, “Basic Internationalization and Localization Classes”](#) on page 6-6.

## 6.2.11 Property Support

A virtual machine conforming to *Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition* does not include the `java.util.Properties` class from the Java SE platform. In the Java SE platform, that class is used for storing system properties such as the name of the host operating system, version number of the virtual machine and so on.

In the Java Card platform, the set of platform properties described in [TABLE 6-2](#) is available. These properties and other manufacturer specific extensions can be accessed by calling the method `System.getProperty(String key)`.

**TABLE 6-2** Java Card System Properties

Key	Explanation	Value
<code>javacard.encoding</code>	Default character encoding	Default value: <code>"ISO8859_1"</code>
<code>line.separator</code>	Default line separator	Default value: <code>"\n"</code>

Property `javacard.encoding` describes the default character encoding name. This information is used by the system to find the correct class for the default character encoding in supporting internationalization.

Property `line.separator` describes the default line separator. This is used by some of the I/O classes to terminate output lines written to an output stream.

Note that the properties defined above can be extended by smart card device manufacturers.

Manufacturer-specific property definitions should be prefixed with the same package name that the manufacturer-specific classes use (e.g., `com.companyname.propertyname`). The `microedition` namespace, as well as the `java`, `javax`, `javacard` and `javacardx` namespaces are reserved, and may only be extended by the Java Card platform specifications.

## 6.3 Classes Derived from Java ME Platform

The Java Card platform supports a number of classes that have been derived from the Java ME platform. Each class that has the same name and package name as a Java ME class is identical to the corresponding Java ME class. The semantics of the classes and their included methods are not changed. The classes do not include any public or protected methods or fields that are not available in the corresponding Java ME classes.

For a definitive reference on the classes listed in this section, refer to the *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition* for all the Java Card platform and framework libraries in Javadoc tool format.

## 6.3.1 The Generic Connection Framework

The Generic Connection framework classes are a generalization of the Java SE technology-network and I/O classes. The goal for this generalized design is to be a functional subset of Java SE classes, which can easily map to common low-level hardware or to any Java SE technology implementation, but with better extensibility, flexibility and coherence in supporting new devices and protocols.

Instead of using a collection of different kinds of abstractions for different forms of communication, a set of related abstractions are used at the application programming level.

The following code line is the generic form used to request a network connection. An implementation specific instance implementing the specified protocol interface is returned.

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

The syntax of the `Connector.open` parameter strings should generally follow the *Uniform Resource Identifier* (URI) syntax as defined in the IETF standard RFC3986 (<http://www.ietf.org/rfc/rfc3986.txt>).

The Generic Connection Framework includes the following base classes:

```
javax.microedition.io.Connector
javax.microedition.io.Connection(interface)
javax.microedition.io.InputConnection(interface)
javax.microedition.io.OutputConnection(interface)
javax.microedition.io.StreamConnection(interface)
javax.microedition.io.ContentConnection(interface)
javax.microedition.io.StreamConnectionNotifier(interface)
javax.microedition.io.DatagramConnection(interface)
```

Additional Generic Connection Framework classes provide network protocol specific extensions to the base classes:

```
javax.microedition.io.Datagram(interface)
javax.microedition.io.UDPDatagramConnection(interface)
javax.microedition.io.HttpConnection(interface)
javax.microedition.io.HttpsConnection(interface)
```

```
javax.microedition.io.SocketConnection(interface)
javax.microedition.io.SecureConnection(interface)
javax.microedition.io.SecurityInfo
javax.microedition.io.ServerSocketConnection(interface)
```

The general purpose Generic Connection Framework exception class is also included:

```
javax.microedition.io.ConnectionNotFoundException
```

### 6.3.1.1 Additional Remarks

In order to read and write data to and from Generic Connections, a number of input and output stream classes are needed. The stream classes supported by the Java Card platform are listed in [Section 6.2.4, “Input-Output Classes”](#) on page 6-4.

## 6.3.2 Public Key Infrastructure Classes

A minimal set of classes to support certificate objects is included for supporting public key infrastructure cryptography.

```
javax.microedition.pki.Certificate(interface)
javax.microedition.pki.CertificateException
```



# Glossary

---

<b>access control mechanism</b>	a mechanism that permits or denies the access to a particular resource by a particular entity. An access control mechanism enforces a security policy.
<b>active applet instance</b>	an applet instance that is selected on at least one of the logical channels.
<b>AID (application identifier)</b>	<p>defined by ISO 7816, a string used to uniquely identify card applet applications and certain types of files in card file systems. An AID consists of two distinct pieces: a 5-byte RID (resource identifier) and a 0 to 11-byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies.</p> <p>A unique AID is associated with each applet class in an applet application module. In addition, a unique AID is assigned to each applet instance during installation. This applet instance AID is used by an off-card client to select the applet instance for APDU communication sessions.</p> <p>Applet instance URIs are constructed from their applet instance AID using the "aid" registry-based namespace authority as follows:</p> <pre>//aid/&lt;RID&gt;/&lt;PIX&gt;</pre> <p>where &lt;RID&gt; (resource identifier) and &lt;PIX&gt; (proprietary identifier extension) are components of the AID.</p>
<b>APDU</b>	an acronym for Application Protocol Data Unit as defined in ISO 7816-4.
<b>APDU-based application environment</b>	consists of all the functionalities and system services available to applet applications, such as the services provided by the applet container.
<b>API</b>	an acronym for Application Programming Interface. The API defines calling conventions by which an application program accesses the operating system and other services.

<b>applet</b>	within the context of this document, a Java Card applet, which is the basic component of applet-based applications and which runs in the APDU application environment.
<b>applet application</b>	an application that consists of one or more applets.
<b>applet container</b>	contains applet-based applications and manages their lifecycles through the applet framework API. Also provides the communication services over which APDU commands and responses are sent.
<b>applet framework</b>	an API that enables applet applications to be built.
<b>applicable credential manager</b>	the credential manager instance, application-assigned or card manager-assigned, that applies for a particular mode of communication. See <a href="#">credential manager</a> .
<b>applicable security requirements</b>	the security requirements instance, application-assigned or card manager-assigned, that applies for a particular mode of communication. See <a href="#">security requirements</a> .
<b>application assembler</b>	takes the output of the application developer and ensures that it is a deployable unit. Thus, the input of the application assembler is the application classes and resources, and other supporting libraries and files for the application. The output of the application assembler is an application archive.
<b>application-defined event</b>	an event that an application may define in its own namespace and may be the only one allowed to fire.
<b>application-defined service</b>	a service that an application may define in its own namespace and may be the only one allowed to register.
<b>application descriptor</b>	see <a href="#">descriptor</a> .
<b>application developer</b>	The producer of an application. The output of an application developer is a set of application classes and resources, and supporting libraries and files for the application. The application developer is typically an application domain expert. The developer is required to be aware of the application environment and its consequences when programming, including concurrency considerations, and create the application accordingly.
<b>application firewall</b>	see <a href="#">firewall</a> .
<b>application framework class loader</b>	a direct child of the extension library class loader, in charge of loading application framework libraries shared among a restricted set of application groups.
<b>application group</b>	a set of one or more applications executing in a common group context.

