



Runtime Environment 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, Mozilla, Netscape, Javadoc, JDK, JVM, NetBeans and Servlet 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, Mozilla, Netscape, Javadoc, JDK, JVM, NetBeans et Servlet 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 regis 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.



Adobe PostScript

Contents

Preface xix

1. Architecture Overview 1-1

1.1 Hardware Overview 1-1

1.1.1 Physical Connectivity 1-2

1.1.2 Logical Connectivity 1-3

1.2 High-level Architecture 1-3

1.3 Runtime Environment for Classic Applet-based Applications 1-5

2. Application Programming Models 2-1

2.1 Web Application Model Overview 2-1

2.2 Applet-based Application Models Overview 2-3

2.3 Unified Naming and Dedicated Application Namespaces 2-5

2.3.1 Unified Naming Scheme 2-5

2.3.2 Dedicated Application Namespaces 2-6

2.3.3 Handling of URIs 2-8

2.4 Context Isolation Basics 2-14

2.4.1 Application Firewall 2-14

2.4.2 Object Access Across Contexts 2-20

2.5 Inter-Application Communication Facilities Overview 2-33

2.6	Applications Not Activated Through a Container-managed Endpoint	2-34
2.7	Multithreading	2-35
2.7.1	Thread Creation	2-36
2.7.2	Thread Execution	2-37
2.7.3	Thread Interruption and Termination	2-40
2.7.4	Thread Ownership	2-40
2.7.5	Thread Safety of API Classes	2-42
2.8	Persistence	2-42
2.8.1	Memory Store and Object Store Terminology	2-42
2.8.2	Persistence By Reachability Principle	2-43
2.8.3	Roots of Persistence	2-44
2.9	Transaction Facility	2-45
2.9.1	Atomicity	2-46
2.9.2	Transaction Demarcation	2-47
2.9.3	Overlapping Transaction Updates	2-53
2.9.4	Transient Arrays and <code>TransientReference</code> Objects	2-54
2.9.5	Power-up After Card Tear	2-54
2.9.6	Aborting A Transaction	2-54
2.10	Restartable Tasks	2-55
2.10.1	Tasks	2-56
2.10.2	Task Registration	2-56
2.10.3	Task Execution	2-56
2.10.4	Task Unregistration	2-57
2.10.5	Lifetime and Persistence of Tasks	2-57
2.10.6	Thread Safety	2-57
2.10.7	Per-Thread Active Context	2-58
2.10.8	Transactional Behavior	2-58

3.	Web Application Environment	3-1
3.1	Servlet Subset Overview	3-1
3.2	Web Application Lifecycle	3-2
3.2.1	Application Module Loading	3-3
3.2.2	Application Instance Identification	3-4
3.2.3	Application Instance Creation	3-5
3.2.4	Application Instance Deletion	3-7
3.2.5	Application Module Unloading	3-8
3.2.6	Restart Upon Platform Reset	3-9
3.2.7	Request Dispatching	3-9
3.2.8	Lifecycle Event Dispatch	3-11
3.2.9	Container-managed Object Lifetime and Persistence	3-11
3.3	Lifecycle and Entry Point Method Invocation	3-14
3.3.1	Servlet, Filter and Listener Lifecycle Methods	3-14
3.3.2	SIO, Event and Restartable Task Entry Point Methods	3-16
3.3.3	Use of Volatile and Persistent Objects	3-16
3.3.4	Multithreading Issues	3-17
3.4	Default Container Behavior and Default Servlet	3-19
3.5	Secure Hosting of Web Applications	3-20
3.5.1	Port-based Virtual Hosting	3-21
3.5.2	Request Dispatching and Redirection	3-23
3.5.3	Retrieving a Web Application Instance's Security Requirements and Credentials	3-26
4.	APDU-based Application Environment	4-1
4.1	Applet Application Overview	4-1
4.2	Applet Application Lifecycle	4-2
4.2.1	Application Module Loading	4-3
4.2.2	Application Instance Identification	4-4

4.2.3	Application Instance Creation	4–5
4.2.4	Application Instance Deletion	4–6
4.2.5	Applet Application Module Unloading	4–6
4.2.6	Restart Upon Platform Reset	4–7
4.2.7	Dispatching APDU Commands	4–7
4.2.8	Container-managed Object Lifetime and Persistence	4–7
4.3	Lifecycle and Entry Point Method Invocation	4–9
4.3.1	Applet Lifecycle Methods	4–9
4.3.2	SIO, Event and Restartable Task Entry Point Methods	4–12
4.3.3	Use of Volatile and Persistent Objects	4–13
4.3.4	Multithreading Issues For Applets	4–13
4.4	Classic Applet Application Support	4–16
4.4.1	Backward Compatibility	4–16
4.4.2	SIO Synchronization Proxy Classes	4–17
4.4.3	Restricted Visibility on Classic Library	4–23
4.4.4	Classic Transaction Model	4–23
4.4.5	Special Security Restrictions	4–23
5.	Card Initialization and Power-up	5–1
5.1	Card Initialization	5–1
5.2	Power-up and Card Reset	5–2
5.3	I/O Interface Reset	5–3
5.3.1	ISO 7816-4 Reset	5–4
5.3.2	Transmission Control Protocol (TCP) Reset	5–4
5.4	Concurrently Active Interfaces	5–5
6.	Security and Access Control Mechanisms	6–1
6.1	Security Policy	6–1
6.1.1	Permission-based Security Policy	6–2

6.1.2	Role-based Security Policy	6-3
6.1.3	Effective Application Security Policy	6-3
6.2	Permission-based Security	6-4
6.2.1	Permissions	6-4
6.2.2	Protection Domains	6-13
6.2.3	Assigning Permissions	6-17
6.2.4	Checking of Permissions	6-17
6.2.5	Security Policy Enforcement	6-19
6.3	Role-based Security	6-23
6.3.1	User Role-based Security	6-23
6.3.2	Client Role-based Security	6-27
6.4	User Authentication and Authorization	6-28
6.4.1	Scheme-specific Authenticators	6-30
6.4.2	Global Authentication of Card Holders	6-32
6.4.3	Session-scoped Authentication of Web Users	6-33
6.4.4	Application-managed Authentication	6-34
6.4.5	Web Container-managed Authentication	6-35
6.4.6	Card Holder Authorization For Remotely Accessible Applications	6-40
6.5	On-card Client Application Authentication and Authorization	6-46
6.5.1	On-card Client Application Authentication	6-46
6.5.2	Authentication Session Duration	6-48
6.6	Security Requirements and Credential Management of Secure Communications	6-49
6.6.1	Assignment of Security Requirements and Credential Managers	6-50
6.6.2	Retrieving Security Requirements and Credential Managers For Establishing Connections	6-51
6.6.3	Invocation of Security Requirements and Credential Managers	6-53

- 6.7 Code Isolation 6–57
 - 6.7.1 Class Loader Delegation Hierarchy 6–58
 - 6.7.2 Class Loading Delegation Principle 6–60
 - 6.7.3 User-defined Class Loaders 6–62
 - 6.7.4 Class-Path Resource Lookup 6–62
- 6.8 Package Access Control 6–63
 - 6.8.1 Built-in Checks 6–63
 - 6.8.2 Package Sealing Checks 6–64
 - 6.8.3 Restriction on the Use of the `Class.forName` Method 6–64
- 6.9 Context Isolation Enhancements 6–65
 - 6.9.1 Context Switches 6–65
 - 6.9.2 Application Namespace Enforcement 6–65
 - 6.9.3 Ownership of Transferable Objects 6–66
- 7. Inter-application Communication 7–1**
 - 7.1 Security Containment Mechanisms 7–1
 - 7.2 Object Ownership Transfer Mechanism 7–3
 - 7.2.1 Transferable Classes 7–3
 - 7.2.2 Transferring Object Ownerships 7–5
 - 7.2.3 Defensive Copy 7–6
 - 7.2.4 Thread Safety 7–6
 - 7.2.5 Transactional Behavior 7–6
 - 7.3 Shareable Interface Object-based Services 7–7
 - 7.3.1 SIO-based Service Definition and Identification 7–8
 - 7.3.2 SIO-based Service Factory Registration 7–11
 - 7.3.3 SIO-based Service Lookup 7–13
 - 7.3.4 Role-based Security for SIO-based Services 7–14
 - 7.3.5 Lifetime and Persistence of SIO-based Services 7–15
 - 7.3.6 Thread Safety 7–15

7.3.7	Per-Thread Active Context	7-15
7.3.8	Transactional Behavior	7-16
7.4	Events	7-16
7.4.1	Event Definition and Identification	7-18
7.4.2	Event Listener Registration	7-22
7.4.3	Event Notification	7-23
7.4.4	Role-based Security for Events	7-24
7.4.5	Lifetime and Persistence of Event Listeners	7-25
7.4.6	Thread Safety	7-25
7.4.7	Per-Thread Active Context	7-26
7.4.8	Transactional Behavior	7-27
8.	Card Management	8-1
8.1	The Card Manager Application	8-1
8.2	The Card Management Facility	8-2
8.3	Unit of Distribution and Deployment	8-2
8.4	Distribution Formats	8-4
8.4.1	Application Module Distribution Format	8-4
8.4.2	Extension Library Distribution Format	8-8
8.4.3	Classic Library Distribution Format	8-9
8.5	Descriptor Formats	8-10
8.5.1	Conventions Used in XML Descriptor Element Diagrams	8-10
8.5.2	Common Rules for Processing the XML Descriptors	8-10
8.5.3	Java Card Platform-specific Application Descriptor	8-11
8.5.4	Web Application Deployment Descriptor	8-15
8.5.5	Applet Application Deployment Descriptor	8-15
8.5.6	The Runtime Descriptor	8-17
8.6	Loading Application Modules	8-25
8.6.1	Code Isolation and Class File Lookup Order Requirements	8-27

8.6.2	Class Dependency Resolution Requirements	8-27
8.6.3	Class Pre-loading Optimizations	8-28
8.7	Loading Libraries	8-29
8.8	Creation of Application Instances	8-31
8.9	Deletion of Application Instance	8-33
8.9.1	Multiple Application/Applet Instance Deletion	8-35
8.10	Unloading of Deployment Units	8-36
8.10.1	Application Module Unloading with Instance Deletion	8-37
9.	File System	9-1
9.1	File System Requirements	9-1
9.2	File System Object Identification	9-2
9.2.1	Application-private File System Objects	9-3
9.3	File Access Permissions	9-3
9.4	Atomicity and Transactional Behavior	9-4
9.5	Generic Connection Framework-based File Access	9-5
9.6	File Resource Event Notifications	9-5
9.7	Thread Safety	9-6
9.8	Platform Reset Behavior	9-6
A.	Default Platform Security Policy	APPENDIXA-1
A.1	Permissions in Default Protection Domain for Web Applications	APPENDIXA-2
A.2	Permissions in Default Protection Domain for Extended Applets	APPENDIXA-4
A.3	Permissions in Default Protection Domain for Classic Applets	APPENDIXA-6
A.4	Permissions in Default Protection Domain for Card Management Applications	APPENDIXA-7
B.	Security Annotations	APPENDIXB-1

B.1	Annotations Defined	APPENDIXB-3
B.1.1	Type Annotations	APPENDIXB-3
B.1.2	Interface Annotation	APPENDIXB-5
B.1.3	Method Annotations	APPENDIXB-5
B.2	Semantics of Annotations	APPENDIXB-7
B.2.1	Scope of Annotations	APPENDIXB-7
B.2.2	Annotated APIs	APPENDIXB-8

Glossary **Glossary-1**

Index **Index-1**

Figures

FIGURE 1-1	Connectivity Layers and Protocol Stacks	1–2
FIGURE 1-2	High-level Architecture	1–5
FIGURE 1-3	Classic Application Development and Deployment Process for Backward Compatibility	1–7
FIGURE 2-1	Contexts Within the Java Card Platform’s Object System	2–16
FIGURE 2-2	Context Switching and Object Access	2–18
FIGURE 3-1	Request Dispatching and Redirection to Web Applications’ Secure Ports	3–25
FIGURE 4-1	SIO Synchronization Proxy Mechanism (Collaboration Diagram)	4–18
FIGURE 6-1	Permission Class Hierarchy	6–5
FIGURE 6-2	Access Control Decision Principle (Sequence Diagram)	6–20
FIGURE 6-3	Access from a Card Holder-Facing Client (Collaboration Diagram)	6–42
FIGURE 6-4	Access from a Non Card Holder-Facing Client (Collaboration Diagram)	6–43
FIGURE 6-5	On-card Client Application Authentication (Sequence Diagram)	6–48
FIGURE 6-6	Security Requirements and Credential Manager Assignment and Lookup Order	6–53
FIGURE 6-7	Invocation of CredentialManager Methods During a TLS Handshake with Asymmetric, or Public Key, Cryptography Authentication	6–56
FIGURE 6-8	Invocation of CredentialManager Methods During a TLS Handshake with Symmetric, or Pre-shared Secret Key, Cryptography Authentication	6–57
FIGURE 6-9	Class Loader Delegation Hierarchy	6–60
FIGURE 7-1	Context Isolation, Code Isolation and Inter-Application Communications	7–2
FIGURE 7-2	SIO-based Service Registry Interactions (Collaboration Diagram)	7–8
FIGURE 7-3	Event Registry Interactions (Collaboration Diagram)	7–18

FIGURE 8-1	Web Application Module Distribution Format	8–5
FIGURE 8-2	Extended Applet Application Module Distribution Format	8–6
FIGURE 8-3	Classic Applet Application Module Distribution Format	8–7
FIGURE 8-4	Java Platform Standard Edition Library JAR Format	8–8
FIGURE 8-5	Classic Library Distribution Format	8–9
FIGURE 8-6	Conventions Used in Diagrams of Elements	8–10
FIGURE 8-7	<code>javacard-app</code> Element Structure	8–12
FIGURE 8-8	<code>card-holder-authorization</code> Element Structure	8–13
FIGURE 8-9	<code>dynamically-loaded-classes</code> Element Structure	8–14
FIGURE 8-10	<code>shareable-interface-classes</code> Element Structure	8–14
FIGURE 8-11	<code>security-role</code> Element Structure	8–15
FIGURE 8-12	<code>applet-app</code> Element Structure	8–16
FIGURE 8-13	<code>applet</code> Element Structure	8–17
FIGURE 8-14	Runtime Descriptor Structure	8–20

Tables

TABLE 2-1	Unified Naming Scheme 2–7
TABLE 2-2	URI Pattern Types 2–13
TABLE 2-3	Classic Set of Permanent Java Card RE EPO 2–22
TABLE 2-4	Classic Set of Temporary Java Card RE EPO 2–22
TABLE 2-5	Extended Set of Permanent Java Card RE EPO 2–22
TABLE 2-6	Card Management Set of Permanent Java Card RE EPO 2–22
TABLE 2-7	Classic Set of Global Arrays 2–24
TABLE 2-8	Transaction Facility Managed Transitions On Method Entry and Exit 2–52
TABLE 3-1	Port-based Hosting of Web Application Instances 3–22
TABLE 4-1	AID Format 4–4
TABLE 6-1	Named Permission Classes 6–6
TABLE 6-2	URI-named Permission Classes 6–10
TABLE 6-3	Application- and Library-defined Permissions 6–12
TABLE 6-4	Biometric Scheme Name to <code>BioBuilder</code> Biometric Type Constant Mapping 6–32
TABLE 6-5	Card Holder Authorization Requirements for Access by User 6–43
TABLE 6-6	Supported Role-to-User Mapping For Locally Accessible and Remotely Accessible Applications 6–44
TABLE 6-7	Connection or Access Endpoint URIs and Modes of Operations 6–54
TABLE 7-1	Standard SIO-based Services 7–10
TABLE 7-2	Application-defined Services 7–11
TABLE 7-3	Platform and Standard Events 7–21

TABLE 7-4	Application-defined Events 7–22
TABLE 7-5	Platform and Card Management-related Standard Application Events 7–26
TABLE 9-1	Application-private File System Object Identification 9–3
TABLE 9-2	Operations Granted By Read and Write Permissions on Files and Directories 9–4
TABLE A-1	Default Included Permission Set of the Default <code>Web</code> Protection Domain APPENDIXA–2
TABLE A-2	Excluded Permission Set of the Default <code>Web</code> Protection Domain APPENDIXA–3
TABLE A-3	Default Included Permission Set of the Default <code>Extended</code> Protection Domain APPENDIXA–4
TABLE A-4	Excluded Permission Set of the Default <code>Extended</code> Protection Domain APPENDIXA–5
TABLE A-5	Default Included Permission Set of the Default <code>Classic</code> Protection Domain APPENDIXA–6
TABLE A-6	Excluded Permission Set of the Default <code>Classic</code> Protection Domain APPENDIXA–6
TABLE A-7	Default Included Permission Set of the Default <code>CardManagement</code> Protection Domain APPENDIXA–7
TABLE A-8	Excluded Permission Set of the Default <code>CardManagement</code> Protection Domain APPENDIXA–8

Code Examples

- [CODE EXAMPLE 2-1](#) Transaction Demarcation Example 2-48
- [CODE EXAMPLE 4-1](#) SIO Synchronization Proxy Classes Example 4-20
- [CODE EXAMPLE 6-1](#) HTML Form For PIN-based Authentication 6-39
- [CODE EXAMPLE 6-2](#) Deployment Descriptor Configuration For PIN-based Authentication 6-40
- [CODE EXAMPLE 6-3](#) HTML Form For Fingerprint Authentication 6-40
- [CODE EXAMPLE 6-4](#) Deployment Descriptor Configuration For Fingerprint Authentication 6-40
- [CODE EXAMPLE B-1](#) Class Security Annotation Example APPENDIXB-3
- [CODE EXAMPLE B-2](#) Interface Security Annotation Example APPENDIXB-5
- [CODE EXAMPLE B-3](#) Method Security Annotation Example APPENDIXB-6

Preface

This book provides a specification of the runtime environment (RE) for the Java Card™ Platform, Version 3.0.1, Connected Edition. The Connected RE differs from the RE available in the Classic Edition of the Java Card Platform, Version 3.0.1. In this book, “Java Card RE” is used to refer to the RE in the Connected Edition. Furthermore, in this book “Java Card 3 Platform” refers to both versions 3.0 and 3.0.1 to distinguish them from all earlier versions.

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.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" when used in uppercase in this specification are to be interpreted as follows:

- **MUST** - This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.
- **MUST NOT** - This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.