<b>application-managed authentication</b>	authentication that is programmatically triggered by an application's code based on some business logic.
<b>application-managed connection endpoint</b>	a client or server connection endpoint managed directly by an application.
<b>application module class loader</b>	a direct child in the class loader delegation hierarchy of either a group library class loader or of the classic library class loader, depending on the type of application model, in charge of loading the application module classes.
<b>application protection domain</b>	the set of permissions effectively granted to an application, that results from the combination of permissions granted by the platform security policy and the permissions granted by the card management security policy.
<b>application security policy</b>	a role-based security policy defined for a specific application and for which all the logical user and client security roles have been mapped to actual user identities and client application identities or characteristics on the platform to which the application is deployed.
<b>application URI</b>	a URI uniquely identifying an application instance on the platform.
<b>atomic operation</b>	an operation that either completes in its entirety or no part of the operation completes at all.
<b>atomicity</b>	state in which a particular operation is atomic. Atomicity of data updates guarantee that data are not corrupted in case of power loss or card removal.
<b>authentication</b>	the process of establishing or confirming an application or a user as authentic using some sort of credentials
<b>authenticator</b>	an authentication service that can be invoked both by applications for application-managed authentication and by the web container for container-managed authentication.
<b>authorization</b>	the process of allowing access to those resources by entities (applications or users) that have been granted authority to use them.
<b>basic logical channel</b>	logical channel 0, the only channel that is active at card reset in the APDU application environment. This channel is permanent and can never be closed.
<b>bootstrap class loader</b>	the root of the class loader delegation hierarchy in charge of loading the Java Card RE system classes.
<b>bytecode</b>	machine-independent code generated by the compiler and executed by the Java virtual machine.
<b>canonicalization (URI)</b>	the combined process of resolving a URI against a base URI, then normalizing it.

<b>card holder</b>	the primary user of a smart card.
<b>card holder-facing client</b>	a client that may directly and safely interact with the card holder. A card holder-facing client may typically be local, co-hosted on the card-hosting device, or in close proximity to the card.
<b>card holder user</b>	a user whose identity may be assumed by the card holder.
<b>card manager</b>	the on-card application to download and install applications and libraries. The card manager receives executable binary and metadata from the off-card installer, writes the binary into the smart card memory, links it with the other classes on the card, and creates and initializes any data structures used internally by the Java Card Runtime Environment.
<b>card management facility</b>	the Java Card platform layer responsible for securely adding and removing application code and instances onto the platform.
<b>card management security policy</b>	a permission-based security policy that is defined by a card management authority and that grants some permissions to an application or group of applications in accordance with the operational environment in which the application or group of applications is deployed.
<b>card session</b>	a card session begins when it is powered up or reset. The card is then able to exchange messages with external clients. The card session ends when the card loses power or is reset.
<b>classic applet</b>	applets with the same capabilities as those in previous versions of the Java Card platform and in the Classic Edition.
<b>classic applet container mutex object</b>	the object that is used by the Java Card RE to synchronize all concurrent accesses to a classic applet's code in order to guarantee its thread safety.
<b>Classic Edition</b>	one of the two editions in the Java Card 3 Platform. The Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and is backward compatible with it, targeting resource-constrained devices that solely support applet-based applications.
<b>classic library</b>	a Java programming language package that does not contain any non-abstract classes that extend the class <code>javacard.framework.Applet</code> . A classic applet application comprises a Java programming language package that contains one or more non-abstract classes that extend the <code>javacard.framework.Applet</code> class.
<b>classic library class loader</b>	a direct child of the shareable interface class loader in charge of loading classic library classes.
<b>classic SIO proxy</b>	see <i>classic SIO synchronization proxy</i> .

<b>classic SIO synchronization proxy</b>	an object that implements a shareable interface of a classic applet application and that synchronizes with all other concurrent accesses to the classic applet application before delegating to the actual SIO. An SIO synchronization proxy is returned to each client of the classic applet application that requests access to that shareable interface.
<b>class loader</b>	a Java Card RE component that defines and enforces a different class namespace for the classes it loads.
<b>class loader delegation hierarchy</b>	the hierarchy of class loaders that enforces code isolation among applications while allowing for sharing of system and library code.
<b>client application</b>	an on-card application that uses services provided by other applications (server applications).
<b>client-role-based security</b>	see <i>role-based security</i> .
<b>Connected Edition</b>	one of the two editions in the Java Card 3 Platform. The Connected Edition has a significantly enhanced runtime environment and a new virtual machine. It includes new network-oriented features, such as support for web applications, including the Java™ Servlet APIs, and also support for applets with extended and advanced capabilities. An application written for or an implementation of the Connected Edition may use features found in the Classic Edition.
<b>connection endpoint (client, server)</b>	see <i>application-managed connection endpoint</i> , <i>container-managed connection endpoint</i> .
<b>container-managed authentication</b>	authentication that is automatically triggered by the web container when a request to a protected resource is received, based on the declarative security configuration of the application.
<b>container-managed connection endpoint</b>	a server connection endpoint managed by a container, such as an HTTP or HTTPS server connection endpoint managed by the servlet container.
<b>container-managed object</b>	an object of which the lifecycle (creation, invocation, deletion...) is managed by a container. Examples are instances of <code>Applet</code> , <code>Servlet</code> and <code>Filter</code> .
<b>context path</b>	the path within the web server a servlet context is rooted at. The context path of a web application corresponds to its application URI.

<b>context switch</b>	a change from one currently active context to another. For example, a context switch is caused by an attempt to access an object that belongs to an application instance that resides in a different application group. The result of a context switch is a new currently active context.
<b>converter</b>	a piece of software that preprocesses all of the Java programming language class files of a classic applet application that make up a package, and converts the package into a standalone classic applet application module distribution format (CAP file). The Converter also produces an export file.
<b>credential</b>	material that can be used to ascertain the identity of a party (authenticate) in order to control access by that party to information or other resources and or to protect the integrity or confidentiality of information exchanges with that party. Examples of credentials are password, PIN or public-key certificates.
<b>credential manager</b>	an object that manages the key and trust material of an application when a secure communication is being established by either that application or by the web container on behalf of that web application.
<b>currently active context</b>	when an object instance method is invoked, an owning context of the object becomes the currently active context for that particular thread of execution.
<b>currently active namespace</b>	corresponds to the application owner identifier of the active context set upon entry into the group context for a particular thread of execution.
<b>currently selected applet</b>	the applet container keeps track of the currently selected Java Card applet. Upon receiving a SELECT FILE command with this applet's AID, the applet container makes this applet the currently selected applet. The applet container sends all APDU commands to the currently selected applet.
<b>declarative security</b>	a means of expressing an application's security structure, including roles, access control, and authentication requirements in a form external to the application, such as in the deployment descriptor of a web application.
<b>default applet</b>	an applet that is selected by default on a logical channel in the APDU application environment when it is opened. If an applet is designated the default applet on a particular logical channel in the APDU application environment on the Java Card platform, it becomes the active applet by default when that logical channel is opened using the basic channel.
<b>default servlet</b>	the application-defined servlet that is used to serve a request when no other servlet applies.
<b>default default servlet</b>	a servlet implementing the default container behavior that serves static resources of web applications. This servlet is used to serve a request to static content when no other servlet applies and no default servlet is defined by the application.

<b>deployer</b>	<p>The deployer takes one or more application archive files provided by an application developer and deploys the application into a card in a specific operational environment. The operational environment includes other installed applications and libraries, as well as standard bodies-defined frameworks. The deployer must resolve all the external dependencies declared by the developer.</p> <p>The deployer is an expert in a specific operational environment. For example, the deployer is responsible for mapping the security roles defined by the application developer to the users that exist in the operational environment where the application is deployed.</p>
<b>deployment unit</b>	entity that can be distributed, deployed and installed on the Java Card platform.
<b>deployment descriptor</b>	see <a href="#">descriptor</a> .
<b>descriptor</b>	a document that describes the configuration and deployment information of an application. A deployment descriptor conveys the elements and configuration information of an application between application developers, application assemblers, and deployers. A runtime descriptor describes the configuration and deployment information of an application that are specific to an operating environment to which the application is to be deployed.
<b>distribution format</b>	structure and encoding of a distribution or deployment unit intended for public distribution.
<b>distribution unit</b>	see <a href="#">deployment unit</a> .
<b>EEPROM</b>	an acronym for Electrically Erasable, Programmable Read Only Memory.
<b>entry point method</b>	well-defined method of an object owned by an application (respectively the Java Card RE) that can be “legally” invoked by another application or the Java Card RE (respectively an application). SIO methods and other container-managed objects’ lifecycle methods are application entry point methods. Java Card RE entry point objects’ methods are Java Card RE entry point methods.
<b>event</b>	an object that encapsulates some occurring condition or situation. In the context of the event notification facility, an event is a shareable interface object that an application (event-producing application) uses to notify its clients (event-consuming applications) of an occurring condition.
<b>event consuming application</b>	an application that registers for notification of events fired by an event producing application.
<b>event listener</b>	an object that is registered to handle events when they occur. In the context of the event notification facility, an event listener is an object that a client application (event-consuming application) registers and uses to handle SIO-based events an application (event-producing application) produces.

<b>event notification facility</b>	a Java Card RE facility (or subsystem) that is used for event-driven inter-application communications.
<b>event notification listener</b>	see <i>event listener</i> .
<b>event producing application</b>	an application that fires events.
<b>event registry</b>	the core component of the event notification facility. The event registry is used for registering for notification of events and for notifying of events.
<b>event URI</b>	a URI that uniquely identifies an event produced by an event-producing application.
<b>export file</b>	a file produced by the Converter tool used during classic applet application development that represents the fields and methods of a package that can be imported by classes in other classic applet applications and classic libraries.
<b>extended applet</b>	an applet with extended and advanced capabilities (compared to a classic applet) such as the capabilities to manipulate <code>String</code> objects and open network connections.
<b>extension library</b>	library that extends the functionality of the platform.
<b>extension library class loader</b>	a direct child of the shareable interface class loader in the class loader delegation hierarchy in charge of loading extension libraries.
<b>externally visible</b>	<p>in the Java Card platform, any classes, interfaces, their constructors, methods, and fields of an application that can be accessed from another application according to the Java programming language semantics, as defined by the <i>Java Language Specification</i>, and code isolation restrictions (see the “Code Isolation” section in <i>Java Card Runtime Environment Specification, Java Card Platform, v3.0.1, Connected Edition</i>).</p> <p>Externally visible items of a classic applet application are represented in an export file. For a classic library package, all classes, interfaces, their constructors, methods, and fields of an application that can be accessed from another application according to the Java programming language access control semantics, as defined by the <i>Java Language Specification</i> are listed in the export file.</p>
<b>file permissions mode</b>	an attribute of a file system object that indicates whether read or write operation on the object are permitted or denied.
<b>filter</b>	a web application component that is used to transform the content or header information of HTTP requests or responses.



<b>finalization</b>	<p>the process by which a Java virtual machine (VM) allows an unreferenced object instance to release non-memory resources (for example, close and open files) prior to reclaiming the object's memory. Finalization is only performed on an object when that object is ready to be garbage collected (meaning, there are no references to the object).</p> <p>Finalization is not supported by the Java Card virtual machine. The method <code>finalize()</code> is not called automatically by the Java Card virtual machine.</p>
<b>firewall</b>	the mechanism that prevents unauthorized accesses to objects in one application group context from another application group context.
<b>flash memory</b>	a type of persistent mutable memory. It is more efficient in space and power than EPROM. Flash memory can be read bit by bit but can be updated only as a block. Thus, flash memory is typically used for storing additional programs or large chunks of data that are updated as a whole.
<b>garbage collection</b>	the process by which dynamically allocated storage is automatically reclaimed during the execution of a program.
<b>global array</b>	an applet environment array objects accessible from any context.
<b>global authentication</b>	the scope of a user authentication that can be tracked globally (card-wide). Global authentication is restricted to card-holder-users. Authorization to access resources protected by a globally authenticated card-holder-user identity is granted to all users.
<b>group context</b>	protected object space associated with each application group and Java Card RE. All objects owned by an application belong to the context of the application group.
<b>group-library class loader</b>	a direct child of the extension library class loader in charge of loading the libraries private to an application group. Libraries, private to different application groups, are loaded by distinct group library class loaders, one per web or extended applet application group.
<b>heap</b>	<p>a common pool of free memory in volatile and persistent spaces usable by a program. A part of the computer's memory used for dynamic memory allocation, in which blocks of memory are used in an arbitrary order.</p> <p>The Java Card virtual machine's volatile heap is typically garbage collected on demand and on card tear.</p> <p>The Java Card virtual machine's persistent heap is typically garbage collected on a less frequent basis. Memory associated with objects allocated from the persistent heap are not necessarily reclaimed.</p>
<b>instance variables</b>	also known as non-static fields.