- **SHOULD** - This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
 - **SHOULD NOT** - This phrase, or the phrase "NOT RECOMMENDED", mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
 - **MAY** - This word, or the adjective "OPTIONAL", mean that an item is truly optional.
-

Overview of New Features

In the Java Card 3 Platform, Connected Edition, a new virtual machine and a new runtime environment were introduced that support three application models, as opposed to the single application model supported by the Classic Edition and previous releases of the Java Card platform.

For a more detailed introduction to the three application models, see [Chapter 2](#).

- **Classic applet application model**- applet-based applications with the same capabilities as those in previous versions of the Java Card platform and in the Classic Edition. These applets use the APDU-based scheme of communication with the card.

In this book, all the chapters and the appendix apply to classic applets, except [Chapter 3](#). However, more information specific solely to classic applets can be found in the classic runtime environment specification, *Runtime Environment Specification for the Java Card Platform, v3.0.1, Classic Edition*.

- **Extended applet application model**- applets with more advanced capabilities than classic applets. These applets use the APDU-based scheme of communication with the card.

In this book, all the chapters and the appendix apply to extended applets, except [Chapter 3](#).

- **Web application model**- applications based on servlets that use the HTTP protocol to support a web-based scheme of communication with the card.

In this book, all the chapters and the appendix apply to web applications, except [Chapter 4](#). However, more information specific to servlets can be found in *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition*.

Before You Read This Specification

To fully use the information in this document, you must have thorough knowledge of the topics discussed in these documents:

- *Java™ Servlet Specification, Version 2.4*
- *Java™ Servlet Specification for the Java Card Platform, Version 3.0.1, Connected Edition*

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

Chapter 1, “Architecture Overview” describes the architecture of the Java Card 3 Platform, Connected Edition.

Chapter 2, “Application Programming Models” briefly describes the three application models.

Chapter 3, “Web Application Environment” describes the web application environment in more depth.

Chapter 4, “APDU-based Application Environment” describes the two APDU-based applet application environments in more depth.

Chapter 5, “Card Initialization and Power-up” describes card initialization and start-up.

Chapter 6, “Security and Access Control Mechanisms” describes security and access control mechanisms.

Chapter 7, “Inter-application Communication” describes inter-application communication.

[Chapter 8, “Card Management”](#) describes the card manager application and the card management facility.

[Chapter 9, “File System”](#) describes the file system which may, optionally, be supported.

[Appendix A, “Default Platform Security Policy”](#) lists the permissions for the default platform security policy.

[Appendix B, “Security Annotations”](#) describes the optional annotations that can be added to applications to make them more secure.

[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.
AaBbCc123	What you type, when contrasted with on-screen computer output	% su 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 following documents might be of interest.

- *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Connected Edition*
- *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*
- *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Classic Edition*
- *Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition*
- *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition* containing the Javadoc™ tool files related to servlets and to the Java Card RE
- *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Classic Edition* containing the Javadoc tool files related to the Classic Edition's Java Card RE and classic applets
- *Java™ Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999)
- *JAR File Specification*
(<http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html>)

Documentation, Support, and Training

Sun Function	URL
Documentation	http://www.sun.com/documentation/
Support	http://www.sun.com/support/
Training	http://www.sun.com/training/

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

Please include the title of your document with your feedback:

Runtime Environment Specification, Java Card Platform, v3.0.1, Connected Edition.

Architecture Overview

This chapter introduces the runtime environment in the Connected Edition of the Java Card 3 Platform, including its hardware and connectivity, and describes how it achieves backward compatibility by emulating the Classic Edition's runtime environment (RE).

The following components of the architecture are described in this chapter:

- [Hardware Overview](#)
- [High-level Architecture](#)
- [Runtime Environment for Classic Applet-based Applications](#)

1.1 Hardware Overview

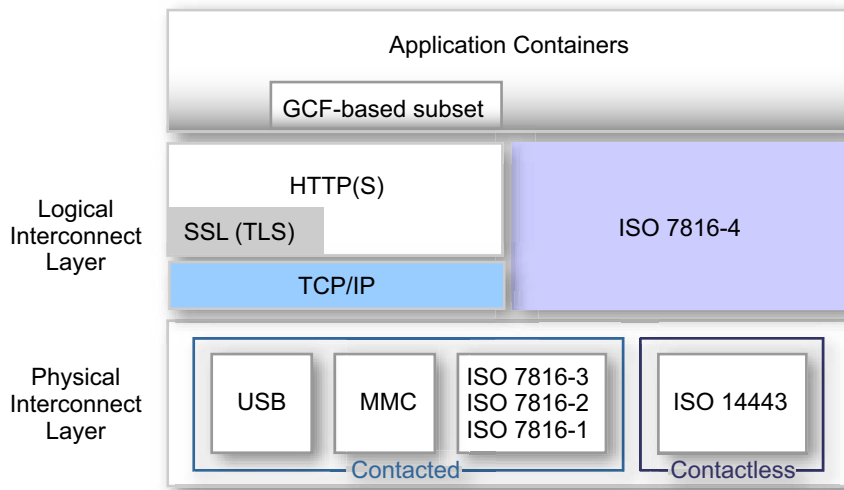
The Java Card 3 Platform, Connected Edition, is intended to run on a wide variety of smart card and secure token devices with constrained resources. The **typical** hardware configuration of the targeted devices is as follows:

- A 32-bit processor
- 128 kilobytes of EEPROM
- 512 kilobytes of ROM
- 24 kilobytes of RAM
- Low power consumption, typically not battery powered, and operating with an external or magnetically-induced power source
- Full duplex, high-speed interface with a hosting device
- Connectivity to some kind of network, typically through the hosting device:
 - Over the full duplex, high-speed interface
 - One or more logical network interfaces

- Two-way connections, possibly intermittent, with limited bandwidth
- Limited timer and real-time clock capabilities. The minimum is a low precision interval timer and an implementation specific means of re-synchronization of the wall clock time with an external time source.

FIGURE 1-1 depicts the connectivity layers and protocol stacks of the Java Card Platform, Connected Edition.

FIGURE 1-1 Connectivity Layers and Protocol Stacks



1.1.1 Physical Connectivity

The target devices for the Java Card Platform, Connected Edition, typically have a high-speed contacted physical interface. In addition, they may have additional I/O interfaces, including contactless physical interfaces.

The high-speed contacted interface is typically one of the following:

- a Universal Serial Bus (USB) interface
- a MultiMediaCard (MMC) interface
- an ISO 7816-3 interface

The contactless interface is typically an ISO 14443-compliant interface.

1.1.2 Logical Connectivity

A Java Card Platform implementation MUST provide to applications a logical network interface supporting the following network protocols:

- TCP (Transmission Control Protocol)
- Transport Layer Security (TLS)
- Hypertext Transfer Protocol (HTTP)
- Secure Hypertext Transfer Protocol (HTTP over TLS)

This logical network interface MAY additionally support the following network protocol:

- Universal Datagram Protocol (UDP)

A Java Card Platform implementation is not required to support the TCP/IP or UDP/IP on card. Connections over HTTP, HTTPS, and TLS protocols and datagrams over the UDP protocol MAY be supported over IP, as well as non-IP protocols, by using a gateway on the terminal or hosting device. The TLS protocol MUST be supported on card.

Applications MUST NOT be required to know that a non-IP protocol is being used. Applications MUST only gain access to the logical network interface through the Generic Connection Framework (GCF) API and indirectly through the *web application container*, see [Section 2.1, “Web Application Model Overview” on page 2-1](#) and [Chapter 3](#). A Java Card Platform implementation MUST recognize both IPv4 and IPv6 literal address formats, such as when passed inside URIs to the GCF API, but MAY only support such addresses according to the underlying supported protocols. See [Section 2.3.3, “Handling of URIs” on page 2-8](#) for details on the URI format that a Java Card Platform implementation MUST support.

A Java Card Platform implementation MUST provide access to the ISO 7816-4 interface to applet-based applications indirectly through the applet container, see [Section 2.2, “Applet-based Application Models Overview” on page 2-3](#) and [Chapter 4](#).

1.2 High-level Architecture

The high-level architecture of the Java Card Platform, Connected Edition, is illustrated in [FIGURE 1-2](#). For additional information on the new architecture in version 3.0.1, see [“Overview of New Features” on page xx](#).

At the core of the Java Card Platform is the *Java Card Virtual Machine* (Java Card VM). The Java Card VM is implemented on top of the *device's operating system*. The Java Card VM is based on a subset of the Connected Limited Device Configuration v1.1 Virtual Machine, which is itself compliant with the Java Virtual Machine Specification and Java Language Specification. The Java Card VM is described in the *Virtual Machine Specification for the Java Card Platform, v3.0.1, Connected Edition*.

On top of the Java Card VM are the Java Card Platform, Connected Edition libraries and Java Card Classic Edition libraries:

- The **Java Card Classic Edition libraries** are the libraries for the Java Card classic APIs, which are backward compatible with the APIs for the Java Card Platform, release 2.2.2. These APIs are described in the *Java Card API Specification, Classic Edition*. A superset of these APIs are also included in the *Java Card API Specification, Connected Edition*. The Java Card Classic Edition APIs include the applet framework API depicted above on the stack.
- The **Java Card Platform, Connected Edition libraries** are the libraries for the Java Card Platform, Connected Edition API. Some of these libraries are defined by the Java Card VM Specification for the Java Card Platform, Connected Edition, others correspond to additional, advanced, functionality described in the current specification document, as well as in the *Java Card API Specification, v3.0.1, Connected Edition*. The Java Card Platform, Connected Edition APIs include the Servlet API depicted above on the stack

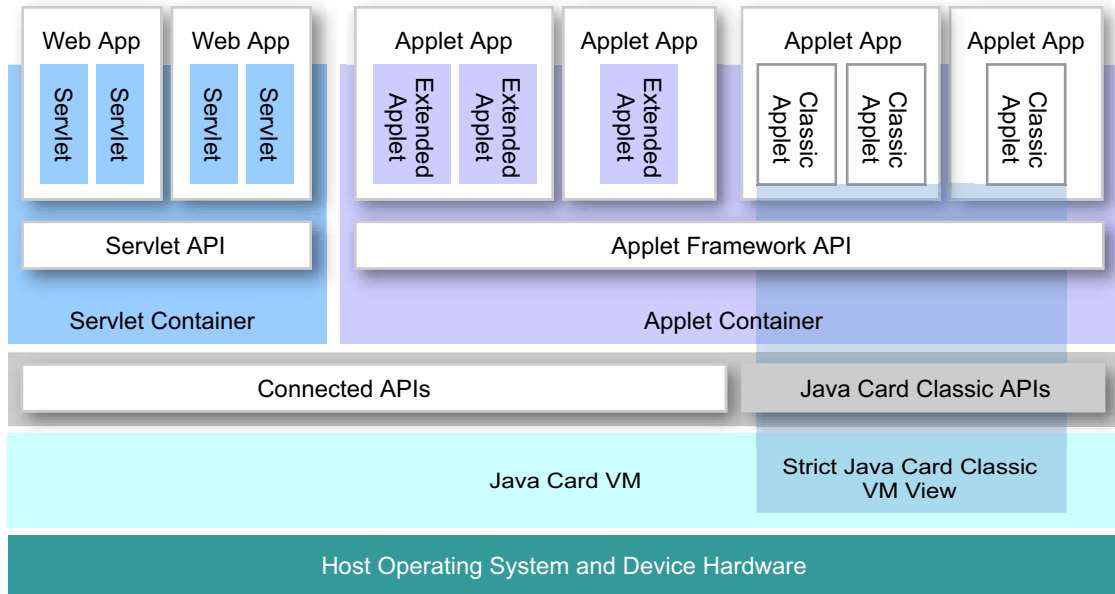
Not depicted on [FIGURE 1-2](#) are libraries from standards bodies that MAY be installed on a Java Card Platform to extend its functionality. These libraries and the functionality they provide MUST be compliant with the specifications of the Java Card Platform, Connected Edition.

On top of the Java Card Platform, Connected Edition and Java Card Classic Edition libraries are the two application containers:

- The *applet container* that contains applet-based applications and manages their lifecycle through the *applet framework API*. Applet-based applications are of two types:
 - **Classic Applet-based Applications.** These applications correspond to ISO 7816 APDU applet-based applications that have been developed against the Java Card classic API and that run in a Java Card v2.2.2 runtime emulation environment that, in particular, guarantees their thread-safety. See [Section 1.3, “Runtime Environment for Classic Applet-based Applications”](#) on page 1-5 for more details.
 - **Extended Applet-based Applications.** These applications correspond to ISO 7816 APDU applet-based applications that have been developed to use the advanced capabilities of the Java Card Platform, Connected Edition, such as multi-threading and networking.

- The *servlet container* that contains web applications and manages their lifecycle through the *servlet API*. These applications correspond to HTTP servlet-based applications that have been developed to use the advanced capabilities of the Java Card Platform, Connected Edition.

FIGURE 1-2 High-level Architecture



1.3 Runtime Environment for Classic Applet-based Applications

The Java Card Platform Connected Edition is backward compatible with Java Card Platform, Classic Edition, which is itself compatible with the release 2.2.2 of the Java Card Platform. The Java Card Platform, Connected Edition, **MUST** support running classic applet-based applications in a Java Card Platform, v2.2.2 runtime emulation environment, which enforces the Java Card RE v2.2.2 runtime requirements.

This Java Card Platform, v2.2.2 runtime emulation environment **MUST** guarantee the following:

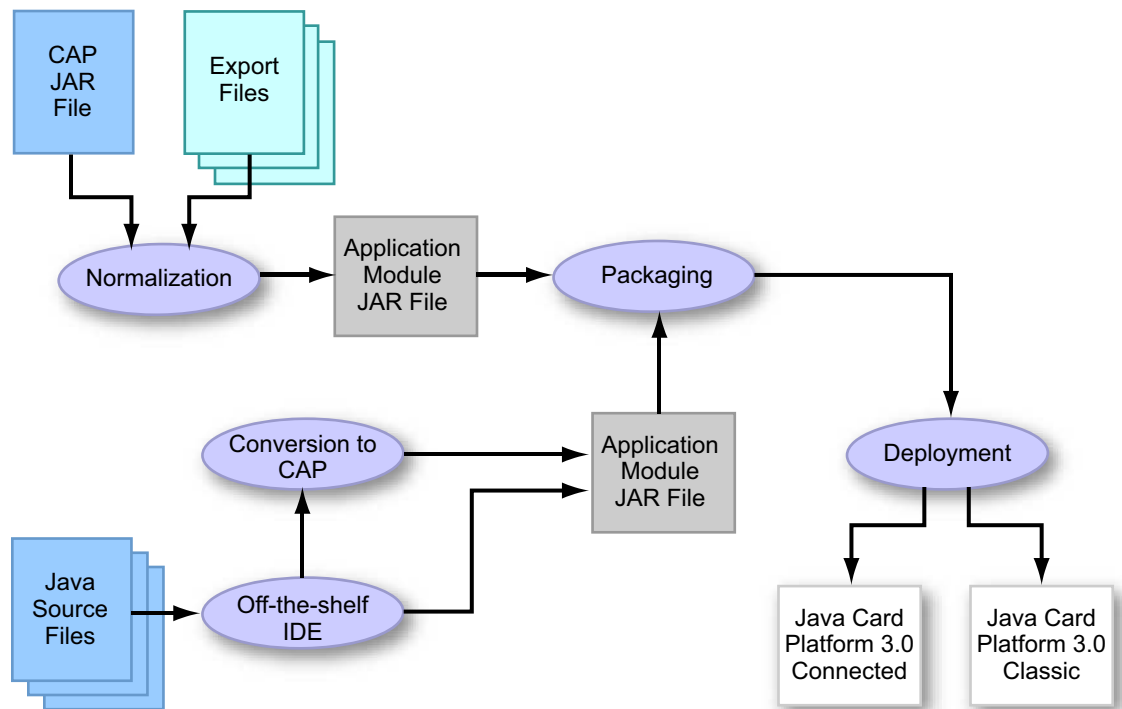
- Classic applet-based applications and libraries have access to Java Card Platform, v2.2.2 APIs.

- Classic libraries are only accessible to Classic applet-based applications.
- Classic applet-based application and library code are executed by a single thread at any time (thread-safety).
- Access control and ownership tagging performed by the Java Card platform firewall are identical to that of the Java Card Platform, v2.2.2 firewall, especially with respect to the Java Card RE entry point object and global array objects defined in Java Card Platform, v2.2.2.
- The persistence model supports the specialized Java Card Platform, v2.2.2 transient array objects.
- The transaction model supports the single transaction rollback logging view of classic applications, as well as the associated API unchanged.
- The on-card initialization of classic applet-based application and library classes does not result in different runtime effects and semantics when compared to using off-card, precomputed, static field images.

Off-card compatibility tools MUST allow for 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. These tools MUST also allow for Java applications packaged for the Java Card Platform, Version 2.2.2, to be transformed, through a process called *normalization*, and repackaged for deployment on both the Java Card 3 platform Connected and Classic Editions.

FIGURE 1-3 depicts the development and deployment process for classic applet-based applications that ensures for these applications the backward compatibility and interoperability with both the Java Card 3 platform Connected and Classic Editions.

FIGURE 1-3 Classic Application Development and Deployment Process for Backward Compatibility



See the *Runtime Environment Specification for the Java Card Platform, v3.0.1, Classic Edition* for an exhaustive list of the requirements for the classic Java Card RE.

Application Programming Models

Three application models are supported in the runtime environment for the Java Card Platform, v3.0.1, Connected Edition. All three, including the web application model, and the two applet-based application models are described in this chapter.

Also described in this chapter are the naming, dispatching and thread management aspects of application programming, as well as object persistence and transaction management.

This chapter describes the following aspects of the application programming model:

- [Web Application Model Overview](#)
- [Applet-based Application Models Overview](#)
- [Unified Naming and Dedicated Application Namespaces](#)
- [Context Isolation Basics](#)
- [Inter-Application Communication Facilities Overview](#)
- [Applications Not Activated Through a Container-managed Endpoint](#)
- [Multithreading](#)
- [Persistence](#)
- [Transaction Facility](#)
- [Restartable Tasks](#)

2.1 Web Application Model Overview

The Java Card Platform, Connected Edition MUST support a subset of the *Java Servlet Specification v2.4* web application model. This section gives an overview of the web application model, whereas the supported subset of the web application model is described in detail in the *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition*.

Web applications interact with off-card web clients via a request/response paradigm implemented by the web container. On the Java Card Platform, the web container **MUST** support HTTP and HTTPS as protocols for requests and responses.

The web container's main responsibilities are:

- contain and manage web applications and their components through their lifecycle.
- provide the network services over which requests and responses are sent, decode MIME-based requests, and format MIME-based responses.
- manage security of web resources such as user authentication and authorization, as well as secure communication requirements.