<b>instantiation</b>	in object-oriented programming, to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function.
<b>instruction</b>	a statement that indicates an operation for the computer to perform and any data to be used in performing the operation. An instruction can be in machine language or a programming language.
<b>internally visible</b>	items that are not externally visible to other applications on the card. See also <i>externally visible</i> .
<b>inter-application communication facility</b>	see <i>service facility</i> , <i>event notification facility</i> .
<b>JAR file</b>	an acronym for Java Archive file, which is a file format used for aggregating and compressing many files into one.
<b>Java Card Platform Remote Method Invocation</b>	a subset of the Java Platform Remote Method Invocation (RMI) system optionally supported by the APDU application environment. It provides a mechanism for a client application to invoke a method on a remote object of an applet application on the card.
<b>Java Card Runtime Environment (Java Card RE)</b>	consists of the Java Card virtual machine and the associated native methods.
<b>Java Card Virtual Machine (Java Card VM)</b>	a subset of the Java virtual machine, which is designed to be run on smart cards and other resource-constrained devices. The Java Card VM acts an engine that loads Java class files and executes them with a particular set of semantics.
<b>Java Card RE context</b>	the context of the Java Card RE has special system privileges so that it can perform operations that are denied to contexts of applications.

<b>Java Card RE entry point object</b>	<p>an object owned by the Java Card RE context that contains entry point methods. These methods can be invoked from any application group context and allows applications to request Java Card RE system services. A Java Card RE entry point object can be either temporary or permanent:</p> <p><b>temporary</b> - references to temporary Java Card RE entry point objects cannot be stored in class variables, instance variables or array components. The Java Card RE detects and restricts attempts to store references to these objects as part of the firewall functionality to prevent unauthorized reuse. Examples of these objects are APDU objects and the APDU byte array.</p> <p><b>permanent</b> - references to permanent Java Card RE entry point objects can be stored and freely reused. Examples of these objects are Java Card RE-owned AID instances.</p>
<b>JDK™ software</b>	an acronym for Java Development Kit. The JDK software is a Sun Microsystems, Inc. product that provides the environment required for software development in the Java programming language. The JDK software is available for a variety of operating systems, for example Sun Microsystems Solaris™ OS and Microsoft Windows.
<b>local variable</b>	a data item known within a block, but inaccessible to code outside the block. For example, any variable defined within a method is a local variable and cannot be used outside the method.
<b>locally accessible web application</b>	an application that may interact with the card holder.
<b>logical channel</b>	as seen at the card edge, works as a logical link to an applet application on the card. A logical channel establishes a communications session between a card applet and the terminal. Commands issued on a specific logical channel are forwarded to the active applet on that logical channel. For more information, see the <i>ISO/IEC 7816 Specification, Part 4</i> . ( <a href="http://www.iso.org">http://www.iso.org</a> ).
<b>MAC</b>	an acronym for Message Authentication Code. MAC is an encryption of data for security purposes.
<b>mask production (masking)</b>	refers to embedding the Java Card virtual machine, runtime environment, and applications in the read-only memory of a smart card during manufacture.
<b>method</b>	a procedure or routine associated with one or more classes in object-oriented languages.
<b>mode (communication)</b>	designates the type or protocol of communication (HTTPS, SSL/TLS, SIO...) and the mode of operation (client or server) that characterizes a communication endpoint.

<b>module (application)</b>	the logical unit of assembly of web or applet-based application. The components of a web application are assembled into a web application module. The components of an applet application are assembled into a applet application module.
<b>multiselectable applets</b>	implements the <code>javacard.framework.MultiSelectable</code> interface. Multiselectable applets can be selected on multiple logical channels in the APDU application environment at the same time. They can also accept other applets belonging to the same applet application being selected simultaneously.
<b>multiselectd applet</b>	an applet instance that is selected and, therefore, active on more than one logical channel in the APDU application environment simultaneously.
<b>named permission</b>	a permission that has a name but no actions list; the named permission is either granted or not. A named permission typically protects a function or functionality.
<b>namespace</b>	a set of names in which all names are unique.
<b>native method</b>	a method that is not implemented in the Java programming language, but in another language. The Card Manger does not load applications containing native methods.
<b>nibble</b>	four bits.
<b>non-card holder-facing client</b>	a client that does not directly interact with the card holder, but interacts with some other-users such as remote administrators. A non-card holder-facing client may typically be a remote system that may interact with the card through the network to which the card-hosting device itself is connected.
<b>non-volatile memory</b>	memory that is expected to retain its contents between card tear and power up events or across a reset event on the smart card device.
<b>normalization (classic applet)</b>	the process of transforming and repackaging a Java application packaged for the Java Card Platform, Version 2.2.2, for deployment on both the Java Card 3 Platform, Connected Edition and the Java Card 3 Platform, Classic Edition.
<b>normalization (URI)</b>	the process of removing unnecessary "." and ".." segments from the path component of a hierarchical URI.
<b>Normalizer</b>	a software tool that allows Java applications programmed for the Java Card Platform, Version 2.2.2, to be deployed on both the Java Card 3 Platform, Connected Edition and on the Java Card 3 Platform, Classic Edition. It also allows Java applications packaged for Version 2.2.2 to be transformed through the normalization process and then repackaged for deployment on both the Connected and Classic Editions.

<b>object-oriented</b>	a programming methodology based on the concept of an <i>object</i> , which is a data structure encapsulated with a set of routines, called <i>methods</i> , which operate on the data.
<b>object owner</b>	the applet instance context or web application context or the Java Card RE context which was the currently active context when the object was instantiated.
<b>object</b>	in object-oriented programming, unique instance of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.
<b>off-card client</b>	see <a href="#">off-card client application</a> .
<b>off-card client application</b>	an application that is not resident on the card, but runs at the request of a user's actions.
<b>off-card installer</b>	the off-card application that transmits the application and library executables to the card manager application running on the card.
<b>off-card proxy generator</b>	a program or tool used to generate classic SIO synchronization proxies prior to packaging and deploying a classic applet application.
<b>on-card client</b>	see <a href="#">client application</a> .
<b>origin logical channel</b>	the logical channel in the APDU application environment on which an APDU command is issued.
<b>"other user"</b>	a user other than a card holder user, such as a remote card administrator.
<b>owning context</b>	the application or Java Card RE context in which an object is instantiated or created.
<b>owner context</b>	see <a href="#">owning context</a> .
<b>package</b>	a namespace within the Java programming language that can have classes and interfaces.
<b>permission</b>	an object that represents access to specific protected resources, such as security-sensitive system resources, or application resources, such as services provided by applications. Permissions are instances of subclasses of the <code>Permission</code> class. A permission has a name and may have an actions list.
<b>permission actions list</b>	an attribute of a permission used to designate those actions for which the resources designated by the target name are protected.
<b>permission-based security</b>	measures defined by a permission-based security policy that restrict access to protected system and library resources.