The server connection endpoints over which requests and responses are received from clients and sent to clients, respectively, by the web container are called *container-managed endpoints*.

A web application is typically composed of:

- **Servlets, request and response filters, lifecycle event listeners and other business logic and utility classes:** Servlets and filters are the components of a web application that generate dynamic content, which is content that is computed upon requests from clients and sent back to the clients as part of the responses. These components are managed and invoked by the container in accordance with their respective lifecycle.
- **Static resources such as HTML documents and embedded images or objects:** Static resources constitute what is called static content, which is content that resides in files or the equivalent. Static content is requested by clients and sent back to the clients unchanged or with some form of pre-processing, such as by response filters.
- **A web application deployment descriptor:** The deployment descriptor conveys the elements and configuration information of a web application between the different actors of an application lifecycle (application developers, application assemblers, and deployers). The deployment descriptor describes the web application's components and how they are mapped to client requests and their security requirements (user authentication and authorization, and secure communication requirements).

Developers are required to implement the following components of web applications: servlets, request and response filters, and lifecycle event listeners.

Web applications are deployed into the web container and are uniquely identified by the specific path at which they are rooted in the web container namespace. This path is called the *context path* of the web application (see [Section 2.3, "Unified Naming and Dedicated Application Namespaces" on page 2-5](#)). All requests for a resource whose URL path component is prefixed with that context path will be routed to that application to be further dispatched to the targeted web resources of that application. Based on the mappings from URL path to web components configured

for that application, web components (servlets and filters) are invoked with objects encapsulating the requests received and the responses to be sent. The web container is multithreaded. Web applications must, therefore, account for multithreading and for servicing multiple requests concurrently (see [Section 2.7, “Multithreading” on page 2-35](#)).

On the Java Card Platform, web applications are limited when compared to more regular web platforms, but they may also use many of the facilities provided by the platform, such as persistence (see [Section 2.8, “Persistence” on page 2-42](#)), transactions (see [Section 2.9, “Transaction Facility” on page 2-45](#)), inter-application communication (see [Section 2.5, “Inter-Application Communication Facilities Overview” on page 2-33](#)), and restartable tasks facilities (see [Section 2.10, “Restartable Tasks” on page 2-55](#)).

The Java Card Platform runtime environment specifics for web applications are described in detail in [Chapter 3](#). On the Java Card Platform, web applications include additional descriptors specific to the Java Card Platform beyond the regular web application deployment descriptor. These descriptors, as well as the Java Card Platform-specific subset of the regular web application deployment descriptor, are described in detail in [Chapter 8](#).

2.2 Applet-based Application Models Overview

The Java Card Platform, Connected Edition MUST support both the classic applet application model and the extended applet application model. This section gives an overview of both applet application models in general, whereas the classic applet application model is described in detail in the *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*.

Applet applications interact with off-card applet clients via the APDU command/response paradigm implemented by the applet container. On the Java Card Platform, the applet container MUST support the ISO 7816-4 defined APDU protocol for commands and responses. The applet container’s main responsibilities are to contain and manage applet applications and their components through their lifecycle.

Extended applet applications differ from classic applet applications in the following respects:

- application code may comprise multiple packages
- application code may use all facilities and libraries of the Java Card Platform, Connected Edition

- may execute concurrently on different threads to process APDU commands received over different I/O interfaces

Classic applet applications have the following characteristics:

- supported on both Java Card Platform, Connected Edition as well as Java Card Platform, Classic Edition. Application code must use only the facilities defined in the *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*, *Virtual Machine Specification, Java Card Platform, v3.0.1, Classic Edition* and the *Application Programming Interface Specification, Java Card Platform, v3.0.1, Classic Edition*.
- application code comprises only one package
- applications execute in a single threaded environment

An applet application is typically composed of:

- **Applet and other business logic and utility classes:** Applets are the components of an applet application that process incoming APDU commands from clients and send APDU responses back to the clients. These components are managed and invoked by the container in accordance with their respective lifecycle.
- **An applet application deployment descriptor:** The deployment descriptor contains information to uniquely identify each concrete applet class in the application.

Applet applications are deployed into the applet container and are uniquely identified on the card, by the URI paths corresponding to their applet instance Application Identifiers (AID), which they are rooted in the applet container namespace. These paths define unique namespaces for the applet application on the card (see [Section 2.3, “Unified Naming and Dedicated Application Namespaces” on page 2-5](#)). Applet instances of an applet application are selected for receiving APDU commands over logical communication channels using the ISO 7816-4 defined APDU SELECT command containing the applet instance AID data.

The applet container dispatches APDU commands to classic applications sequentially even when received concurrently over different I/O interfaces.

The applet container is able to dispatch APDU commands received on different I/O interfaces concurrently to extended applet applications. Extended applet applications must, therefore, account for multithreading and for servicing multiple commands concurrently. (See [Section 2.7, “Multithreading” on page 2-35](#).)

On the Java Card Platform, extended applet applications may also use many of the facilities provided by the platform, such as persistence (see [Section 2.8, “Persistence” on page 2-42](#)), transactions (see [Section 2.9, “Transaction Facility” on page 2-45](#)), inter-application communication (see [Section 2.5, “Inter-Application Communication Facilities Overview” on page 2-33](#)), and restartable tasks facilities (see [Section 2.10, “Restartable Tasks” on page 2-55](#)).

The Java Card Platform runtime environment specifics for applet applications are described in detail in [Chapter 4](#). On the Java Card Platform, applet applications include additional descriptors specific to the Java Card Platform beyond the regular applet application deployment descriptor. These descriptors, as well as the Java Card Platform-specific subset of the regular applet application deployment descriptor, are described in detail in [Chapter 8](#).

2.3 Unified Naming and Dedicated Application Namespaces

The Unified Naming scheme supported by the Java Card Platform, Connected Edition allows for applications of all types and their respective resources to be uniquely named and uniformly addressed on the platform. Each application is assigned a dedicated resource namespace rooted under its own unique name.

2.3.1 Unified Naming Scheme

All applications **MUST** be named with a relative URI (or more properly, a relative URI-reference), one that starts with “//”. These URIs are called *application URIs*.

Every web application **MUST** be named with a relative URI that uses the *null* registry-based authority. The path component of a web application URI **MUST** correspond exactly to its *context path*. Web application URIs are of the following form:

`//<context path>`

where *<context path>* is a web application context path (which **MUST** always start with a ‘/’ and which **MUST** not include any path segment parameters), see the “Servlet Context” chapter of the *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition*.

Every applet application **MUST** be named with a relative URI that uses the “aid” registry-based authority. The path component of an applet application URI **MUST** correspond to its hexadecimal encoded *AID* (Application Identifier). Applet application URIs are of the following form:

`//aid/<RID>/<PIX>`

where *<RID>* (resource identifier) and *<PIX>* (proprietary identifier extension) are the two classical components of applet application AID, see the “Java Card Applet Lifetime” chapter of *Runtime Environment Specification for the Java Card Platform, v3.0.1, Classic Edition*.

All the bytes of the RID and PIX including any leading 0 byte values MUST be represented in the character string notation. A RID byte string is 5 bytes in length. Its character string equivalent MUST be exactly 10 characters in length. A PIX byte string can be from 0 to 11 bytes in length. A PIX byte string of N bytes in length MUST have an equivalent character string representation of exactly $2*N$ characters in length.

In order to comply with the requirement for non-overlapping application URI namespaces (defined in [Dedicated Application Namespaces](#)), when a `<PIX>` component of an AID URI is 0 in length, it MUST be replaced by `'-'`. In such a case, the AID URI has the following form:

```
//aid/<RID>/-
```

AID objects which encapsulate applet application AIDs can be transformed into the equivalent application URI String objects and vice versa using the `getURI(AID)` and `getAID(String)` methods of the `javacardx.framework.JCSystem` class.

2.3.2 Dedicated Application Namespaces

Two application instances MUST NOT have the same application URI name.

An application instance MUST NOT be named with the reserved `///platform` and `///standard` URIs or be named relatively to these reserved URIs.

Application URI namespaces MUST NOT overlap. For example, there must not be two web application instances named `///transit` and `///transit/pos`.

An application URI defines the root of a dedicated namespace within which all its resources are named. All resources of an application MUST be named relatively to that application's URI.

Any attempt to create or register resources under another application's namespace MUST result in a security exception.

Creation and registration of resources under the reserved `///platform` and `///standard` namespaces MUST be subject to resource-type-specific access control.

Access to resources under another application's namespace MUST be subject to resource-type-specific access control.

TABLE 2-1 shows the URI namespace formats for applications as well as resources on the Java Card Platform.

TABLE 2-1 Unified Naming Scheme

URI Naming Format	Example
Web Applications	
//<context path>	///transit/pos
<context path>	/transit/pos
Applet Applications	
//aid/<RID>/<PIX>	//aid/a000000062/03010c0d01
SIO-based Services	
SIO root URIs:	
• sio://<context path>	sio:///transit/pos
• sio://aid/<RID>/<PIX>	sio://aid/a000000062/03010c0d01
• sio://<reserved path>	sio:///standard
sio://<context path>/<sio path>	sio:///transit/pos/ticketbook
sio://aid/<RID>/<PIX>/<sio path>	sio://aid/A000000062/03010C0D01/purse
sio://<reserved path>/<sio path>	sio:///standard/auth/holder/global/owner/biometric
Events	
Event root URIs:	
• event://<context path>	event:///transit/pos
• event://aid/<RID>/<PIX>	event://aid/a000000062/03010c0d01
• event://<reserved path>	event:///platform
event://<context path>/<event path>	event:///transit/pos/credited
event://aid/<RID>/<PIX>/<event path>	event://aid/A000000062/03010C0D01/debited
event://<reserved path>/<event path>	event:///platform/clock/resynced
Other Resources	
Resource type root URIs:	
• <scheme>://<context path>	file:///transit/pos
• <scheme>://aid/<RID>/<PIX>	file://aid/a000000062/03010c0d01
• <scheme>://<reserved path>	N/A
<scheme>://<context path>/<resource path>	file:///transit/pos/log/transactions.txt
<scheme>://aid/<RID>/<PIX>/<resource path>	file://aid/A000000062/03010C0D01/log.txt
<scheme>://<reserved path>/<resource path>	N/A

2.3.3 Handling of URIs

The Java Card Platform uses URIs for designating:

- applications and application resources, as well as platform and standard resources, see [Section 2.3.1, “Unified Naming Scheme” on page 2-5](#)
- the target endpoints of connections handled by the Generic Connection Framework (GCF), see the API for `javax.microedition.io.Connector`

A Java Card Platform implementation **MUST** support the URI syntax as defined by *RFC 3986: Uniform Resource Identifiers (URI): Generic Syntax* with the additional following amendments:

- **Default Normalization of URIs** - A Java Card Platform implementation **MUST** by default perform an enhanced syntax-based normalization in order to test URI strings for equivalence. See [“Default Normalization of URIs” on page 10](#). As a result, the followings apply:
 - **No default port assumption** - A network URI with an explicit port, where the port is the default for the scheme, is not equivalent to one where the port is omitted.
 - **No host resolution** - A network URI with a host, where the host is a domain name of a network host, is not equivalent to one where the host has been resolved to its IP address.
 - **No interpretation and no default value assumption for path segment parameters** - Path segments are considered opaque and are compared byte-for-byte.
- **Scheme-based and Protocol-based Normalization of Network URIs** - A Java Card Platform implementation **MUST** postpone any network scheme and protocol-dependent handling of a URI until the designated resource is being accessed or located. Scheme-based and protocol-based normalizations **MUST** only be performed when actually locating and controlling access to the network resource designated by the URI. See [“Scheme-based and Protocol-based Normalization of Network URIs” on page 11](#). As a result, applications and permission-based security policies should use network URIs in a scheme-based and protocol-based normalized form to reduce false negatives during security checks.
- **URI Patterns** - A Java Card Platform implementation **MUST** support the Java Card Platform-specific URI syntax extension for patterns that designate one or more resources. See [Section 2.3.3.2, “URI Patterns” on page 2-12](#).

2.3.3.1 Canonicalization of URIs

URIs can be absolute or relative. A URI which starts with a scheme (preceding a ‘:’) such as `sio`, `event` or `file`, is an absolute URI. Otherwise it is a relative URI or, more properly, a relative URI reference.

In order to properly interpret or compare URIs, the Java Card Platform MUST implement the following requirements:

- All URI matching operations MUST apply on canonicalized forms of the URIs. Therefore, URIs MUST first be resolved, then normalized. See [“Resolution of Relative URI References” on page 9](#) and [“Default Normalization of URIs” on page 10](#).

API and SPI methods MUST accept well-formed URIs (including those identifying applications or the platform) passed by applications but MUST only return or pass to applications such URIs in their canonical forms.

Resolution of Relative URI References

Relative URI references MUST be resolved as per *RFC 3986* against a base URI:

- The default base URI MUST be the current application’s root URI, for the scheme appropriate for the use. The current application’s root URI for a specific scheme MUST be constructed as follows:

`<scheme>:<application URI>/`

where `<scheme>` is the scheme, `<application URI>` is the canonical URI of the current application as determined from the current thread’s active namespace (see [Section 2.7.2.2, “Per-Thread Active Context and Active Namespace” on page 2-38](#)). Note that a `’/’` is appended to construct the default base URI.

- If the API or SPI explicitly states a base URI, the relative URI reference MUST be resolved against that base URI. For example, the `javacardx.io.FileConnection.open` method may be passed a relative file URI reference which must be resolved against the URI of the currently open file.

The following are examples of relative URI reference resolutions:

- Resolution of an absolute-path reference (a relative URI reference that starts with a single `’/’`):
 - In the context of application `///transit/pos` and for an SIO-based service use, `/transit/pos/ticketbook` would be resolved into `sio:///transit/pos/ticketbook` and `/standard/auth/holder/global/owner/fingerprint` would be resolved into `sio:///standard/auth/holder/global/owner/fingerprint`.
 - In the context of application `//aid/A0000000062/03010C0D01/` and for an event use, `/A0000000062/03010C0D01/debited` would be resolved into `event://aid/A0000000062/03010C0D01/debited` and `/platform/clock/resynced` would be resolved into `event://aid/platform/clock/resynced`.
- Resolution of a relative-path reference (a relative URI reference that does not start with a `’/’`):

- In the context of application `///transit/pos` and for an SIO-based service use, `ticketbook` would be resolved into `sio:///transit/pos/ticketbook`.
- In the context of application `//aid/A000000062/03010C0D01/` and for an event use, `debited` would be resolved into `event://aid/A000000062/03010C0D01/debited`.

Default Normalization of URIs

By default, all URIs handled by the Java Card RE, API and SPI MUST be normalized as per *RFC 3986's Syntax-Based Normalization* which includes:

- **Case Normalization** - Any hexadecimal digits within a percent-encoding triplet (for example, `%3a` versus `%3A`) are case-insensitive and therefore MUST be normalized to lower case. The scheme and host components of a URI are considered case-insensitive, and therefore MUST be normalized to lower case.
- **Percent-Encoding Normalization** - Any percent-encode octets in a URI MUST be normalized by decoding any percent-encoded octet that corresponds to an unreserved character.
- **Path Segment Normalization** - The complete path segments `"/"` and `"/."` are intended only for use within relative URI references and MUST be removed as part of the resolution process described in [“Resolution of Relative URI References” on page 9](#). Path segments are otherwise considered opaque.

In addition, URIs MUST be further normalized as follows:

- **IP Address Normalization** - In addition to the case normalization of the host, an IPv6 addresses MUST be normalized by converting the address to the “preferred form” `x:x:x:x:x:x:x:x` and removing leading zeros from each individual field (see *RFC 2373, IP Version 6 Addressing Architecture*).
- **Port Specification Normalization** - Any port number stated in the port specification of a URI MUST be normalized by removing any leading zeros.
- **Registry-based Authority Normalization** - Platform and standard URIs (URIs with root path component `/platform` and `/standard` respectively) that use the default registry-based URI authority, and those that use the `aid` registry-based URI authority are considered equivalent. Platform and standard URIs MUST be normalized to use the default registry-based URI authority.
- **Application and Platform-identifying URI Normalization** -
 - A URI identifying the platform MUST be normalized to `///platform`.
 - A URI identifying the web application MUST be normalized to `//<context path>` where `<context path>` starts with a `'/'` and does not include any path segment parameters.

- A URI identifying an applet application **MUST** be normalized to `//aid/<RID>/<PIX>` or `//aid/<RID>/-` where `<RID>` and `<PIX>` are upper-case hexadecimal digit strings, `<RID>` is 10 digits of length and `<PIX>` has an even length of at the most 22 digits, including any necessary leading zeros.

These normalization rules **MUST** also apply to the root path of an application resource URI which identifies the application.

For example,

`event:///transit/pos/foo/bar/../../../../ticketbook/overdraft` must be rewritten as `event:///transit/pos/ticketbook/overdraft`. And `HTTP://java.SUN.com:080/products/javacard` must be rewritten `http://java.sun.com:80/products/javacard`. Also, `event://aid/platform/clock/resynced` must be rewritten `event:///platform/clock/resynced`.

Scheme-based and Protocol-based Normalization of Network URIs

Scheme-based and protocol-based normalizations **MUST** only be performed when actually locating and controlling access to network resources designated by the URI. In addition to syntax-based normalization, network URIs **MUST** be normalized as per RFC 3986's *Scheme-based Normalization* and *Protocol-based Normalization* which include depending on the schemes:

- **Port Specification Normalization** - A URI with a port specification that corresponds to the default port for the scheme **MUST** be normalized with the default port omitted.
- **Host Normalization** - A URI with a host (a host name or an IP address) **MUST** be normalized with the complete domain name for the host and **MUST** only default to use its IP address when it can't be resolved. Reverse name resolution is on a best effort basis. To preserve GCF semantics on server connections (for example, `ssl://:369` or `ssl://`), no default is defined for the host subcomponent of a URI. An empty host subcomponent **MUST** not be normalized with the "localhost" name, even if a port is specified. And conversely, a host subcomponent with the "localhost" name **MUST** not be normalized to an empty host subcomponent, even if no port is specified.
- **Empty Path Normalization** - A URI with an empty path **MUST** be normalized with a path of `"/`". This normalization **MUST** not be applied to GCF connection schemes for which the path is irrelevant such as for the schemes `ssl`, `socket` and `datagram`.

If an optional protocol is not supported, a Java Card Platform implementation **MAY** not perform this normalization for that protocol.

Scheme-based and Protocol-based Normalization of URIs **MUST** only be performed by the methods of the `javax.microedition.io.Connector` class.