<b>permission-based security policy</b>	a security policy that maps some of the characteristics of an application requesting access to a protected resource to a set of permissions granted to the application.
<b>permission name</b>	an attribute of a permission used to designate a protected function or resource, or a set thereof.
<b>permission target name</b>	the name attribute of a permission object (permission name) that designates the resource or set of resources that are protected with that permission.
<b>permission type</b>	a type defined by a permission class.
<b>persistent object</b>	persistent objects and their values persist from one card session to the next, indefinitely. Persistent object values are typically updated atomically using transactions. The term persistent does not mean there is an object-oriented database on the card or that objects are serialized and deserialized, just that the objects are not lost when the card loses power.
<b>PIX</b>	see <a href="#">AID (application identifier)</a> .
<b>platform event</b>	a well-defined event fired by the platform. Examples are clock resynchronization events.
<b>platform protection domain</b>	a set of permissions granted to an application or group of applications by the platform security policy. A platform protection domain is defined by two sets of permissions: a set of included permissions that are granted and a set of excluded permissions that are denied and can never be granted.
<b>platform security policy</b>	the permission-based security policy that maps application models to sets of permissions granted to applications implementing these application models. For each of the application models, the platform security policy guarantees the consistency and integrity of the applications implementing the application model.
<b>principal</b>	an entity that can be authenticated by an authentication protocol. A principal is identified by a <i>principal name</i> and authenticated by using <i>authentication data</i> . The content and format of the principal name and the authentication data depend on the authentication protocol.
<b>programmatic security</b>	a means for a security aware application to express the security model of the application when declarative security alone is not sufficient.
<b>protected content</b>	see <a href="#">protected resource</a> .
<b>protected resource</b>	an application or system resource that is protected by an access control mechanism.

<b>protection domain</b>	a set of permissions granted to an application or group of applications.
<b>RAM (random access memory)</b>	temporary working space for storing and modifying data. RAM is non-persistent memory; that is, the information content is not preserved when power is removed from the memory cell. RAM can be accessed an unlimited number of times and none of the restrictions of EEPROM apply.
<b>reachability disrupting object</b>	a special object that prevents the promotion of a volatile object to become a persistent object. If a volatile object is referenced by a persistent object, which is not a reachability disrupting object, or by a root of persistence, the volatile object is automatically promoted and becomes a persistent object. An example of reachability disrupting object is a <code>TransientReference</code> object.
<b>reference implementation</b>	a fully functional and compatible implementation of a given technology. It enables developers to build prototypes of applications based on the technology.
<b>remote interface</b>	<p>an interface of an applet application, which extends, directly or indirectly, the interface <code>java.rmi.Remote</code>.</p> <p>Each method declaration in the remote interface or its super-interfaces includes the exception <code>java.rmi.RemoteException</code> (or one of its superclasses) in its throws clause.</p> <p>In a remote method declaration, if a remote object is declared as a return type, it is declared as the remote interface, not the implementation class of that interface.</p> <p>In addition, Java Card RMI imposes additional constraints on the definition of remote methods of an applet application. See <i>Runtime Environment Specification, Java Card Platform, Version 3.0.1, Classic Edition</i>.</p>
<b>remote methods</b>	the methods of a remote interface of an applet application.
<b>remote object</b>	an object of an applet application whose remote methods can be invoked remotely from the off-card client. A remote object is described by one or more remote interfaces of an applet application.
<b>remote user</b>	an user whose identity may be assumed by a remote entity, such as a remote card administrator.
<b>remotely accessible web application</b>	an application that is not expected to interact with the card holder but with other-users, potentially remote.
<b>resolution (URI)</b>	the process of resolving one URI against another, base URI. The resulting URI is constructed from components of both URIs in the manner specified by RFC 3986, taking components from the base URI for those not specified in the original.

<b>resource URI</b>	a URI that uniquely identifies a resource on the platform. Examples are service URI, event URI, application URI and file URI.
<b>RFU</b>	acronym for Reserved for Future Use.
<b>RID</b>	see <a href="#"><i>AID (application identifier)</i></a> .
<b>RMI</b>	an acronym for Remote Method Invocation. RMI is a mechanism for invoking instance methods on objects located on remote virtual machines (meaning, a virtual machine other than that of the invoker).
<b>role (development)</b>	the actions and responsibilities taken by various parties during the development, deployment, and running of an application. In some scenarios, a single party may perform several roles. In others, each role may be performed by a different party.
<b>role (security)</b>	an abstract notion used by an application developer in an application that can be mapped by the deployer to a user, or group of users, in a security policy domain.
<b>role-based security</b>	measures defined by a role-based security policy that restrict access by clients or by users to protected application resources.
<b>role-based security policy</b>	a security policy that maps some of the characteristics of an application requesting access to protected resources, such as its identity and the identity of the user on behalf of whom the access is requested to roles permitted to access the protected resources.
<b>ROM (read-only memory)</b>	memory used for storing the fixed program of the card. A smart card's ROM contains operating system routines as well as permanent data and user applications. No power is needed to hold data in this kind of memory. ROM cannot be written to after the card is manufactured. Writing a binary image to the ROM is called masking and occurs during the chip manufacturing process.
<b>root URI</b>	a URI that identifies the root of an application's namespace for a particular scheme. Examples are an application's service root URI, an application's event root URI.
<b>runtime descriptor</b>	see <a href="#"><i>descriptor</i></a> .
<b>runtime environment</b>	see <a href="#"><i>Java Card Runtime Environment (Java Card RE)</i></a> .
<b>secure port redirector</b>	a web application container that redirects HTTP requests for protected content sent over unsecure connections to the secure port over which that content can be served. Protected content must be served only over a secure port.
<b>security constraint</b>	a declarative way of defining the protection of web content. A security constraint associates authorization and or user data constraints with HTTP operations on web resources.



<b>security policy domain</b>	the scope over which security policies are defined and enforced by a security administrator of the security service. A security policy domain is also sometimes referred to as a <i>realm</i> .
<b>security policy</b>	designates the protected resources that can be accessed by individual applications or groups of applications. These protected resources may be security-sensitive system resources or application resources such as services provided by other applications.
<b>security requirements</b>	the required security characteristics for a particular secure communication being established by either an application or by the web container on behalf of a web application.
<b>server application</b>	an on-card application that provides a service to its clients.
<b>service</b>	a shareable interface object that a server application uses to provide a set of well-defined functionalities to its clients.
<b>service facility</b>	a Java Card RE facility (or subsystem) that is used for inter-application communications.
<b>service factory</b>	an object that the Java Card RE invokes to create a service - on behalf of the server application that registered that service - for a client application that looked up the service.
<b>service registry</b>	the core component of the service facility. The service facility is used for registering and looking up services.
<b>service URI</b>	a URI that uniquely identifies a service provided by a server application.
<b>servlet</b>	a web application component, managed by a container, that generates dynamic web content and that runs in the web application environment.
<b>servlet container</b>	see <a href="#">web application container</a> .
<b>servlet context</b>	a container-managed object that defines a servlet's view of the web application within which the servlet is running. A servlet context is rooted at a known path within a web server: a context path.
<b>servlet definition</b>	a unique name associated with a fully qualified class name of a class implementing the <code>servlet</code> interface. A set of initialization parameters can be associated with a servlet definition. See <i>Java Servlet Specification, Connected Edition</i> .
<b>servlet mapping</b>	a servlet definition that is associated by a servlet container with a URL path pattern. All requests to that path pattern are handled by the servlet associated with the servlet definition. See <i>Java Servlet Specification, Connected Edition</i> .

<b>session-scoped authentication</b>	the scope of a user authentication that is tracked on a per-session basis. This prevents a user authenticated in a conversational session under one identity to gain unauthorized access to protected resources authorized to another, simultaneously authenticated, identity.
<b>shareable interface</b>	an interface that defines a set of shared methods. These interface methods can be invoked from an application in one group context when the object implementing them is owned by an application in another group context.
<b>shareable interface class loader</b>	the direct child of the bootstrap class loader in the class loader delegation hierarchy in charge of loading publicly exposed shareable interfaces.
<b>shareable interface object (SIO)</b>	an object that implements the shareable interface.
<b>shareable interface object-based service</b>	see <i>service</i> .
<b>smart card</b>	a card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Unlike magnetic stripe cards, smart cards carry both processing power and information. They do not require access to remote databases at the time of a transaction.
<b>SPI</b>	an acronym for Service Provider Interface or sometimes for System Programming Interface. The SPI defines calling conventions by which a platform implementer may implement system services.
<b>standard event</b>	a standard event with a well-defined semantic that an application may fire. Examples are standard application lifecycle events such as application creation and deletion events, and standard resource lifecycle events such as resource creation and deletion events.
<b>standard service</b>	a standard service with a well-defined interface that an application may provide and register. Examples are authenticators - authentication services.
<b>restartable task</b>	an object implementing the <code>Runnable</code> interface that has been registered for recurrent execution over card sessions. A task executes in its own thread.
<b>restartable task registry</b>	a Java Card RE facility that is used for registering tasks for recurrent execution over card sessions.
<b>terminal</b>	is typically a computer in its own right with an interface which connects with a smart card to exchange and process data.
<b>thread</b>	the basic unit of program execution. A process can have several threads running concurrently each performing a different job, such as waiting for events or performing a time consuming job that the program doesn't need to complete before going on. When a thread has finished its job, it is suspended or destroyed.

<b>thread's active context</b>	when an object instance method is invoked, the owning context of the object becomes the currently active context for that particular thread of execution. Synonymous with <i>currently active context</i> .
<b>transaction</b>	an atomic operation in which the developer defines the extent of the operation by indicating in the program code the beginning and end of the transaction.
<b>transaction facility</b>	a Java Card RE facility that enables an application to complete a single logical operation on application data atomically, consistently and durably within a transaction.
<b>transient object</b>	the state of transient objects do not persist from one card session to the next, and are reset to a default state at specified intervals. Updates to the values of transient objects are not atomic and are not affected by transactions.
<b>transferable classes</b>	<p>classes whose instances can have their ownership transferred to a context different from their currently owning context. Transferable classes are of two types:</p> <p><b>Implicitly transferable classes</b> - Classes whose instances are not bound to any context (group contexts or Java Card RE context) and can, therefore, be passed and shared between contexts without any firewall restrictions. Examples are Boolean and literal <code>String</code> objects.</p> <p><b>Explicitly transferable classes</b> - Classes whose instances must have their ownership explicitly transferred to another application's group context in order to be accessible to that other application. Examples are arrays and newly created <code>String</code> objects.</p>
<b>transfer of ownership</b>	a Java Card RE facility that allows for an application to transfer the ownership of objects it owns to an other application. Only instances of transferable classes can have their ownership transferred.
<b>trusted client</b>	an on-card or off-card application client that an on-card application trusts on the basis of credentials presented by the client.
<b>trusted client credentials</b>	credentials that an on-card application uses to ascertain the identity of clients it trusts.
<b>uniform resource identifier (URI)</b>	a compact string of characters used to identify or name an abstract or physical resource. A URI can be further classified as a uniform resource locator (URL), a uniform resource name (URN), or both. See RFC 3986 for more information.
<b>uniform resource locator (URL)</b>	a compact string representation used to locate resources available via network protocols or other protocols. Once the resource represented by a URL has been accessed, various operations may be performed on that resource. See RFC 1738 for more information. A URL is a type of uniform resource identifier (URI).

<b>user role-based security</b>	see <i>role-based security</i> .
<b>verification</b>	a process performed on an application or library executable that ensures that the binary representation of the application or library is structurally correct.
<b>volatile memory</b>	memory that is not expected to retain its contents between card tear and power up events or across a reset event on the smart card device.
<b>volatile object</b>	an object that is ideally suited to be stored in volatile memory. This type of object is intended for a short-lived object or an object which requires frequent updates. A volatile object is garbage collected on card tear (or reset).
<b>web application</b>	<p>a collection of servlets, HTML documents, and other web resources that might include image files, compressed archives, and other data. A web application is packaged into a web application archive.</p> <p>All compatible servlet containers must accept a web application and perform a deployment of its contents into their runtime. This may mean that a container can run the application directly from a web application archive file or it may mean that it will move the contents of a web application into the appropriate locations for that particular container. See <i>Java Servlet Specification, Connected Edition</i>.</p>
<b>web application archive</b>	<p>the physical representation of a web application module. A single file that contains all of the components of a web application. This archive file is created by using standard JAR file tools, which allow any or all of the web components to be signed.</p> <p>A web application archive file is identified by the <code>.war</code> extension and is often referred to as a WAR file. A new extension is used instead of <code>.jar</code> because that extension is reserved for files which contain a set of class files and that can be placed in the classpath. As the contents of a web application archive are not suitable for such use, a new extension was required. See <i>Java Servlet Specification, Connected Edition</i>.</p>
<b>web application container</b>	contains and manages web applications and their components (for example, servlets) through their lifecycle. Also provides the network services over which HTTP requests and responses are sent and manages security of web applications.
<b>web application environment</b>	in addition to the Java Card RE, consists of all the functionalities and system services available to web applications, such as the services provided by the web application container.
<b>web client</b>	an off-card entity that requests services from an on-card web application. A typical example is a web browser.

**XML schema** description of a type of XML document as a set of rules to which an XML document must conform in order to be considered valid according to that schema.