2.3.3.2 URI Patterns

URI patterns are used to designate one or more resources. URI patterns are of the following types:

- exact patterns, which designate exactly one resource,
- path-prefix patterns ending with a `"/*`, which designate a collection of resources,
- application-generic-path patterns starting with `"/~"`, which designate a resource or collection of resources in the current application's namespace,
- server-based authority patterns whose DNS hostname, if one is specified, starts with `"*."` or equals `"*"` or whose port number, if one is specified, equals `"*"`,
- registry-based authority patterns whose URI authority name equals `"*"`,
- or any combination of the above.

Note – The root of the namespace designated by the path prefix URI pattern may or may not be itself included in the set of resources designated, depending on the context. For example, in the context of permissions, the path prefix pattern `sio:///transit/pos/*` does not include `sio:///transit/pos`. The path prefix pattern `file:///transit/pos/logs/*` does not include the directory designated by `file:///transit/pos/logs`, see [Section 6.2.1.2, “URI-named Permissions” on page 6-7](#). In the context of servlet mapping the path prefix pattern `/transit/pos/*` includes the web resource `/transit/pos`, see the “Mapping Requests to Servlets” chapter of the *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition*.

TABLE 2-2 shows the different URI pattern types supported on the Java Card Platform.

TABLE 2-2 URI Pattern Types

Pattern Type	Description	Example(s)
Exact pattern	Designates a single resource	/transit/pos sio:///transit/pos/ticketbook event://aid/A000000062/03010C0D01/debited file:///transit/pos/ticketbook/log/transactions.txt http://java.sun.com:80/products/javacard
Path-prefix pattern	Designates a set of resources: all the resources with the same scheme and authority, and with a URI path component starting with the same path prefix.	/* /transit/* sio:///transit/pos/* event://aid/A000000062/03010C0D01/* file:///transit/pos/ticketbook/log/* http://java.sun.com:80/products/*
Application-generic-path pattern	Designates a set of resources: all the resources with the same scheme and authority, and with a path relative to their owning application's root URI matching exactly the path suffix (the path after the '~').	sio:///~/ticketbook event://aid~/debited file:///~/ticketbook/log/transactions.txt
Registry-based authority pattern	Designates a set of resources in a web application or applet application: all the resources with the same scheme and URI path component.	sio:///~/~/ticketbook event:///~/~/debited file:///~/~/ticketbook/log/transactions.txt
Server-based authority pattern	Designates a set of resources: all the resources with the same scheme and URI path component, and with a hostname and port number matching the hostname pattern and port number pattern.	http://*.sun.com:80/products/javacard https://java.sun.com:*/products/javacard https://*:*/*products/javacard ssl://java.sun.com:* socket://*:5555

2.4 Context Isolation Basics

This section describes the basic requirements of the firewall-enforced context isolation mechanism originally defined on the Java Card Platform, Classic Edition as applicable to the Java Card Platform, Connected Edition. [Section 6.9, “Context Isolation Enhancements” on page 6-65](#) describes the additional features of the context isolation facility on the Java Card Platform, Connected Edition, v 3.0.1.

Any implementation of the Java Card RE MUST support isolation of contexts and applications. Isolation means that one application cannot access the components, fields and methods of objects of an application in another context unless the other application explicitly provides an interface for access. The Java Card RE mechanisms for context isolation and object sharing are detailed in the following sections.

The basic minimum protection requirements of contexts and firewalls are described in this document because the features are not transparent to the application developer. Developers must be aware of the behavior of objects, APIs, and exceptions related to the firewall.

Java Card RE implementers are free to implement additional security mechanisms beyond those of the application firewall, but these mechanisms MUST be transparent to applications and MUST NOT change the externally visible operation of the VM.

2.4.1 Application Firewall

The *application firewall* within Java Card technology is runtime-enforced protection and is separate from the Java technology protections. The Java programming language protections still apply to Java Card applications. The Java programming language ensures that strong typing and protection attributes are enforced.

Application firewalls are always enforced in the Java Card VM. They allow the VM to automatically perform additional security checks at runtime.

2.4.1.1 Firewall Protection

The Java Card technology-based firewall (Java Card firewall) provides protection against the most frequently anticipated security concern: developer mistakes and design oversights that might allow sensitive data to be “leaked” to another application. An application may be able to obtain an object reference from a publicly accessible location. However, if the object is owned by an application protected by its own firewall, the requesting application must satisfy certain access rules before it can use the reference to access the object.

The firewall also provides protection against incorrect code. If incorrect code is loaded onto a card, the firewall still protects objects from being accessed by this code.

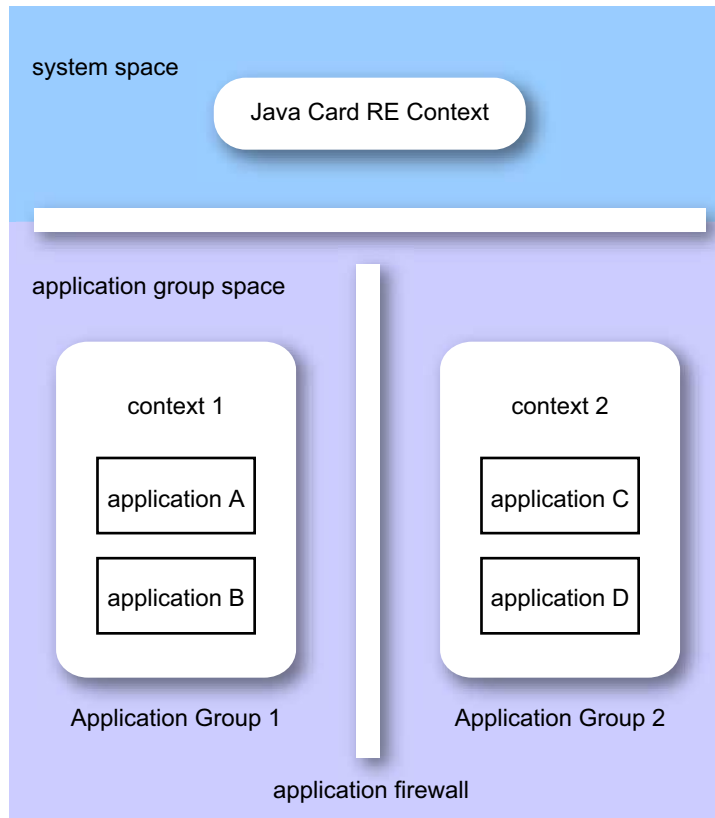
2.4.1.2 Contexts and Context Switching

Firewalls essentially partition the Java Card platform's object system into separate protected object spaces called *group contexts*. These are illustrated in [FIGURE 2-1](#). The firewall is the boundary between one group context and another. The Java Card RE MUST allocate and manage a *group context* for each *application group* containing applications¹. An application group corresponds to a set of application modules (see [Section 8.4.1, "Application Module Distribution Format" on page 8-4](#)) which have a trust relationship among themselves as determined by the application provider and the card manager.

All application instances within a single application group MUST share the same context. There is no firewall between individual application instances within the same application group. That is, an application instance can freely access objects belonging to another application instance that resides in the same application group.

1. Note that a library is not assigned a separate context. Objects from a library belong to the context of the creating application instance.

FIGURE 2-1 Contexts Within the Java Card Platform’s Object System



In addition, the Java Card RE MUST maintain its own *Java Card RE context*. This context is much like the context of an application, but it MUST have special system privileges so that it can perform operations that are denied to contexts of applications. For example, access from the Java Card RE context to any application instance’s context must be allowed, but the converse, access from an application instance’s context to the Java Card RE context, must be prohibited by the firewall.

Active Contexts in the VM

At any point in time, there may be multiple *active contexts* within the VM, one per thread. The *currently active context* of a thread is described in [Section 2.7.2.2, “Per-Thread Active Context and Active Namespace”](#) on page 2-38. This can be either the Java Card RE context or an application’s context. All bytecodes that access objects MUST be checked at *runtime* against the currently active context in order to determine if the access is allowed. A `java.lang.SecurityException` MUST be thrown when an access is disallowed.

Context Switching in the VM

If access is allowed, the VM determines if a *context switch* is required. A context switch occurs when certain well-defined conditions, as described in [Section 2.4.2.8, “Class and Object Access Behavior” on page 2-28](#), are met during the execution of invoke-type bytecodes. For example, a context switch may be caused by an attempt to access a shareable object that belongs to an application instance that resides in a different group context. The result of a context switch is a new currently active context.

During a context switch, the previous context and object owner information **MUST** be pushed on the VM thread stack, a new context becomes the currently active context for that thread, and the invoked method **MUST** execute in this new context. Upon exit from that method the VM **MUST** perform a restoring context switch. The original context (of the caller of the method) is popped from the stack and is restored as the currently active context for that thread. Context switches can be nested. The maximum depth depends on the amount of VM stack space available.

Most method invocations in Java Card technology do not cause a context switch. For example, a context switch is unnecessary when an attempt is made to access an object that belongs to an application instance that resides in the same group context. Context switches only occur during invocation of and return from certain methods, as well as during exception exits from those methods (see [Section 2.4.2.8, “Class and Object Access Behavior” on page 2-28](#)).

Further details of contexts and context switching are provided in later sections of this chapter.

2.4.1.3 Object Ownership

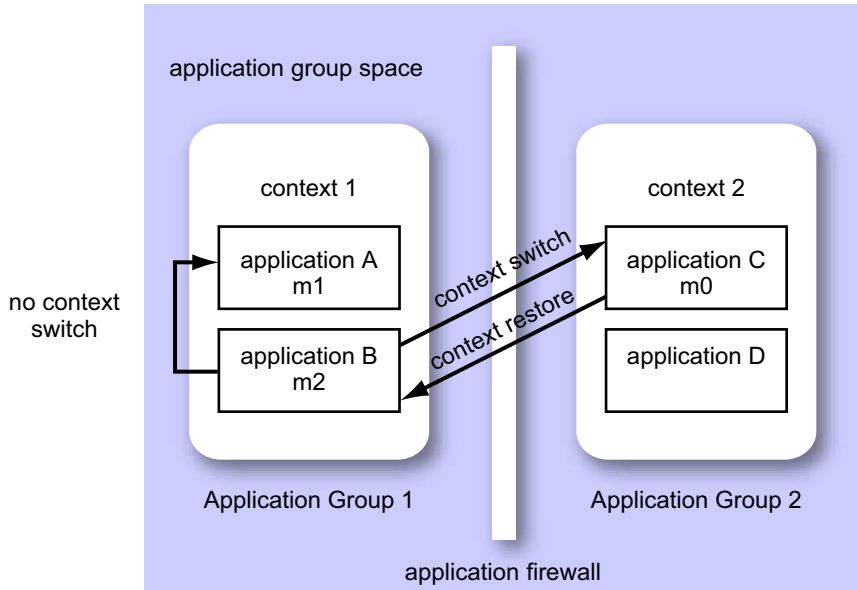
Any given object in the Java Card platform’s object space has a context and an owner associated with it. When a new object is created, it **MUST** be associated with the currently active context, but the object **MUST** be *owned* by the application instance within the currently active context when the object is instantiated. An object can be owned by an application instance, or by the Java Card RE.

Following are the combined rules of context and object ownership within the firewall:

- Every application instance belongs to a group context. All application instances from the application group **MUST** belong to the same group context.
- Every object **MUST** be owned by an application instance (or the Java Card RE). An application instance is identified by its application URI name (and AID for applet applications). When executing in an instance method of an object (or a static class method called from within), the object’s owner **MUST** be in the currently active context.

FIGURE 2-2 below illustrates method transitions from method m1 in application instance A to method m2 in application instance B, within the same group context, context 1, which MUST NOT trigger a context switch. The transition from method m2 in application instance B to a method m0 in application instance C MUST trigger a context switch since application C is in a different group context, context 2.

FIGURE 2-2 Context Switching and Object Access



2.4.1.4 Object Access

In general, an object MUST only be *accessed* by its owning context, that is, when the owning group context is the currently active context. The firewall MUST prevent an object from being accessed by another application in a different group context.

In implementation terms, each time an object is accessed, the object's owner group context is compared to the currently active context. If these do not match, the access is not performed and a `SecurityException` MUST be thrown.

An object is accessed when one of the following bytecodes is executed using the object's reference:

```
getfield, putfield, invokevirtual, invokeinterface,
athrow, <T>aload, <T>astore, arraylength, checkcast, instanceof,
monitorenter
```

<T> refers to the various types of array bytecodes, such as iaload and aastore.

This list MUST include any special or optimized forms of these bytecodes implemented in the Java Card VM for performance or other optimization purposes.

2.4.1.5 Transient Objects and Contexts

Transient objects of `CLEAR_ON_RESET` type also behave like all other objects in that they MUST be accessed only when the currently active context is the object's owning context (the currently active context at the time when the object was created).

Transient objects of `CLEAR_ON_DESELECT` type MUST only be created or accessed by an applet application, when the currently active context is the context of the currently selected applet. If any of the `makeTransient` factory methods are called to create a `CLEAR_ON_DESELECT` type transient object, from a web application or when the currently active context is not the group context of the currently selected applet (even if the attempting group context is that of an active applet instance on another logical channel), the method MUST throw a

`java.lang.SystemException` with reason code of `ILLEGAL_TRANSIENT`. If an attempt is made to access a transient object of `CLEAR_ON_DESELECT` type from a web application or when the currently active context is not the context of the currently selected applet (even if the attempting context is that of an active applet instance on another logical channel), the Java Card RE MUST throw a `java.lang.SecurityException`.

Applications that are part of the same application group share the same group context. Every application instance from the application group shares all its object instances with all other instances from the same application group. This includes transient objects of both `CLEAR_ON_RESET` type and `CLEAR_ON_DESELECT` type owned by these application instances.

The transient objects of `CLEAR_ON_DESELECT` type owned by any applet instance in the same application group MUST be accessible when any of the applet instances is the currently selected applet.

2.4.1.6 Static Fields and Methods

Instances of classes (objects) are owned by application instances within group contexts. Classes themselves are not.

Runtime context checks MUST NOT be performed when a class static field is accessed. A context switch MUST NOT occur when a static method or static initializer method (<clinit>) is invoked. Similarly, `invokespecial` MUST NOT cause a context switch.

Public static fields and public static methods **MUST** be accessible from any context: Static methods execute in the same context as their caller.

Objects referenced in static fields are just regular objects. They are owned by whomever created them and standard firewall access rules apply. If it is necessary to share them across multiple contexts, these objects need to be *shareable interface objects* (SIOs), see [Section 2.4.2.4, “Shareable Interfaces” on page 2-25](#), or transferable across the firewall, see [Section 6.9.3.1, “Ownership of Objects Created by Transferred Objects” on page 6-66](#).

The conventional Java technology protections are still enforced for static fields and methods. In addition, when applications are loaded, the card management facility verifies that each attempt to link to an external static field or method is permitted by the code isolation and package access control checks, see [Section 6.7, “Code Isolation” on page 6-57](#) and [Section 6.8, “Package Access Control” on page 6-63](#).

2.4.2 Object Access Across Contexts

The application firewall confines an applications actions to its designated context. To enable applications to interact with each other and with the Java Card RE, some well-defined yet secure mechanisms are provided so one context can access an object belonging to another context.

These mechanisms are provided in the Java Card API and are discussed in the following sections:

- [Section 2.4.2.1, “Java Card RE Entry Point Objects” on page 2-20](#)
- [Section 2.4.2.2, “Global Arrays” on page 2-24](#)
- [Section 2.4.2.3, “Java Card RE Privileges” on page 2-25](#)
- [Section 2.4.2.4, “Shareable Interfaces” on page 2-25](#)
- [Section 6.9.3.1, “Ownership of Objects Created by Transferred Objects” on page 6-66](#)

2.4.2.1 Java Card RE Entry Point Objects

To allow for applications to request system services performed by privileged Java Card RE routines, the Java Card API designates *Java Card RE entry point objects*. These are objects owned by the Java Card RE context, but they are flagged as containing entry point methods. Access to these classes and objects is controlled by the `javacardx.spi.framework.JCREPermission` permissions configured for these objects, see [Section 6.2.5.2, “Context-switch-triggered Security Checks” on page 6-20](#).

The firewall protects the fields and components of these objects from access by applications. The entry point designation allows the methods of these objects to be invoked from any context. When that occurs, a context switch to the Java Card RE context is performed. These methods are the gateways through which applications request privileged Java Card RE system services. The requested service **MUST** be performed by the entry point method only after verifying that the method parameters are within bounds and all objects passed in as parameters are accessible from the caller's context.

Following are the two categories of Java Card RE entry point objects:

- Temporary Java Card RE entry point objects

Like all Java Card RE entry point objects, methods of temporary Java Card RE entry point objects can be invoked from any context. However, references to these objects cannot be stored in class variables, instance variables or array components. The Java Card RE **MUST** detect and restrict attempts to store references to these objects as part of the firewall functionality to prevent unauthorized reuse. In addition, the Java Card RE **MUST** allow access to these objects only by threads that are passed these objects as parameters via applet application entry points.

The APDU object is an example of a temporary Java Card RE entry point object.

- Permanent Java Card RE entry point objects

Like all Java Card RE entry point objects, methods of permanent Java Card RE entry point objects can be invoked from any context. Additionally, references to these objects can be stored and freely re-used.

Java Card RE owned AID instances are examples of permanent Java Card RE entry point objects.

The Java Card RE **MUST** perform the following tasks:

- Determine what privileged services are provided to applications
- Define classes containing the entry point methods for those services
- Configure the access control permissions for these classes, see [Section 6.2.5.2, “Context-switch-triggered Security Checks”](#) on page 6-20.
- Create one or more object instances of those classes
- Designate those instances as Java Card RE entry point objects
- Designate Java Card RE entry point objects as temporary or permanent
- Make references to those objects available to applications as needed

Note – Only the *methods* of these objects are accessible through the firewall. The fields of these objects are still protected by the firewall and can only be accessed by the Java Card RE context.

The Java Card Platform, Connected Edition, MUST implement Java Card RE-owned instances of the classes listed in [TABLE 2-3](#) and [TABLE 2-5](#) as *permanent Java Card RE entry point objects*.

TABLE 2-3 Classic Set of Permanent Java Card RE EPO

Java Card RE-owned instances of the <code>javacard.framework.AID</code>

Java Card RE-owned instances of the classes listed in [TABLE 2-4](#) are *temporary Java Card RE entry point objects*.

TABLE 2-4 Classic Set of Temporary Java Card RE EPO

Java Card RE-owned instances of <code>javacard.framework.APDU</code>
--

Java Card RE-owned instances of the classes listed in [TABLE 2-5](#) are *permanent Java Card RE entry point objects* (EPO).

TABLE 2-5 Extended Set of Permanent Java Card RE EPO

Java Card RE-owned singleton instance of <code>java.lang.Runtime</code>
Java Card RE-owned instances of <code>java.security.Permission</code>
Java Card RE-owned singleton instance of <code>javacardx.facilities.EventRegistry</code>
Java Card RE-owned singleton instance of <code>javacardx.facilities.ServiceRegistry</code>
Java Card RE-owned singleton instance of <code>javacardx.facilities.TaskRegistry</code>
The Java Card RE-owned instances of <code>java.io.PrintStream</code> assigned to <code>java.lang.System.err</code> and <code>java.lang.System.out</code>

Java Card RE-owned instances of the classes listed in [TABLE 2-6](#) are *permanent Java Card RE entry point objects* (EPO).

TABLE 2-6 Card Management Set of Permanent Java Card RE EPO

Java Card RE-owned instances of <code>javacardx.spi.security.CryptoProvider</code>
Java Card RE-owned instances of the classes that encapsulates the characteristics of protection domains and other card management objects such as application instances and deployment units

The set of Permanent Java Card RE EPO classes MAY be extended by standards bodies provided the classes respect the contract of Java Card RE EPO requirements defined above.

Note – Some of the classes listed in [TABLE 2-3](#), [TABLE 2-5](#) and [TABLE 2-6](#) may be instantiated by an application. Such application-owned instances are not subject to *Java Card RE entry point objects* access rules.

Ownership of Object Passed as Parameter to Java Card RE Entry Point Objects

Before performing a context switch to the Java Card RE context upon a call to a Java Card RE entry point object, the Java Card RE **MUST** ensure that the parameters of the call are owned by either the calling application or its group context, are not owned by any context (such as for implicitly transferable objects), or are Java Card RE entry point objects. Otherwise, a security exception **MUST** be thrown.

Ownership of Object Returned by Java Card RE Entry Point Objects

Before performing a context switch back to the context of the calling application upon returning from a call to a Java Card RE entry point object, the Java Card RE **MUST** make the object to be returned (if any) accessible to the calling application by assigning the ownership of the object to the calling application's context, unless otherwise stated in the Java Card API, that is unless:

- the object to be returned is not owned by any context (such as an implicitly transferable object or a global array),
- or the object to be returned is a Java Card RE entry point object,
- or the object to be returned is owned by another application, such as an SIO.

Especially, the `java.lang.Object.toString` method of a Java Card RE entry point object **MUST** return a `java.lang.String` instance owned by the calling application, or not owned by any context if the `String` instance corresponds to an interned `String`. The `java.lang.Object.getClass` method of a Java Card RE entry point object **MUST** return a `java.lang.Class` instance that is not owned by any context since `Class` objects are impolitely transferable objects.

Note – The `toString` method of a Java Card RE entry point object **SHOULD** not convey any sensitive information and **SHOULD**, in a production environment, just default to the implementation defined for `Object.toString`.

Ownership of Errors and Exceptions Thrown by the Java Card RE

The `java.lang.Throwable` class and subclasses thereof are implicitly transferable classes, see [Section 7.2.1, “Transferable Classes” on page 7-3](#). Therefore, all errors and exceptions thrown by the Java Card RE are implicitly transferable objects, meaning direct instances of implicitly transferable *throwable* classes.

Note – Exceptions and errors that are designated Java Card RE entry point objects on the Java Card Platform, Classic Edition by standards body specifications **MUST** be designated implicitly transferable objects on the Java Card Platform, Connected Edition.

2.4.2.2 Global Arrays

Some applet environment array objects require that they be accessible from any context. The firewall would ordinarily prevent these objects from being used in a flexible manner. The Java Card VM allows an array object to be designated as *global*.

Global arrays have properties similar to those of temporary Java Card RE entry point objects. These objects are owned by the Java Card RE context, but can be accessed from any context. However, references to these objects cannot be stored in class variables, instance variables or array components. The Java Card RE **MUST** detect and restrict attempts to store references to these objects as part of the firewall functionality to prevent unauthorized reuse. In addition, the Java Card RE **MUST** allow access to these objects only by threads that are passed these objects as parameters via applet application entry points.

The Java Card Platform, Connected Edition, **MUST** implement Java Card RE-owned instance(s) of these array objects listed in [TABLE 2-7](#) as *global array objects*.

Note – Arrays that are designated as global arrays on the Java Card Platform, Classic Edition by standards body specifications **MUST** also be designated as global arrays and accessible only to classic applications and extended applet applications on the Java Card Platform, Connected Edition.

TABLE 2-7 Classic Set of Global Arrays

APDU buffer byte array

`bArray`, the byte array parameter to the `Applet.install` method

Note – Global array objects listed in [TABLE 2-7](#) are instances of byte arrays which may also be instantiated by an application. Such application-owned instances of byte arrays are not subject to *global array objects* access rules.

Note – Because of the global status of the APDU buffer, it **MUST** be cleared to zeroes whenever an applet is selected, before the applet container accepts a new APDU command. This is to prevent an applet’s potentially sensitive data from being “leaked” to another applet via the global APDU buffer. The APDU buffer can be accessed from a shared interface object context and is suitable for passing data across different contexts. The applet is responsible for protecting secret data that may be accessed from the APDU buffer.

2.4.2.3 Java Card RE Privileges

The Java Card RE context has a special privilege. The Java Card RE context can access fields, methods and components of any object on the card including `CLEAR_ON_DESELECT` transient objects owned by the currently selected application.

Note – Although the Java Card RE context is allowed to invoke any method through the firewall, it **MUST** only invoke the designated lifecycle and entry point methods of the application model of the application, and methods on the objects passed to the API as parameters.

The Java Card RE context is the currently active context when the VM begins running after a card reset. The Java Card RE context is the “root” context and is always either the currently active context or the bottom context saved on the stack of every Java Card RE-managed thread.

2.4.2.4 Shareable Interfaces

Shareable interfaces are a feature in the Java Card API to enable application interaction. A shareable interface defines a set of shared interface methods. These interface methods can be invoked from one context even if the object implementing them is owned by an application in another context.

In this specification, an object instance of a class implementing a shareable interface is called a *shareable interface object* (SIO).

To the owning context, the SIO is a normal object whose fields and methods can be accessed. To any other context, the SIO is an instance of the shareable interface, and only the methods defined in the shareable interface are accessible. All other fields and methods of the SIO MUST be protected by the firewall.

Shareable interfaces provide a secure mechanism for inter-application communication, as described in the following sections.

2.4.2.5 Determining the Previous Context

When an application calls `JCSystem.getPreviousContextAID`, the Java Card RE MUST return the instance AID of the applet application instance active at the time of the last context switch, if the previous application is an applet application.

When an application calls `JCSystem.getPreviousURI`, the Java Card RE MUST return the URI of the most recently active application namespace.

If an application calls `JCSystem.getPreviousContextAID` or `JCSystem.getPreviousURI`, from the `run` method of a newly started application-managed thread, the Java Card RE MUST return `null`.

Java Card RE Context

The Java Card RE context neither has an AID, nor a namespace URI. If an application calls the `getPreviousContextAID` method when the context of the application was entered directly from the Java Card RE context, this method MUST return `null`.

If an application calls the `getPreviousURI` method when the context of the application was entered via an application lifecycle or entry point method from the Java Card RE context, this method MUST return `null`.

2.4.2.6 Shareable Interface Details

A shareable interface is simply one that extends (either directly or indirectly) the *tagging* interface `javacard.framework.Shareable`.

Java Card API Shareable Interface

Interfaces extending the `Shareable` tagging interface have this special property: Calls to the interface methods take place across Java Card platform's application firewall boundary by means of a context switch.

The Shareable interface serves to identify all shared objects. Any object that needs to be shared through the application firewall MUST directly or indirectly implement this interface. Only those methods specified in a shareable interface are available through the firewall.

Implementation classes can implement any number of shareable interfaces and can extend other shareable implementation classes.

A shareable interface simply defines a set of service methods. A service provider class implements the shareable interface and provides implementations for each of the service methods of the interface. A service client class accesses the services by obtaining an object reference, casting it to the shareable interface type, and invoking the service methods of the interface.

The shareable interfaces within the Java Card technology MUST have the following properties:

- When a method in a shareable interface is invoked, a context switch occurs to the context of the object's owner.
- When the method exits, the context of the caller is restored.
- Exception handling is enhanced so that the currently active context is correctly restored during the stack frame unwinding that occurs as an exception is thrown.

2.4.2.7 Obtaining Shareable Interface Objects

Inter-application communication is accomplished when a client application invokes a shareable interface method of an SIO belonging to a server application. The client application discovers and obtains the SIO from the server application using either the classic application mechanisms or the new service and event facilities of the Java Card Platform, Connected Edition.

Obtaining Shareable Interface Objects from a Web Application or an Extended Applet Application

The server application implements the `ServiceFactory` interface and registers its SIO service with the `ServiceRegistry`. The client application is able to lookup services via the `ServiceRegistry`. Or, the client application implements the `EventNotificationListener` interface and registers its listener with the `EventRegistry` to receive notification of SIO-based events fired by the server application. The server application creates an instance of the SIO-based Event class and fires the event through the `EventRegistry`. These mechanisms are described in [Section 7.3, “Shareable Interface Object-based Services”](#) on page 7-7.

Obtaining Shareable Interface Objects from a Classic Applet Application

The server classic applet application implements the `Applet.getShareableInterfaceObject` method. The `JCSys` class provides the `JCSys.getAppletShareableInterfaceObject` method to enable an client applet application to request SIO objects from the server classic applet application. These mechanisms are described in more detail in the “Applet Isolation and Object Sharing” chapter of the *Runtime Environment Specification, Java Card Platform, Classic Edition, v 3.0.1*.

2.4.2.8 Class and Object Access Behavior

A static class field is *accessed* when one of the following Java programming language bytecodes is executed:

`getstatic`, `putstatic`

An object is accessed when one of the following Java programming language bytecodes is executed using the object’s reference:

`getfield`, `putfield`, `invokevirtual`, `invokeinterface`, `athrow`,
`<T>aload`, `<T>astore`, `arraylength`, `checkcast`, `instanceof`,
`monitorenter`

`<T>` refers to the various types of array bytecodes, such as `iaload`, `aastore`, etc.

This list includes any special or optimized forms of these bytecodes implemented in the Java Card VM for performance or other optimization purposes.

Prior to performing the work of the bytecode as specified by the Java VM, the Java Card VM MUST perform an *access check* on the referenced object. If access is denied, a `java.lang.SecurityException` MUST be thrown.

The Java Card 3.0.1 Platform defines additional permission-based security checks during context switching to allow for a finer-grained access control during sharing across group contexts, see [Section 6.9.1, “Context Switches” on page 6-65](#).

The access checks performed by the Java Card VM depend on the type and owner of the referenced object, the bytecode, and the currently active context. They are described in the following sections.

Accessing Static Class Fields

Bytecodes:

`getstatic`, `putstatic`

- If the Java Card RE is the currently active context, access is allowed.
- Otherwise, if the bytecode is `putstatic` and the field being stored is a reference type and the reference being stored is a reference to a temporary Java Card RE entry point object or a global array, access is denied.
- Otherwise, access is allowed.

Accessing Array Objects

Bytecodes:

`<T>aload, <T>astore, arraylength, checkcast, instanceof`

- If the Java Card RE is the currently active context, access is allowed.
- Otherwise, if the bytecode is `aastore` and the component being stored is a reference type and the reference being stored is a reference to a temporary Java Card RE entry point object or a global array, access is denied.
- Otherwise, if the array is owned by an application in the currently active context, access is allowed.
- Otherwise, if the array is designated global, access is allowed.
- Otherwise, access is denied.

Accessing Class Instance Object Fields

Bytecodes:

`getfield, putfield`

- If the Java Card RE is the currently active context, access is allowed.
- Otherwise, if the bytecode is `putfield` and the field being stored is a reference type and the reference being stored is a reference to a temporary Java Card RE entry point object or a global array, access is denied.
- Otherwise, if the object is owned by an application in the currently active context, access is allowed.
- Otherwise, if the object is an implicitly transferable object, access is allowed.
- Otherwise, access is denied.

Accessing Class Instance Object Methods

Bytecodes:

`invokevirtual`

- If the object is owned by an application in the currently active context, access is allowed.
- Otherwise, if the object is designated a Java Card RE entry point object, access is allowed. Context is switched to the object owner's context (MUST be Java Card RE).
- Otherwise, if Java Card RE is the currently active context, access is allowed. Context is switched to the object owner's context.
- Otherwise, if the object is an implicitly transferable object, access is allowed.
- Otherwise, access is denied.

Accessing Standard Interface Methods

Bytecodes:

`invokeinterface`

- If the object is owned by an application in the currently active context, access is allowed.
- Otherwise, if the object is designated a Java Card RE entry point object, access is allowed. Context is switched to the object owner's context (MUST be Java Card RE).
- Otherwise, if the Java Card RE is the currently active context, access is allowed. Context is switched to the object owner's context.
- Otherwise, if the object is an implicitly transferable object, access is allowed².
- Otherwise, access is denied.

Accessing Shareable Interface Methods

Bytecodes:

`invokeinterface`

- If the object is owned by an application in the currently active context, access is allowed.
- Otherwise, if the object is owned by a non-multiselectable classic application instance that is not in the context of the currently selected application instance, and that is active on another logical channel, access is denied. See Section 4.2, "Multiselectable Applets" of the *Runtime Environment Specification, Java Card Platform, Classic Edition, v 3.0.1*.

2. Note though, that currently none of the implicitly transferable classes defined in this specification implements an interface.

- Otherwise, if the object's class implements a `Shareable` interface, and if the interface being invoked extends the `Shareable` interface, access is allowed. Context is switched to the object owner's context.
- Otherwise, if the Java Card RE is the currently active context, access is allowed. Context is switched to the object owner's context.
- Otherwise, access is denied.

Throwing Exception Objects

Bytecodes:

`athrow`

- If the object is owned by an application in the currently active context, access is allowed.
- Otherwise, if the Java Card RE is the currently active context, access is allowed.
- Otherwise, if the object is an implicitly transferable object, access is allowed.
- Otherwise, access is denied.

Accessing Classes

Bytecodes:

`checkcast, instanceof`

- If the object is owned by an application in the currently active context, access is allowed.
- Otherwise, if the object is designated a Java Card RE entry point object, access is allowed.
- Otherwise, if the Java Card RE is the currently active context, access is allowed.
- Otherwise, if the object is an implicitly transferable object, access is allowed.
- Otherwise, access is denied.

Accessing Standard Interfaces

Bytecodes:

`checkcast, instanceof`

- If the object is owned by an application in the currently active context, access is allowed.
- Otherwise, if the object is designated a Java Card RE entry point object, access is allowed.

- Otherwise, if the Java Card RE is the currently active context, access is allowed.
- Otherwise, if the object is an implicitly transferable object, access is allowed.
- Otherwise, access is denied.

Accessing Shareable Interfaces

Bytecodes:

`checkcast`, `instanceof`

- If the object is owned by an application in the currently active context, access is allowed.
- Otherwise, if the object's class implements a `Shareable` interface, and if the object is being cast into (`checkcast`) or is being verified as being an instance of (`instanceof`) an interface that extends the `Shareable` interface, access is allowed.
- Otherwise, if the Java Card RE is the currently active context, access is allowed.
- Otherwise, if the object is an implicitly transferable object, access is allowed.
- Otherwise, access is denied.

Accessing Array Object Methods

Note – The method access behavior of global arrays is identical to that of Java Card RE entry point objects.

Bytecodes:

`invokevirtual`

- If the array is owned by an application in the currently active context, access is allowed.
- Otherwise, if the array is designated a global array, access is allowed. Context is switched to the array owner's context (Java Card RE context).
- Otherwise, if Java Card RE is the currently active context, access is allowed. Context is switched to the array owner's context.
- Otherwise, access is denied.

Accessing Monitors for Objects

The Java Card Virtual Machine **MUST** restrict access to the monitors for objects owned by applications in other contexts.

Bytecodes:

`monitorenter`

- If the Java Card RE is the currently active context, access is allowed.
- Otherwise, if the object is owned by an application in the currently active context, access is allowed.
- Otherwise, if the object is an instance of `java.lang.Thread` owned by the Java Card RE, access is allowed.
- Otherwise, if the object is the *classic applet container mutex object* owned by the Java Card RE, access is allowed. See [Section 4.4.2, “SIO Synchronization Proxy Classes” on page 4-17](#).
- Otherwise, if the object is an implicitly transferable object, access is allowed.
- Otherwise, access is denied.

2.5 Inter-Application Communication Facilities Overview

On the Java Card Platform, Connected Edition, an application can communicate with another application using the following two inter-application communication facilities:

- **Services:** an application can publish *shareable interface object-based services* it wants to provide to other applications. This facility extends the classic shareable interface mechanism and allows for all application models, including classic applet applications, extended applet applications, and web applications to interact through shareable interface objects in a unified way.

This facility allows for:

- applications to define and dynamically register (and unregister) SIO-based services
- applications to lookup SIO-based services registered by other applications

This facility also ensures a seamless integration of classic Java Card runtime environment SIO registration and lookup.

To use this facility, component developers are required to implement SIO-based services and their associated factories.

See [Section 7.3, “Shareable Interface Object-based Services” on page 7-7](#) for details on inter-application communication facility.

- **Events:** the platform or an application can notify other applications of particular conditions. When these conditions occur, they are encapsulated in shareable interface objects called *events* and are passed for handling to objects called *event listeners* that have been registered for notification of these conditions. This facility builds on the inter-application communication facility and allows for web applications and applet applications to communicate asynchronously with each other through events.

This facility allows for:

- applications to dynamically register and unregister for notification of other applications' events and platform events
- applications to define and fire events

This facility cannot be used directly by classic applet applications.

To use this facility, component developers are required to implement SIO-based events and event listeners.

See [Section 7.4, “Events” on page 7-16](#) for details on the event notification facility.

2.6 Applications Not Activated Through a Container-managed Endpoint

The server connection endpoint over which HTTP requests and HTTP responses are received and sent, respectively, by the web container on behalf of applications it manages is called a *container-managed endpoint*. Similarly, the endpoint over which APDU commands and APDU responses are received and sent, respectively, by the applet application container on behalf of applications it manages is also called a *container-managed endpoint*. When an HTTP request or an APDU command is received over one of these container-managed endpoints, the targeted application is determined and the request or command is dispatched to that application for processing. This constitutes the primary mode of activation of an application on the Java Card Platform.