# Index

---

## A

- access control mechanism, Glossary-1
- active applet instance, Glossary-1
- AID (application identifier), Glossary-1
- AnnotationDefault attribute, 5-6
- APDU, Glossary-1
- APDU-based application environment, Glossary-1
- API, Glossary-1
- applet, Glossary-2
- applet application, Glossary-2
- applet container, Glossary-2
- applet framework, Glossary-2
- applicable credential manager, Glossary-2
- applicable security requirements, Glossary-2
- application assembler, Glossary-2
- application descriptor, Glossary-2
- application developer, Glossary-2
- application development platform, 2-2
- application firewall, Glossary-2
- application framework class loader, Glossary-2
- application group, Glossary-2
- application module class loader, Glossary-3
- application protection domain, Glossary-3
- application security policy, Glossary-3
- application URI, Glossary-3
- application-defined event, Glossary-2
- application-defined service, Glossary-2
- application-level security, 3-3
- application-managed authentication, Glossary-3
- applications

- compact representations, use of, 5-8
- concurrent execution, 3-2
- dynamic class loading restrictions, 3-5
- managing, 3-2
- persistent storage, 3-2
- public distribution, 5-6
- public representation, 5-6
- application-specific resource files, 5-7
- asynchronous exceptions, unsupported, 4-2, 5-4
- atomic operation, Glossary-3
- atomicity, Glossary-3
- authentication, Glossary-3
- authenticator, Glossary-3
- authorization, Glossary-3

## B

- basic logical channel, Glossary-3
- bootstrap class loader, Glossary-3
- bytecode, Glossary-3
- bytecodes not supported, list of, 5-1

## C

- calendar classes, list of, 6-5
- canonicalization (URI), Glossary-3
- card holder, Glossary-4
- card holder user, Glossary-4
- card holder-facing client, Glossary-4
- card management facility, Glossary-4
- card management security policy, Glossary-4
- Card Manager, 3-2
- card manager, Glossary-4

- card session, Glossary-4
- case conversion, 6-8
- certificate objects, classes to support, 6-12
- character
  - case conversion, 6-8
  - case conversion support, 6-8
  - encodings, 6-8
  - internationalization support, 6-8
  - sets, 6-8
  - tables, 6-8
  - Unicode character support, 6-8
- character encoding, default, 6-9
- class file verification, implementation of, 5-5
- class file verifier, 3-4
- class files
  - attributes, list of, 5-6
  - defining class files lookup order, 5-7
  - lookup order, 5-7
  - lookup order restrictions, 5-7
  - lookup strategy, 5-7
  - runtime verification, 5-5
  - verification, purpose of, 3-4
- class instances, finalization not supported, 4-2, 5-4
- class loader, Glossary-5
- class loader delegation hierarchy, Glossary-5
- class loaders
  - platform-defined, 5-3
  - user-defined, 5-3
- `Class.getResourceAsStream` method, 5-7
- classes
  - calendar, list of, 6-5
  - collection, list of, 6-4
  - `com.sun.javacard.i18n.uclc.DefaultCaseConverter`, 6-8
  - data types, list of, 6-3
  - error, list of, 6-8
  - exception, list of, 6-6
  - Generic Connection framework, 6-11
  - input, list of, 6-4
  - `InputStreamReader`, 6-8
  - internationalization, list of, 6-6, 6-9
  - Java ME platform, derived from, 6-10
  - Java SE platform, derived from, 6-2
  - `java.util.Properties`, 6-9
  - localization, list of, 6-6
  - manufacturer-specific, 6-10
  - output, list of, 6-4
  - `OutputStreamWriter`, 6-8
  - prelinking, 5-8
  - preloading, 5-8
  - public key infrastructure, 6-12
  - ROMizing, 5-8
  - security, list of, 6-6
  - system, list of, 6-2
  - time, list of, 6-5
  - utility, list of, 6-5
- classic applet, Glossary-4
- classic applet container mutex object, Glossary-4
- Classic Edition, Glossary-4
- classic library, Glossary-4
- classic library class loader, Glossary-4
- classic SIO proxy, Glossary-4
- client-role-based security, Glossary-5
- closed network environment, 5-6
- collection classes, list of, 6-4
- `com.sun.javacard.i18n.uclc.DefaultCaseConverter` class, 6-8
- compressed JAR files
  - representation format, 5-7
  - support for, 5-5
- Connected Edition, Glossary-5
- connection endpoint, Glossary-5
- connection endpoint (client, server), Glossary-5
- `Connector.open` parameter, 6-11
- constraints
  - on thread operations, 5-3
  - on user-supplied classes, 5-2
  - on user-supplied methods, 5-2
- container-managed authentication, Glossary-5
- container-managed object, Glossary-5
- content
  - concurrent execution, 3-2
  - dynamic downloading, 2-2
  - managing, 3-2
  - persistent storage, 3-2
- context path, Glossary-5
- context switch, Glossary-6
- converter, Glossary-6
- credential, Glossary-6
- credential manager, Glossary-6
- currently active context, Glossary-6
- currently active namespace, Glossary-6