Developers may create applications that are not primarily activated through a container-managed endpoint. These applications may, for example, provide services to other on-card applications or provide network services to off-card applications, such as through an *application-managed server endpoint*. These applications typically are automatically started or restarted upon a platform reset and must execute in their own application-managed threads, see [Section 2.7, “Multithreading” on page 2-35](#).

An application developer may create such an application by registering a restartable task (see [Section 2.10, “Restartable Tasks” on page 2-55](#)), which will be automatically invoked or started, respectively, upon a platform reset.

To setup an application for automatic start, a web application developer may use the `contextInitialized` method of a `javax.servlet.ServletContextListener` object to register the task when the web application is instantiated, see [Section 3.2.3, “Application Instance Creation” on page 3-5](#). An extended applet application developer may similarly use the constructor or the `install` method of an applet for the same purpose, see [Section 4.2.3, “Application Instance Creation” on page 4-5](#).

2.7 Multithreading

The Java Card virtual machine supports multithreading and concurrent execution of applications, see *Java Card Virtual Machine Specification, Java Card Platform, v3.0.1, Connected Edition*. The Java Card API includes a Java SE subset of the `java.lang.Thread` API, which allows for an application to create and handle threads of control.

The Java Card Runtime Environment (RE) MUST provide the following multithreaded application programming environments:

- the web application environment
- the extended applet application environment

While the classic applet application programming environment does not support multithreading, it MUST still operate concurrently with the other two application environments.

The web application container, the extended applet application container and other facilities of the Java Card RE MAY use multiple threads, referred to in this specification as Java Card RE-managed threads, to invoke the *entry point methods* of applications to:

- concurrently manage the lifecycle of application components (such as applets, servlets, filters, and listeners),
- concurrently service incoming HTTP requests or process APDU commands,
- asynchronously notify of occurring events,
- execute background application tasks.

Web applications and extended applet applications must be thread-aware and must account for such concurrent processing. See [Chapter 3](#) and [Chapter 4](#) for more details.

Classic applet applications are not thread-aware; the classic applet application environment MUST, therefore, guarantee the thread-safety of classic applet applications.

Additionally, web applications and extended applet applications may themselves spawn new threads, referred to in this specification as application-managed threads.

The above-mentioned Java Card RE-managed and application-managed threads are the only threads to which application developers may be exposed. All these threads may be handled using the `java.lang.Thread` API. Some restrictions MUST apply when handling a thread based on:

- the ownership of the thread as per the context isolation principle, see [Section 2.7.4, “Thread Ownership” on page 2-40](#).
- the permissions granted as per the access control mechanism, see [Section 6.8, “Package Access Control” on page 6-63](#).

Some Java Card Platform implementations MAY rely internally on various additional threads. The management of these threads is implementation-dependent.

2.7.1 Thread Creation

The Java Card Runtime Environment MAY create a new thread of control or MAY reuse an already created thread of control to invoke an application’s *entry point method*, a lifecycle method, an event notification method or a task execution method.

In resource-constrained configurations and for performance reasons, a Java Card Platform implementation MAY limit the number of concurrent Java Card RE-managed threads. This may limit the number of HTTP requests and APDU commands the platform may handle concurrently. A Java Card Platform implementation MAY also rely on a pool of reusable threads. An application must, therefore, not rely on different `Thread` objects being used for each subsequent entry point method invocation.

Web applications and extended applet applications may create new threads. A Java Card Platform implementation MAY restrict the creation of application-managed threads to ensure that other components of the platform are not negatively impacted. An application must, therefore, account for such restrictions and must handle situations where the creation of a new thread results in a security exception being thrown.

To create a new thread of control, an application must instantiate a `Thread` object, optionally set its initial priority, and run it by invoking its `start` method.

As per the context isolation principle, when a `Thread` object is instantiated, it MUST be assigned an owner context, see [Section 2.7.4, “Thread Ownership” on page 2-40](#).

The creation of threads by applications **MUST** be subject to access control. When a `Thread`'s constructor is invoked, a security check **MUST** ensure that the permission `javacardx.framework.JCRuntimePermission` with the target name `thread.create` is granted.

2.7.2 Thread Execution

An application may alter the scheduling of a thread's execution it owns by doing one of the following:

- Changing the thread's priority by calling the thread's `setPriority` method.
- Cause the current thread to relinquish control by calling the `Thread.sleep` or `Thread.yield` methods.

Thread priority changes by applications **MUST** be subject to access control. When the `Thread.setPriority` method is invoked, a security check **MUST** ensure that the permission `javacardx.framework.JCRuntimePermission` with the target name `thread.modify` is granted. See [Section 2.7.4, “Thread Ownership” on page 2-40](#) for details on other restrictions that apply.

2.7.2.1 Clock-dependent Thread Operations

The `Thread.sleep(long millis)` method **MUST** put the currently executing thread to sleep for *at least* the specified number of milliseconds. There is no guarantee the thread will wake up in exactly the specified time. Other thread scheduling can interfere, as can the granularity and accuracy of the system clock, among other factors. If the thread is interrupted while it is sleeping, then an `InterruptedException` **MUST** be thrown.

A Java Card platform implementation with a coarse system clock granularity and limited system clock accuracy **MAY** take a conservative approach to guarantee that a call to the `sleep` method does not return before the specified number of milliseconds, unless interrupted, such as by using an upper-bound conservative estimate.

After a platform reset, a thread that was sleeping at the time of the card tear or power off is terminated as any other thread. See [Section 2.7.3, “Thread Interruption and Termination” on page 2-40](#) for more details.

2.7.2.2 Per-Thread Active Context and Active Namespace

All objects MUST be associated with an *owner context*, which corresponds to a *group context*, as well as an *application owner identifier* (application URI or AID). Refer to [Section 2.4, “Context Isolation Basics” on page 2-14](#) for more details on context isolation and context switching.

For each thread of control, distinct *active context* and *active namespace* references MUST be maintained:

- A thread’s *active context* is the *currently active context* and MUST be set to the owner context of the object whose method is being executed. The currently active context does not change when a static method or a method of a context-less object is executed, see [Section 2.4.1.2, “Contexts and Context Switching” on page 2-15](#) and [Section 6.9.3, “Ownership of Transferable Objects” on page 6-66](#), respectively.
- A thread’s *currently active namespace* MUST correspond to the application owner identifier of the active context set upon entry into the group context.

As a method of an object is executed the *currently active context* is set to the owner context of that object. The currently active context does not change when a static method or a method of a context-less object is executed, see [Section 7.2, “Object Ownership Transfer Mechanism” on page 7-3](#).

Each object created by a thread MUST be bound to the thread’s currently active context.

Note – A thread’s currently active context is distinct from the owner context of the Thread object itself.

The active context and namespace references of a newly started thread (as per a call to the `Thread.start` method) MUST be set to the currently active context and namespace of its parent thread.

The legacy API’s `javacard.framework.JCSystem.getAID` method is applicable to applet applications (both classic and extended) to obtain the AID (application identifier) of the owner of the calling object. This method MUST return `null` if the owner application identifier cannot be expressed as an AID object, that is if the application is a web application.

A call to the `javacardx.framework.JCSystem.getURI` method MUST return the currently active namespace, meaning the application URI, associated with the current thread.

A call to the `javacardx.framework.JCSystem.getPreviousURI` method MUST return the previously active namespace, meaning the application URI, that was associated with the current thread before it entered the current group context. This method MUST ignore any interposed Java Card RE context.

A call to the `javacardx.framework.JCSystem.getClientURI` method MUST return the namespace, meaning application URI, of the client of a shareable interface object invoked through one of its shareable interface methods. The client of a shareable object is defined as follows:

- If the shareable interface method was called by another application from a group context distinct from that of the server application (the owner of the SIO) or if no shareable interface method was called, the client corresponds to the previously active application as determined by a call to the `JCSystem.getPreviousURI` method.
- If the shareable interface method was called by another application from within the same group context as that of the server application (the owner of the SIO), the client corresponds to the currently active application as determined by a call to the `JCSystem.getURI` method.

A call to the `javacardx.framework.JCSystem.getServerURI(Shareable)` method MUST return the namespace, meaning application URI, that will get associated with the current thread if that thread enters another group context of the shareable interface object passed as the argument.

Caution – The `getURI` and `getAID` methods differ in that the first one returns the application URI currently associated with the current thread, meaning the currently active namespace, while the second one returns the application identifier of the owner of the calling object. An extended applet, invoked through its `process` method, calling these two methods may get the same application identifier only if it is currently executing in an object that it owns. If the object is owned by a different applet in the same group context, the two application identifiers may be different.

2.7.2.3 Thread Security

Each application's group context is bound to a set of permissions--a *protection domain*--which determines the protected functions an application is permitted to access (Section 6.1, "Security Policy" on page 6-1). At any time during its execution, a thread keeps track of the currently active context. The set of permissions that must be enforced during a particular method's invocation MUST be the set of permissions associated with the active context of the currently executing thread:

- After a context switch was performed from a client application's group context to a server application's group context, the set of permissions to be enforced MUST be the set of permissions associated with the server application's group context. That context corresponds to the currently active context.
- After a context switch was performed from an application's group context to the Java Card RE context, the set of permissions to be enforced MUST be the set of permissions associated with that application's group context. That context does not correspond to the currently active context but to the previously active context.

2.7.3 Thread Interruption and Termination

A thread may terminate in one of the two following ways:

- the thread's run method returns normally
- the thread's run method returns abruptly due to an uncaught exception

A Java Card RE-managed thread MAY not terminate when returning from the application's entry point method it is executing. A Java Card Platform implementation MAY rely on a pool of reusable Java Card RE-managed threads. Such threads will typically not terminate when returning from the application's entry point method it is executing.

Application-managed threads can be interrupted by calling the thread's `interrupt` method. Interruption of threads by applications MUST be subject to access control. When the `Thread.interrupt` method is invoked, a security check MUST ensure that the permission `javacardx.framework.JCRuntimePermission` with the target name `thread.modify` is granted. See [Section 2.7.4, "Thread Ownership" on page 2-40](#) for details other restrictions that apply.

After a card reset, both Java Card RE and application threads are terminated. Persistent `Thread` objects, meaning `Thread` objects reachable from a root of persistence as per the persistence by reachability model, MUST be in the "terminated" state and their associated thread of control MUST NOT be restarted.

2.7.4 Thread Ownership

As per the context isolation principle (see [Section 2.4, "Context Isolation Basics" on page 2-14](#)), when a `Thread` object is instantiated, it MUST be assigned an owner context, according to the following rules:

- Threads created by an application MUST be owned by that application.
- Threads created by the Java Card RE MUST be owned by the Java Card RE. This includes the web application container, the extended applet application container and other facilities of the Java Card RE to invoke an application's entry point method, such as a lifecycle method, an event notification method or a task execution method.

This imposes certain restrictions on what an application can do on a thread, depending on whether it is the owner of the thread or if the thread is owned by the Java Card RE or another application in a different group context.

In all cases, the current application MUST be allowed to invoke the static methods of the `Thread` class and MUST, therefore, be allowed to perform any of the following operations:

- Obtain a reference to the currently executing thread object by calling the `Thread.currentThread` method.
- Cause the currently executing thread to relinquish control by calling the `Thread.sleep` or `Thread.yield` methods.
- Obtain the current number of active threads in the Java Card VM by calling the `Thread.activeCount` method.

If the thread is owned by the Java Card RE or another application in a different group context, the current application **MUST NOT** be allowed to invoke any of the thread's instance methods and, therefore, **MUST NOT** be allowed to perform any of the following operations:

- Obtain or change the thread's priority by calling the thread's `getPriority` or `setPriority` method.
- Wait for the thread to die by calling the thread's `join` method.
- Interrupt the thread by calling the thread's `interrupt` method.
- Obtain a string representation of the thread by calling the thread's `toString` method.
- Test if the thread is alive by calling the thread's `isAlive` method.
- Invoke the thread's `run` and `start` methods.

As per the context isolation principle (see [Section 2.3, “Unified Naming and Dedicated Application Namespaces” on page 2-5](#)), any attempt to call a `Thread` instance method on any thread not owned by the current group context **MUST** result in a security exception being thrown. This especially applies to the HTTP request, APDU command and event notification threads, which are always owned by the Java Card RE and which an application may retrieve by calling the `Thread.currentThread` method.

These restrictions **MUST** apply in addition to the permission-enforced access control restrictions applicable to the currently executing thread. These restrictions must be taken into account by all applications regardless of what the initial entry point method invoked was, whether it was a lifecycle method, a shareable interface method, an event notification method or an application task run method.

These restrictions must be taken into account by extension libraries, which cannot easily discriminate between threads that are owned by the calling application and those that are not, by properly handling the security exceptions that may be thrown.

The firewall **MUST NOT** prevent any application from storing a reference to a thread, whether it is one of the threads it created, a Java Card RE-owned thread, or another application's thread. However, an application may only actively manage threads it owns.

2.7.5 Thread Safety of API Classes

The following Java Card API classes and methods are required to be thread-safe:

- All static methods of API and SPI classes MUST be thread-safe.
- All instance methods of permanent Java Card RE entry point objects (see [Section 2.4.2.1, “Java Card RE Entry Point Objects” on page 2-20](#)) MUST be thread-safe.
- The thread safety of other classes and methods must be explicitly handled by application and library developers when objects can be concurrently accessed by multiple threads.

Note – Temporary Java Card RE entry point objects (see [Section 2.4.2.1, “Java Card RE Entry Point Objects” on page 2-20](#)) are not required to be thread-safe because they MUST only be used within a single thread at a time. References to such objects cannot be passed to other threads because the firewall prevents storing references to such objects in any field variable or array component. See [Section 4.3.4.1, “Thread Safety” on page 4-14](#).

2.8 Persistence

Code and data persistence across card tear (or reset) and power up is key to the special nature of smart card devices and the Java Card platform. The Java Card Virtual Machine and application code persist across card tear (or reset) and card power up. Objects may be made persistent as well, if they are reachable from the roots of persistence.

2.8.1 Memory Store and Object Store Terminology

The term *volatile memory* is used to indicate that the memory is not expected to retain its contents between card tear and power up events or across a reset event on the smart card device. For the purposes of the *Runtime Environment 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.

The term *non-volatile memory* is used to indicate that the memory is expected to retain its contents between card tear and power up events or across a reset event on the smart card device. For the purposes of the *Runtime Environment 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, EEPROM and Flash memory. The *Runtime Environment 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.

A *volatile object* is 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 **MUST** be garbage collected on card tear (or reset). A Java Card conformant implementation **SHOULD** store the object in volatile memory but need not guarantee it due to the limited availability of volatile memory store in smart card devices. The volatile object may, therefore, be physically stored in volatile memory or non-volatile memory or some combination of the two types of memory store.

A *persistent object* is an object that **MUST** be stored in non-volatile store. This type of object is intended to be a long-lived object. A persistent object **MUST** retain its contents between card tear and power up events or across a reset event.

All objects, volatile objects and persistent objects, are garbage collected when no longer referenced from other objects. Garbage collection on volatile objects is typically initiated automatically as needed. Garbage collection on all objects - especially persistent objects - **MAY** be initiated on demand via the `java.lang.System.gc()`, `java.lang.Runtime.gc()` or `javacard.framework.JCSystem.requestObjectDeletion()` methods.

2.8.2 Persistence By Reachability Principle

Every object created by executing the `new`, `newarray`, `anewarray` or `multianewarray` bytecode **MUST** be a volatile object. The newly created object continues to be a volatile object as long as it is referenced only from other volatile objects or the stack or by *reachability disrupting objects* (described in [Section 2.8.2.1, “Reachability Disrupting Objects” on page 2-44](#)). If a volatile object is referenced by a persistent object, which is not a reachability disrupting object, or by a root of persistence such as a static reference field, the volatile object **MUST** be promoted and becomes a persistent object. If the newly promoted persistent object is not a reachability disrupting object, it **MUST** promote all the volatile objects it references to become persistent objects. Each of these newly promoted persistent objects, which is not a reachability disrupting object, **MUST** in turn, recursively, promote the volatile objects it references.

Thus, an object is maintained as a volatile object or persistent object according to its reachability³ from another object that acts as its root of persistence. The root of persistence for an application depends on the application model it implements.

2.8.2.1 Reachability Disrupting Objects

The reachability graph is disrupted by some special objects. If a volatile object is referenced by these special objects, promotion to a persistent object MUST NOT be triggered. These special objects are:

- Transient array object of type `Object` - This special array object is created by using the `javacard.framework.JCSystem.makeTransientObjectArray()` method. The components of the transient array object MUST be stored in volatile memory store, even when the transient array object is promoted to become a persistent object. The reference components of the array MUST be cleared to `null` upon a card tear (or card reset); the reference components of the array MUST also be cleared to `null` upon applet deselection event if the array is of the `CLEAR_ON_DESELECT` type. Detailed behavior of these objects is described in Chapter 5 of the *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*.
- Instances of `TransientReference` class - An instance of this special class is created, as usual, as a volatile object by executing the new bytecode. It may be promoted to become a persistent object just like any other volatile object. This special object internally encapsulates a generic object reference field that can be accessed via its `get` and `set` methods. The internal reference field itself MUST be stored in volatile memory store. The internal reference field MUST be cleared to `null` upon a card tear (or card reset) event.

These special objects may themselves be volatile objects or persistent objects but have the same disrupting effect on the reachability graph. These reachability disrupting objects are, therefore, used by an application to prevent the promotion of volatile objects that are short lived or require frequent updates.

Note – Being referenced by a reachability disrupting object does not prevent a volatile object from being promoted to become a persistent object via other promotion triggering references.

2.8.3 Roots of Persistence

All static fields are stored in non-volatile store and act as roots of persistence.

3. Reachability from a source object to a destination object is defined as whether or not there is a path from the source object to the destination object that follows the object references.

In addition, each of the application models supported by the Java Card 3 platform Connected Edition has its own model-specific roots of persistence:

- The web application model - An instance implementing the `javax.servlet.ServletContext` interface class acts as the root of persistence for a web application. Detailed requirements are described in [Section 3.2.9, “Container-managed Object Lifetime and Persistence”](#) on page 3-11.
- The APDU application model - Instances of the `javacard.framework.Applet` class act as the roots of persistence for an APDU-based application. Detailed requirements are described in [Section 4.2.8, “Container-managed Object Lifetime and Persistence”](#) on page 4-7.

Each application model is associated with its own application container. The application container is anchored by a Java Card RE owned persistent object. The application container **MUST** manage the persistence of the instantiated applications and their roots of persistence. The application container **MUST** manage the persistence of all container-managed objects by depending on the persistence by reachability principle. The persistence requirement of a container-managed object is based on the characteristics of the object itself, including its life span, frequency of updates, reuse requirements and security.

If a container-managed volatile object is promoted inadvertently or maliciously to become a persistent object by the application, the container **MAY** create a new volatile object to replace the promoted persistent object, provided the lifetime characteristics of the object are not violated.

A container-managed volatile object **MAY** be reused by the application container over several request handling cycles and sessions, provided the object is reused only by application(s) within the same group context.

All application-managed objects that need to be persistent must be referenced directly or indirectly by the roots of persistence of their application model.

2.9 Transaction Facility

The Transaction Facility enables an application to complete a single logical operation on application data atomically, consistently and durably within a transaction. The transaction facility provides atomicity, which ensures that updates to data either all occur, or none occur. Consistency allows the application to establish a consistent state before the start and after the end of the transaction. The transaction facility provides durability, which ensures that when the transaction is successfully completed, the updates are committed.

The Java Card Platform, Connected Edition supports a Transaction Facility that extends the “Transactions and Atomicity” subsystem described in Chapter 7 of *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*. The Transaction facility associates each transaction with a commit buffer. The commit buffer records all the updates that have not yet been committed. The commit buffer is used to rollback updates when the transaction does not complete successfully.

The Transaction Facility on the Java Card Platform, Connected Edition provides the following feature extensions:

- Support for multiple concurrent transactions - more than one transaction may be ongoing at the same time
- Support for nested transactions - a sub-transaction within an ongoing transaction may be initiated and may complete independently before the original transaction. The success or failure of the sub-transaction does not directly affect the original transaction. The success or failure of the original transaction does not directly affect the nested transaction
- Better programmer control and program audit of transaction durations - a method is annotated to explicitly declare its transactional behavior

2.9.1 Atomicity

Atomicity ensures that either all the updates to application state are completed or none are. It avoids the possibility of partially updated states.

2.9.1.1 Single Item Atomicity

The Java Card platform guarantees that any update to a single static class field or a single field of a persistent object, or a single component of an array will be atomic. That is, if the smart card loses power during the update of a data element that **MUST** be preserved across card tear and power up, that data element **MUST** be restored to its previous value.

Some methods also guarantee atomicity for block updates of multiple data elements. For example, the atomicity of the `javacard.framework.Util.arrayCopy` method guarantees that either all bytes are correctly copied or else the destination array is restored to its previous byte values.

Applications might not require atomicity for array updates. The `javacard.framework.Util.arrayCopyNonAtomic` method and other similar methods are provided for this purpose. These methods do not use the transaction commit buffer even when called with a transaction in progress.

2.9.1.2 Multiple Updates

An application might need to atomically update several different fields or array components in several different objects. The Java Card platform provides a transaction facility to ensure that either all updates take place correctly and consistently, or else all fields or components are restored to their previous values. Fields and array components of both volatile and persistent objects are protected by the transaction facility.

A transaction is defined with an explicit starting point and ending point during which updates are collectively performed atomically. During the period of the transaction, between start and end (or start and abort), all updates to fields of objects and components of arrays (except for the components of transient arrays, global arrays and the fields of `TransientReference` objects) are tracked in a commit buffer. The updates are conditional and only committed when the end of the transaction is reached. However, accessing an updated field or component even before being committed, returns the updated state⁴. If the transaction is aborted before the end of the transaction is reached, all the updated data are reverted back to their state at the start of the transaction, using the data in the commit buffers.

Note that when a transaction is in progress, updates to all fields, including fields and components of volatile objects, fields and components of persistent objects and static fields, are tracked in a commit buffer. Aborting a transaction reverts both type of objects to their states at the start of the transaction.

2.9.2 Transaction Demarcation

Runtime Environment Specification, Java Card Platform, v3.0.1, Connected Edition uses annotations metadata to mark an entire method for a transaction demarcation. A method annotated with a `TransactionType` annotation declares an explicit transaction behavior upon entry into the method which is applicable until normal return from the method. If the method throws an uncaught exception, the transaction is aborted.

The `TransactionType` annotation is defined in the package `javacardx.framework`. The value of the `TransactionType` annotation is given by the enum `TransactionTypeValue`. The possible enum values are:

- `MANDATORY`
- `REQUIRED`
- `REQUIRES_NEW`
- `SUPPORTS`

4. The Java Card transaction facility does not support the *isolation* property. Partially updated state may be accessed while a transaction is in progress.

- NOT_SUPPORTED
- NEVER

The `TransactionType` annotation may be specified on a class as well as on a method, but not on a constructor (See [Section 2.9.2.7, “Constructors and Static Initializers” on page 2-52](#)). A class level annotation sets the default for all methods on the class. A method level annotation overrides the default. If a class does not specify the `TransactionType` annotation, it is assumed to be `SUPPORTS`.

The `TransactionType` annotation, when specified on an interface or an interface method, has just an informative purpose and simply describes the transactional behavior that is expected of classes which implement the interface. The platform **MUST** only enforce the `TransactionType` annotation (or default) specified by the classes which implement the interface.

Note – All platform implementations of interfaces specified in *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition* **MUST** implement the `TransactionType` annotation behavior specified by the interfaces.

The `TransactionType` annotation allows the application programmer to specify the six different transaction behaviors on a method, `MANDATORY`, `REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`, `NOT_SUPPORTED` and `NEVER` using their corresponding enum values.

The following code shows annotation usage for transaction tagging. The `TicketBookSIO` class does not explicitly use the `TransactionType` annotation and is by default assigned the `SUPPORTS` `TransactionType` annotation. The `setBalance()` method overrides the default transaction tag with a `REQUIRED` transaction tag. The `getBalance()` method retains the default `SUPPORTS` transaction tag.

CODE EXAMPLE 2-1 Transaction Demarcation Example

```
public class TicketBookSIO implements TicketBookSI {
...
    @TransactionType(REQUIRED) public void setBalance() {
        ...
    }
    ...
    public int getBalance() {
        ...
    }
...
}
```


New threads **MUST NOT** be associated with any transaction initially. A new transaction may be created based on the annotation of the `run()` method of the `Runnable` instance from which a thread is starting execution.

Java Card RE managed threads **MUST NOT** invoke application entry point methods, such as container-managed lifecycle methods or service entry point methods with a transaction in progress. A new transaction may be created based on the annotation of the entry point methods.

Note – The `Applet.install()` method is an exception to this rule. The `Applet.install()` method **MUST** be invoked with a transaction in progress.

Each `TransactionType` annotation value and its associated behavior is described in detail in the following sections.

2.9.2.1 `@TransactionType(REQUIRES_NEW)`

When a method is annotated with the `REQUIRES_NEW` `TransactionType` annotation, a new commit buffer, separate and distinct from any commit buffers previously in use, is created to track updates made by the application until either the transaction ends or is aborted. Upon normal exit from the method, the updates are committed and the caller's transaction mode is resumed. An exception exit results in the transaction being aborted.

2.9.2.2 `@TransactionType(REQUIRED)`

When a method is annotated with the `REQUIRED` `TransactionType` annotation, the commit buffer in use, if any, by the caller is used to continue logging updates during this method also. Upon normal exit from this method, the commit buffer logging resumes in the callers' method. An exception exit simply results in a control transfer back into the caller's transaction.

If the caller is not within a transaction, or if this method is being started in a new thread, a new commit buffer is created. Upon normal exit from the method, the updates are committed and the caller's transaction mode is resumed. An exception exit results in the transaction being aborted.

2.9.2.3 `@TransactionType(SUPPORTS)`

When a method is annotated with the `SUPPORTS` `TransactionType` annotation, the commit buffer in use, if any, by the caller is used to log updates in this method also. If the caller is not within a transaction, or if this method is being started in a new

thread, no commit buffer is created and updates are not tracked. Upon normal exit from the method, the commit buffer logging resumes in the caller's method. An exception exit simply results in a control transfer back into the caller's transaction.

Note – `SUPPORTS TransactionType` annotation MUST be the default transaction behavior.

Note – All classes of a classic applet application MUST be tagged with `SUPPORTS TransactionType` annotation⁵ to ensure backward compatibility with the Java Card v2.2.2 platform.

2.9.2.4 `@TransactionType(NOT_SUPPORTED)`

Upon entry into a method which is annotated with the `NOT_SUPPORTED TransactionType` annotation, any ongoing transaction is suspended until exit from the method. Upon normal exit from the method, the caller's transaction mode is resumed. An exception exit simply results in a control transfer back into the caller's transaction.

2.9.2.5 `@TransactionType(MANDATORY)`

If a method annotated with the `MANDATORY TransactionType` annotation is called when no transaction is in progress, The Java Card RE MUST throw a `TransactionException` with exception reason "Transaction Required". Otherwise, the behavior is exactly the same as a method tagged with the "REQUIRED" `TransactionType` annotation.

2.9.2.6 `@TransactionType(NEVER)`

If a method annotated with the `NEVER TransactionType` annotation is called when a transaction is in progress, The Java Card RE MUST throw a `TransactionException` with exception reason "Transaction Disallowed". Otherwise, the behavior is exactly the same as a method tagged with the `NOT_SUPPORTED TransactionType` annotation.

5. Off-card pre-processing tools which process classic applications - Converter and Normalizer- typically force the default annotation - `SUPPORTS TransactionType`.

Note – All platform implementations of interfaces and classes specified in *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition* that are annotated MANDATORY, REQUIRED, REQUIRES_NEW or SUPPORTS for `TransactionType` MUST have a transactional behavior that is coherent at the level of the interface they expose. Not all updates to their state (in particular to temporary computational state) need to be transacted provided it has no incidence on the interface-level transactional behavior of the class or interface.

Note – An implementation of an interface or class specified in *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition* that is annotated SUPPORTS for `TransactionType` MAY rely on the transaction state of the caller for maintaining consistent internal state only on methods that update state visible at the level of the interface they expose. For example, the `get` method of the `java.util.Hashtable` class and *getter* methods in general need not be called with a transaction in progress.

TABLE 2-8 below describes the possible transitions managed by the transaction facility upon entry into a method and exit from a method based on the transaction state of the calling method. The cells shown with a shaded background (and double asterisks **) have transactions in progress where conditional updates are tracked in the commit buffer.

TABLE 2-8 Transaction Facility Managed Transitions On Method Entry and Exit

			Method TransactionType Annotation					
		EVENT	REQUIRES_NEW	REQUIRED	SUPPORTS	NOT_SUPPORTED	MANDATORY	NEVER
C a l l i n g S t a t e	No transaction in progress	Enter method	Create new commit buffer **	Create new commit buffer **	No action	No action	Throw exception	No action
		Normal return	Commit updates **	Commit updates **	No action	No action	N/A	No action
		Exception return	Rollback updates in commit buffer **	Rollback updates in commit buffer **	No action	No action	N/A	No action
	Transaction in progress **	Enter method	Create new commit buffer **	Use caller's commit buffer **	Use caller's commit buffer **	Suspend transaction	Use caller's commit buffer **	Throw exception
		Normal return	Commit updates **	No action **	No action **	Resume transaction	No action	N/A
		Exception return	Rollback updates in commit buffer **	No action **	No action **	Resume transaction	No action	N/A

2.9.2.7 Constructors and Static Initializers

Constructors and Static Initializers do not support the TransactionType annotation. Constructors and Static Initializers MUST always execute with the NOT_SUPPORTED TransactionType behavior.

Note – If a tear occurs during the execution of a class' Static Initializers, the class MUST remain uninitialized and the Java Card Platform MUST reinitialize the class by invoking the Class Initializers as described in the *Java™ Virtual Machine Specification (Java Series), Second Edition*.

2.9.2.8 Classic Applet Applications

The Java Card RE MUST support the transaction behavior described in Chapter 7 of *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition* for classic applet applications.

Classic applet application code MUST run single-threaded, use exactly one commit buffer and encounter a maximum transaction depth of 1. They use the `beginTransaction()`, `commitTransaction()` and `abortTransaction()` APIs to control the demarcation of the transaction.

2.9.2.9 Crossover Thread Behavior

An extended applet application or a web application thread may enter classic applet application code via a shareable interface object method. All classic applet application methods MUST be tagged with the `SUPPORTS TransactionType` annotation. Hence, upon entry, if a transaction in progress, the transaction facility MUST continue using the caller's commit buffer. If the `beginTransaction()` method is now called, a `TransactionException` exception with reason code `TransactionException.ILLEGAL_USE` MUST be thrown; if `commitTransaction()` or `abortTransaction()` method is called, a `TransactionException` exception with reason code `TransactionException.NOT_IN_PROGRESS` MUST be thrown. If a classic applet application is entered with no transaction in progress, the `beginTransaction()`, `commitTransaction()` and `abortTransaction()` methods MUST behave as normal.

A classic applet application thread may enter an extended applet application or a web application code via a shareable interface object method. A transaction may be in progress, at the point of entry into the SIO method. In such a case, the transaction facility MUST enforce the transitions described previously based on `TransactionType` annotations of the methods upon entry and exit.

The `beginTransaction()`, `commitTransaction()`, `abortTransaction()`, `getTransactionDepth`, `getUnusedCommitCapacity()` and `getMaxCommitCapacity()` APIs throw exceptions when accessed from extended applet applications and web applications.

2.9.3 Overlapping Transaction Updates

When the same data element - object field, array component, or static field - is updated from two different transactions on concurrently executing threads - the resulting state of that element will either be the original state or a state written by

one or the other of the threads, depending on the order in which the threads execute and whether the transactions are committed or aborted. The developer is strongly encouraged to avoid such a programming pattern.

When updated from multiple transactions on the same thread (assuming no updates from other threads), the same data element - object field, array component, or static field - may also result in an unexpected state. When the same data element is updated in multiple transactions, each nested within another, the platform **MUST** ensure the following outcomes:

- when the transaction which is started first is aborted, data elements updated within that transaction **MUST** be restored to their state prior to the start of the transaction
- when the transaction which is started first is committed, the data elements updated within that transaction **MUST** be committed to the state of last update within the last committed transaction among all the transactions

2.9.4 Transient Arrays and TransientReference Objects

Array components of transient array objects and global array objects are not conditionally updated during a transaction. Updates to components of transient array objects and global array objects **MUST** be committed immediately.

The private reference field of a `TransientReference` object is not conditionally updated during a transaction. An update to the private reference field of a `TransientReference` object **MUST** be committed immediately.

2.9.5 Power-up After Card Tear

When a card tear or power loss occurs and the card is subsequently powered-up, all ongoing transactions **MUST** be aborted. All the commit buffers **MUST** be examined in the reverse order in which they were created, and all updates recorded in each of the commit buffers **MUST** be examined in the chronologically reverse order of updates to restore the corresponding object field, class field or array component.

2.9.6 Aborting A Transaction

When an exception is thrown by a transacted method with `REQUIRES_NEW TransactionType` annotation, the transaction **MUST** be aborted.

When an exception is thrown by a method with `REQUIRED` `TransactionType` annotation and for which a new transaction was started upon entry, that transaction **MUST** be aborted.

When the `abortTransaction()` API is called by a classic applet application, the transaction **MUST** be aborted.

Transaction abortion requires that the commit buffer be traversed to rollback all updates performed since the start of the method. This **MUST** be attempted only after the `finally` clause has executed.

Updates tracked in the commit buffer **MUST** be reversed starting with the chronologically last update and ending with the first on each recorded field or array component.

2.9.6.1 Accessing An “Aborted Object”

If a new object is created within a transaction, and the transaction is subsequently aborted, the newly created object may now be obsolete. But, since references to the object may be stored outside the aborted transaction, the object must be garbage collected only when no references to it remain.

2.10 Restartable Tasks

The Restartable Task facility provided by the Java Card Platform, Connected Edition allows for web applications and extended applet applications to register recurrent tasks that get executed in separate application threads and are automatically restarted after each platform reset.

Only web applications and extended applet applications can register tasks for recurrent execution. Classic applet applications cannot use this facility.

The `javacardx.facilities.TaskRegistry` class allows for applications to dynamically register and unregister recurrent tasks.

The `TaskRegistry` instance **MUST** be a singleton. It **MUST** be a permanent Java Card runtime environment entry point object (see [Section 2.4.2.1, “Java Card RE Entry Point Objects” on page 2-20](#)). It is retrieved by calling the `TaskRegistry.getTaskRegistry` method.

2.10.1 Tasks

Task objects **MUST** implement the `java.lang.Runnable` interface.

2.10.2 Task Registration

To create a recurrent task, an application must register a task object with the registry. The application must call the `TaskRegistry.register` method and provide a task object. The application may additionally specify if the task is to be executed immediately or upon the next platform reset.

Attempts by an application which previously registered a task to register the same task again **MUST** have no effect other than the currently executing task's thread, if any, to be interrupted if the task is only to be started after the next platform reset.

Attempting to register a task that is already registered by another application **MUST** result in a security exception being thrown.

Registration of tasks **MUST** be subject to access control. When the `TaskRegistry.register` method is invoked, a security check **MUST** ensure that the permission `javacardx.facilities.TaskRegistryPermission` with the target name `task.register` is granted.

Note – An application may register a task even though it is not granted the permission to create and start a new thread, see [Section 2.7.1, “Thread Creation” on page 2-36](#).

Applications may register tasks anytime during their lifetime.

2.10.3 Task Execution

Each registered task **MUST** be executed in its own separate thread.

Each registered task **MUST** be automatically restarted in a new thread when the platform is reset.

Tasks **MUST** be restarted after the application containers have been restarted but before any request or command gets dispatched to applications.

Tasks **MUST** be started only once during a card session. If a task finishes normally or abnormally it **MUST NOT** be restarted until the next platform reset.

2.10.3.1 Thread Ownership

All restartable task registry threads **MUST** be owned by the Java Card RE. See [Section 2.7.4, “Thread Ownership” on page 2-40](#) for more details on the restrictions that apply to Java Card RE-owned threads.

2.10.4 Task Unregistration

An application may unregister a task it previously registered at any time. To unregister a task, an application must provide the same task object that was registered.

The application must call the `TaskRegistry.unregister` method and provide a task object. The application may additionally specify if the task’s thread is to be interrupted first.

Attempting to unregister a task registered by another application **MUST** result in a security exception being thrown.

Unregistration of tasks **MUST** be subject to access control. When the `TaskRegistry.unregister` method is invoked, a security check **MUST** ensure that the permission `javacardx.facilities.TaskRegistryPermission` with the target name `task.unregister` is granted.

Applications may unregister tasks anytime during their lifetime. Attempting to unregister a task that is not - or no longer - registered has no effect.

2.10.5 Lifetime and Persistence of Tasks

A task **MUST** remain registered until it is removed from the registry by the application that registered it or when that application is deleted. See [Section 8.9, “Deletion of Application Instance” on page 8-33](#) for more details on application instance deletion.