currently selected applet, Glossary-6

## D

daemon threads, 5-3

data type classes, list of, 6-3

data types not permitted, 5-2

declarative security, Glossary-6

default applet, Glossary-6

default character encoding, 6-9

default default servlet, Glossary-6

default servlet, Glossary-6

deployer, Glossary-7

deployment descriptor, Glossary-7

Deprecated attribute, 5-6

devices

- access limitations based on security policies, 3-3

- basic security coverage, 3-3

- general characteristics of, 2-1

- protected transactions, 3-3

- using compressed JAR file representation  
format, 5-7

distribution format, Glossary-7

distribution unit, Glossary-7

double data type, nonuse of, 5-2

dynamic class loading, protection of, 3-5

## E

EEPROM, Glossary-7

eliminated features, list of, 5-3

EnclosingMethod attribute, 5-6

end-to-end security, 3-3

entry point method, Glossary-7

Error classes

- limited set of, 4-2

- required, 4-3

- virtual machine error handling, 5-4

error classes

- list of, 6-8

- virtual machine support for, 4-3

error handling

- limitations, 4-2, 5-4

- virtual machine behavior, 4-3

event consuming application, Glossary-7

event notification listener, Glossary-8

event producing application, Glossary-8

event registry, Glossary-8

event URI, Glossary-8

exception classes, list of, 6-6

exception limitations, 5-4

exceptions, limitations, 4-2

export file, Glossary-8

extended applet, Glossary-8

extension library, Glossary-8

## F

file formats, supported, 5-5

float data type, nonuse of, 5-2

floating point support removal, 1-2, 4-1, 5-1

## G

Generic Connection Framework

- base classes, list of, 6-11

- classes, 6-11

- exception class, 6-12

- network protocol extensions, 6-11

## H

hardware

- host operating system, 2-3, 2-4

- management of, 2-3

hardware requirements, 2-2

host operating system

- managing hardware, 2-3, 2-4

- non-supported items, 2-3, 2-4

## I

implementation-level optimizations, 5-8

InnerClasses attribute, 5-6

input classes, list of, 6-4

InputStreamReader class, 6-8

internationalization classes, list of, 6-6, 6-9

internationalization, support for, 6-10

interval timer, support for, 2-3

invalid class files, rejecting, 5-4

## J

Java Archive (JAR) file format, compression, 5-5

Java Card platform

- application-level security, 3-3

- class library subsets, 6-2

- end-to-end security, 3-3
- libraries, 6-1
- limited error handling, causes for, 4-2
- lowest common denominator standard, 6-1
- low-level security, 3-3
- minimal set of libraries, 6-1
- property support, 6-10
- requirements, 2-4
- sandbox security model, 3-4
- security models, 3-3
- virtual machine security, 3-3
- Java Language Specification
  - adherence to, 4-1
  - compliance with, 2-4
  - error handling capabilities, implementation of, 4-2
  - `java.lang.Error` class, 4-2
- Java ME class libraries, subset of, 6-2
- Java ME platform, classes derived from, 6-10
- Java SE class libraries, subset of, 6-2
- Java SE platform, classes derived from, 6-2
- Java virtual machine
  - application-level security, 3-4
  - architecture of, 3-1
  - bytecodes not supported, 5-1
  - end-to-end security, 3-6
  - exception limitations, 4-2
  - floating point, no support for, 5-1
  - low-level security, 3-4
  - support for dynamic loading of applications, 3-5
  - support for error classes, 4-3
  - verification of class files, 3-4
- Java Virtual Machine Specification, class loading
  - restrictions, 3-6
- `java.io` package, 6-2
- `java.lang.*` package, 6-2
- `java.lang.Error` class, 4-2
- `java.security` package, 6-2
- `java.util` package, 6-2
- `java.util.Properties` class, 6-9
- `javacard.encoding` property, 6-10
- `javax.microedition.io` package, 6-2
- `javax.microedition.pki` package, 6-2

## L

libraries

- classes added, list of, 1-2, 1-3
- classes enhanced, list of, 1-3
- Java Card Virtual Machine architecture, 3-1
- minimum set of, providing, 6-1
- subsets of, 6-2
- `LineNumberTable` attribute, 5-6
- localization classes, list of, 6-6
- `LocalVariableTable` attribute, 5-6
- `LocalVariableTypeTable` attribute, 5-6
- lookup order
  - class files, 5-7
  - defined by, 5-7
  - restrictions on, 5-7
- low-level security, 3-3

## M

memory

- non-volatile, definition, 2-3
- ratio of volatile to non-volatile, 2-3
- requirements, 2-3
- volatile, definition, 2-3

methods

- `Class.getResourceAsStream`, 5-7
- `Object.finalize`, 4-2, 5-4
- removed, 1-2
- `System.getProperty`, 6-10
- user-supplied, constraints on, 5-2

## N

- namespaces, reserved, 6-10
- network bandwidth, conserving, 5-5, 5-8
- network connection, code example of, 6-11
- Normalizer, Glossary-12

## O

- `Object.finalize` method, 5-4
- `Object.finalize` method, exclusion of, 4-2
- optimizations, implementation-level, 2-4, 5-8
- output classes, list of, 6-4
- `OutputStreamWriter` class, 6-8

## P

packages

- `java.io`, 6-2
- `java.lang.*`, 6-2
- `java.security`, 6-2

- java.util, 6-2
- javax.microedition.io, 6-2
- javax.microedition.pki, 6-2
- platform-defined class loaders, 5-3
- properties, javacard.encoding, 6-10
- property definitions, manufacturer-specific, 6-10
- protected content, Glossary-14
- public distribution of applications, 5-6
- public key infrastructure classes, 6-12
- public key infrastructure cryptography, 6-12
- public representation of applications, 5-6

## R

- Reader objects, 6-8
- realm, Glossary-17
- reference support, removed, 1-2
- related abstractions, 6-11
- reserved namespaces, 6-10
- resource files, application-specific, 5-7
- ROMizing classes, 5-8
- runtime verifier, 5-5
- Runtime.ext method, 1-2
- RuntimeInvisibleAnnotations attribute, 5-6
- RuntimeInvisibleParameterAnnotations attribute, 5-6
- RuntimeVisibleAnnotation attribute, 5-6
- RuntimeVisibleParameterAnnotations attribute, 5-6

## S

- sandbox security model
  - description of, 3-4
  - requirements of, 3-5
- search order, class files, 5-7
- security
  - application-level, 3-3
  - end-to-end, 3-3
  - levels of, 3-3
  - low-level, 3-3
  - sandbox model, 3-4
- security architecture, 3-3
- security classes, list of, 6-6
- Signature attribute, 5-6
- smart card devices, *See* devices
- software requirements, 2-4

- SourceFile attribute, 5-6
- StackMapTable attribute, 5-5
- Synthetic attribute, 5-6
- system classes, list of, 6-2
- system packages, protecting, 3-5
- system properties storage, not supported, 6-9
- system resources, securing access to, 3-4
- System.exit method, 1-2
- System.getProperty method, 6-10

## T

- thread groups, 5-3
- thread operations constraints, 5-3
- thread's active context, Glossary-19
- time classes, list of, 6-5
- time zones, support for, 6-5
- transport formats, alternative, 5-8
- type checking, 5-5

## U

- Unicode character support, 6-8
- user-defined class loaders, 5-3
- user-supplied classes, constraints on, 5-2
- user-supplied methods, constraints on, 5-2
- utility classes, list of, 6-5

## V

- virtual machine
  - asynchronous exceptions, unsupported, 4-2
  - class file attributes, list of ignored, 5-6
  - compliance with Java Language Specification, 2-4
  - conformance to Java Card platform, 4-1
  - error handling behavior, 4-3
  - error handling limitations, 4-2
  - features, eliminated, 5-3
  - implementing additional error checks, 4-3, 5-4
  - low-level security, 3-4
  - prelinking classes, 5-8
  - preloading classes, 5-8
  - reading Java class files, 5-6
  - rejecting invalid class files, 5-4
  - ROMizing classes, 5-8
  - system properties storage, not supported, 6-9
  - throwing errors, 4-3
- virtual machine security, 3-3

## Virtual Machine Specification

- calendar classes, 6-5
- characteristics of, 2-1
- CLDC specification, differences from, 1-2
- collection classes, list of, 6-4
- data type classes, list of, 6-3
- devices, characteristics, 2-1
- dynamic downloading of content, 2-2
- error classes, 6-8
- exception classes, 6-6
- hardware requirements, 2-2
- input classes, list of, 6-4
- internationalization classes, 6-6, 6-9
- localization classes, 6-6
- memory, available, 2-3
- minimal host operating system, 2-4
- non-device specific APIs, 2-2
- output classes, list of, 6-4
- scope of, 2-5
- security classes, 6-6
- software requirements, 2-4
- system classes, list of, 6-2
- time classes, 6-5
- time zone support, 6-5
- utility classes, 6-5

## W

- Writer objects, 6-8