The registry **MUST** ensure that task objects are persistent across card sessions. Therefore, applications do not have to hold on to references to these objects to ensure their persistence.

2.10.6 Thread Safety

The `TaskRegistry` implementation **MUST** be thread safe to account for concurrent registrations and unregistrations.

Tasks concurrently run in their own separate threads. Therefore, developers must explicitly deal with synchronization when tasks are accessing shared resources.

2.10.7 Per-Thread Active Context

Each thread is associated to a current active context and namespace. When entering the `run` method of a task, the thread's current active context and namespace **MUST** be changed to the owner context of the task. See [Section 2.7.2.2, “Per-Thread Active Context and Active Namespace”](#) on page 2-38.

Note – The current active namespace when executing the `run` method of task may not always correspond to the namespace of the application that registered the task as an application may register a task object owned by another application from the same group context. Note though, that the current active namespace will always correspond to the namespace of one of the applications from the same group context as that of the application which registered the task.

2.10.8 Transactional Behavior

The `TaskRegistry` singleton instance is a permanent Java Card RE entry point object and therefore **MUST NOT** participate in any application transaction. The `TaskRegistry` class **MUST** be annotated with the `NOT_SUPPORTED TransactionType` annotation, see [Section 2.9.2, “Transaction Demarcation”](#) on page 2-47. All registration operations **MUST** nevertheless be atomic and **MUST** ensure that the task registry is at all times consistent.

Web Application Environment

This chapter describes the interactions and dependencies between the platform, the web container and web applications:

- [Servlet Subset Overview](#)
- [Web Application Lifecycle](#)
- [Lifecycle and Entry Point Method Invocation](#)
- [Default Container Behavior and Default Servlet](#)
- [Secure Hosting of Web Applications](#)

3.1 Servlet Subset Overview

The *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition* is a subset of the Java Servlet Specification v2.4. In addition to features that have been subset out due to dependencies on APIs or features not supported by the Java Card Virtual Machine or intended for deployment in web containers that are JavaServer Pages™ (JSP™) technology enabled or part of a Java™ Platform, Enterprise Edition (Java EE) application server, the following features have been removed or adapted to the Java Card platform:

- Filters are only invoked under request from web clients, see [Section 3.2.7, “Request Dispatching”](#) on page 3-9.
- Client Certificate-based user authentication is not supported. The SSL certificate a web client uses to establish a secure connection with the web container cannot be used for container-managed user authentication.
- The `javacardx.servlet.request.X509Certificate` request attribute must be used to retrieve the client SSL certificate associated with a request. See [Section 3.5.3.1, “Retrieving a Connection’s Specific Security Characteristics”](#) on page 3-28.

- The web application's structured hierarchy of directories is accessible through a Generic Connection Framework (GCF)-based file access mechanism when it is stored on a file system. See [Section 3.2.3, "Application Instance Creation" on page 3-5](#).
- The web container does not provide a private temporary working directory for each servlet context, meaning web application.
- Usage of URL patterns for servlet mapping, filter mapping and security constraints web resource collection designation is restricted, see [Section 3.2.7, "Request Dispatching" on page 3-9](#).
- The default container behavior when a request to a web application cannot be mapped to a servlet of the web application is restricted, see [Section 3.4, "Default Container Behavior and Default Servlet" on page 3-19](#).
- The deployment of web applications disassociates the loading of applications from the instantiation of applications.

3.2 Web Application Lifecycle

Web application modules are the logical units of assembly of web applications. All the components of a web application MUST be assembled into a web application module. See [Section 8.4.1.1, "Web Application Module Distribution Format" on page 8-4](#) for details on the structure of web application modules.

When being deployed, a web application will typically go through the following lifecycle:

1. Loading of the web application module, see [Section 3.2.1, "Application Module Loading" on page 3-3](#).
2. Creation of an instance of the web application, see [Section 3.2.2, "Application Instance Identification" on page 3-4](#) and [Section 3.2.3, "Application Instance Creation" on page 3-5](#).
3. Execution of the web application instance:
 - Dispatch of requests, see [Section 3.2.7, "Request Dispatching" on page 3-9](#).
 - Dispatch of lifecycle events, see [Section 3.2.8, "Lifecycle Event Dispatch" on page 3-11](#).
 - Invocation of registered SIO-based services and event listeners, see [Section 3.3.2, "SIO, Event and Restartable Task Entry Point Methods" on page 3-16](#).
 - Execution of registered restartable tasks, see [Section 3.3.2, "SIO, Event and Restartable Task Entry Point Methods" on page 3-16](#).

4. Restart upon platform reset, see [Section 3.2.6, “Restart Upon Platform Reset” on page 3-9](#).
5. Deletion of the web application instance, see [Section 3.2.4, “Application Instance Deletion” on page 3-7](#).
6. Unloading of the web application module, see [Section 3.2.5, “Application Module Unloading” on page 3-8](#).

Some web application modules may contain only dynamic contents; others may contain only static content; others may contain both dynamic and static contents. All such configurations of web applications MUST be managed according to the lifecycle described above.

A web application may be instantiated many times, but there MUST NOT be more than one instance of the web application at any time, that is, multi-instantiation of a web application MUST NOT be supported.

Note – While the *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition* refers to the deployment of web applications as a single operation which combines loading the application and instantiating the application, on the Java Card Platform the deployment of web applications disassociates the loading of applications from the instantiation of applications.

3.2.1 Application Module Loading

When a web application module is being loaded, the following requirements MUST be implemented by the Java Card Platform:

- The description of the web application MUST be retrieved from its web application descriptor located at `WEB-INF/web.xml` and from the Java Card Platform-specific application descriptor located at `META-INF/javacard.xml`.
- The runtime configuration of the web application MUST be retrieved from the applicable runtime descriptor.
- Classes and resources¹ in the module’s `WEB-INF/classes` directory MUST be made available to the application class loader assigned for loading the module. See [Section 6.7.1, “Class Loader Delegation Hierarchy” on page 6-58](#) for more details on application class loaders.
- Consistency of the descriptors (especially references to platform resources) and between descriptors (especially cross-references), and consistency between descriptors and code (such as class name references) MUST be checked. Note that formal validation of these descriptors against their respective schema is not

1. Class-path resources, that is resources that can be retrieved by the class loader using the `Class.getResourceAsStream` method.

required on-card, see [Section 8.5, “Descriptor Formats” on page 8-10](#). The interpretation of the descriptors MUST be strict; especially, in a web application deployment descriptor, a servlet mapping string (other than an extension mapping string) not starting with a “/” MUST never be matched by a request URL because a servlet path must always start with a “/”. See the “Mapping Requests To Servlets” chapter of the *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition* for more details.

- The web application’s structured hierarchy of directories MUST be stored to be made available to the web container for servicing the application’s static content once the application has been instantiated.

Any error during loading, especially module format errors, descriptor format errors, descriptor and code consistency errors, MUST result in the loading of the application module to fail and the application module to be rejected.

Refer to [Section 8.6, “Loading Application Modules” on page 8-25](#) for the other generic requirements that MUST be implemented by the Java Card Platform when an application module is loaded.

3.2.2 Application Instance Identification

Every web application instance MUST be rooted at a specific path within the web container known as its *context path*. The context path of a web application instance determines the URL namespace of the contents (both static and dynamic) of the application. All requests whose URL start with that context path MUST be routed to that application instance.

Every web application instance is named with a relative URI - its *application URI* - that uses the *null* registry-based authority and whose path component corresponds exactly to its context path. This application URI defines the root of a dedicated namespace within which all its resources, not only web contents but also SIO-based services, events and files, MUST be named.

A *legal* context path MUST satisfy the following requirements:

- it MUST be normalized and MUST NOT contain any “.” or “..” path components,
- it MUST be absolute and start with a “/”,
- it MUST be unique (in a case-sensitive manner),
- it MUST NOT overlap other application namespaces,
- it MUST NOT overlap any of the `/platform` and `/standard` reserved namespaces.

See [Section 2.3, “Unified Naming and Dedicated Application Namespaces” on page 2-5](#) for details on the additional constraints on naming web applications.

Note – To maximize portability, a web application developer should name or refer to a web application’s resources both in code and descriptors using relative URIs.

3.2.3 Application Instance Creation

When a web application instance is being created, the following requirements MUST be implemented by the Java Card Platform:

1. An instance of the `ServletContext` interface MUST be created and rooted at a *legal* context path in the web container’s namespace. The context path MUST correspond to the application’s URI. This instance of the `ServletContext` MUST serve as the logical root of persistence for the application instance, see [Section 3.2.9, “Container-managed Object Lifetime and Persistence” on page 3-11](#).
2. The web application’s structured hierarchy of directories MUST be made available to the web container for servicing the application’s static content relatively to the assigned context path. The root of this hierarchy serves as the document root for files that are part of the application:
 - a. The `WEB-INF` and `META-INF` directories are not part of the public document tree of the application. A file contained in the `WEB-INF` and `META-INF` directories MUST NOT be served directly to a client by the web container.
 - b. All the files with the exception of those under the `WEB-INF/classes`, `WEB-INF/lib` (if present²) and `META-INF` directories and with the exception of the deployment descriptor `WEB-INF/web.xml` MUST be made available to the application’s code in the following ways:
 - i. These files MUST be made available to the application’s code using the `ServletContext.getResourceAsStream` method.
 - ii. These files MUST be made available to the application’s code using the `ServletContext.getRequestDispatcher` and `ServletRequest.getRequestDispatcher` methods.
 - iii. The `ServletContext.getRealPath` method MUST return the real path of these files on the card’s file system (if any is supported). A static resource served for the request URI `<context path><resource path>` MUST have a real path in the form of `<context path><resource path>`. If no file system is supported, the `ServletContext.getRealPath` method MUST return `null`. The real path of a resource file on the card’s file system may be built

2. Note that libraries placed under the `WEB-INF/lib` directory are not supported and that this directory must be ignored

into a file URL of the form `file://<real path>` and used to access the resource file with a GCF file connection. See [Chapter 9](#) for more details on file system support.

Note – The resource path *<resource path>* when passed as parameter to the `getRealPath` method must begin with a "/" and is interpreted as relative to the context root denoted by *<context path>*.

- c. The class-path resources in the module's `WEB-INF/classes` directory **MUST** be made available to the application's code using the `Class.getResourceAsStream` method.
3. The following steps, described in more detail in the "Web Applications" chapter of the *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition*, **MUST** be performed before any request can be dispatched to the web application instance:
 - a. Create an instance of each life cycle event listener described in the web application deployment descriptor. Notify context event listeners of the web context initialization event.
 - b. Create and initialize an instance of each filter described in the web application deployment descriptor. The container initializes the filter instance by calling its `init` method.
 - c. Create and initialize an instance of each *load-on-startup* servlet described in the web application deployment descriptor in the applicable order. The container initializes the servlet instance by calling its `init` method.

Any exception during this sequence **MUST** result in the application creation failing. See [Section 3.3.1.2, "Handling of Errors and Exceptions Outside of Request Processing"](#) on page 3-16.

4. The web container **MUST** listen and accept connections on an application-dedicated secure port, if required. This is in addition to listening and accepting connections on the default plain and secure ports. See [Section 3.5, "Secure Hosting of Web Applications"](#) on page 3-20.
5. The web container **MAY** start dispatching requests as per [Section 3.2.7, "Request Dispatching"](#) on page 3-9.

All files in the application's namespace **MUST** only be accessible to that application. An implementation **MAY** store and manage a web application's resources as live objects bound to that application's group context or **MAY** store and manage a web application's resources on a file system with access permissions restricted to that application, see [Chapter 9](#).

Write access to a web application's structured hierarchy of directories **MUST** be subject to access control. When a `javax.microedition.io.Connector.openXXX` method is invoked, a security check **MUST** ensure that the permission `javacardx.io.ConnectorPermission` with the target canonicalized file URI and the action `write` is granted.

Note – Because a web application may be instantiated subsequently multiple times from the same loaded module code, the developer must account for static fields and initializers not being reset or re-run at each instantiation. Similarly, if an application's static content was updated at runtime, such as by writing to a GCF file connection, the static content will not be restored to its initial state.

Refer to [Section 8.8, “Creation of Application Instances” on page 8-31](#) for the other generic requirements, such as assignment to the proper group context and protection domain, that **MUST** be implemented by the Java Card Platform when an application instance is created.

3.2.4 Application Instance Deletion

When a web application instance is being deleted, the following requirements **MUST** be implemented by the Java Card Platform:

1. The container **MUST** stop dispatching requests to the application instance through request dispatchers both from web clients on the default ports and the application-dedicated secure port, as well as from other web applications. Any requests subsequently received on these ports **MUST** be responded to with an HTTP status code 404 (Not Found). Any attempt to dispatch a request using a request dispatcher **MUST** fail.
2. The container **MUST** stop listening and accepting connections on the application-dedicated secure port and **MUST** close all such already open connections.
3. Once the application instance has finished servicing the requests currently being handled or after some container-defined timeout, the following steps, described in more details in the *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition*, **MUST** be performed:
 - a. All servlet instances **MUST** be removed from service. The container first calls the `destroy` method on a servlet to enable the servlet to release any resources and perform other cleanup operations. The `destroy` method is only called once all threads within the servlet's `service` method have exited or after a timeout period has passed.

- b. All filter instances **MUST** be removed from service. The container first calls the `destroy` method on a filter to enable the filter to release any resources and perform other cleanup operations. The `destroy` method is only called once all threads within the filter's `doFilter` method have exited or after a timeout period has passed.
 - c. The container **MUST** invalidate all HTTP sessions the application instance may participate in and session listeners **MUST** be notified of session invalidations in the applicable order.
 - d. Context listeners **MUST** be notified of context destruction in the applicable order.
4. All GCF connections open by the application instance, including network connections as well as file connections, **MUST** be closed.
5. When no container-managed thread is still executing the application's code through any of its entry point methods, the web application's `ServletContext` instance **MUST** be destroyed and the context path corresponding to its application URI **MUST** be released. The web application's structured hierarchy of directories **MUST** no longer be accessible to the web container for servicing the former application's static content.

The URI identifying a web application instance, meaning its context path, becomes unassigned once the application instance has been deleted. The context path **MUST** only be reassigned once the application instance is considered effectively deleted, [Section 8.9, “Deletion of Application Instance” on page 8-33](#).

Refer to [Section 8.9, “Deletion of Application Instance” on page 8-33](#) for the other generic requirements that **MUST** be implemented by the Java Card Platform when an application instance is deleted.

3.2.5 Application Module Unloading

When a web application module is being unloaded, the following requirements **MUST** be implemented by the Java Card Platform:

- Any active instance of the application **MUST** first have been deleted as per [Section 3.2.4, “Application Instance Deletion” on page 3-7](#).
- Classes and resources in the module's `WEB-INF/classes` directory **MUST** no longer be available to the application class loader that was assigned for loading the module.
- The web application's structured hierarchy of directories **MUST** be removed from storage.

Refer to [Section 8.10, “Unloading of Deployment Units”](#) on page 8-36 for the other generic requirements that MUST be implemented by the Java Card Platform when an application module is unloaded.

3.2.6 Restart Upon Platform Reset

After a platform reset, the Java Card Platform MUST perform the following steps before any request can be dispatched to a web application instance:

1. The web container MUST handle all volatile container-managed objects as described in [Section 3.2.9.4, “Behavior Upon Platform Reset”](#) on page 3-13 and MUST perform the following actions in sequence:
 - i. All active sessions are cleared - volatile HTTP session objects are garbage collected. Listeners registered to receive session destruction events are not notified.
 - ii. All pending requests and responses are cleared - volatile HTTP request and response objects are garbage collected. Listeners registered to receive request destruction events are not notified.

Refer to [Chapter 5](#) for the other generic requirements that MUST be implemented by the Java Card Platform after a platform reset.

3.2.7 Request Dispatching

When a web application instance has been created, the following requirements MUST be implemented by the Java Card Platform:

1. The web container MUST accept requests for the web application instance both from web clients on the default ports and the application-dedicated secure port, as applicable, and from other web applications. The web container MUST forward to the web application instance requests whose start of the URL matches the context path of the application instance.

Note – On the Java Card Platform, application URIs, and therefore context paths, MUST NOT overlap, see [Section 2.3, “Unified Naming and Dedicated Application Namespaces”](#) on page 2-5. Therefore, searching for the web application instance with the longest context path that matches the start of the request URL is always equivalent to searching for the first web application instance with the context path that matches at least the first path component of the request URL.

2. On requests received from web clients, the web container MUST implement the following requirements:
 - a. The security constraint whose web resource collection URL pattern and HTTP method match the request's URI and HTTP method MUST be enforced.

Note – On the Java Card Platform, web resources to which a security constraint applies MUST only be designated with a URL pattern to which a servlet has been mapped.

- i. The authorization constraints MUST be enforced. An authorization constraint establishes a requirement for authentication and names the authorization roles permitted to perform the constrained requests. See [Section 6.4.5, “Web Container-managed Authentication” on page 6-35](#) and [Section 6.3.1, “User Role-based Security” on page 6-23](#) for more details.
 - ii. The user data constraints MUST be enforced. A user constraint establishes requirements that the constrained requests be received over a protected transport layer connection. See [Section 3.5, “Secure Hosting of Web Applications” on page 3-20](#).
 - b. The filter chain and servlet or static content whose URL pattern match the request's URI MUST be invoked in the applicable order.

Note – On the Java Card Platform, web resources to which a filter applies MUST only be designated with a URL pattern to which a servlet has been mapped.

3. On requests received from other web applications through the request dispatcher, the web container MUST implement the following requirements:
 - a. The servlet whose URL pattern matches the request's URI MUST be invoked. See [Section 3.2.7.1, “Restriction On the Use of Request Dispatchers” on page 3-11](#) for details on the restrictions that apply.

Note – On the Java Card Platform, URL patterns to which servlets are mapped MUST NOT overlap.

3.2.7.1 Restriction On the Use of Request Dispatchers

A web application **MUST** be allowed to retrieve the servlet context of another application in the same group context using the `ServletContext.getContext` method, by passing the context path of that other application. The web application can then retrieve a request dispatcher from that other application's servlet context to issue a `forward` or `include` call to one of that other application's resources.

A web application **MUST NOT** be allowed to retrieve the servlet context of an application in another group context. The `ServletContext.getContext` method **MUST** return `null` in such a situation.

Note – Application developers and assemblers must be aware that declarative security constraints defined in the web application's deployment descriptor does not apply when a servlet uses the `RequestDispatcher` to invoke a static resource or servlet using a `forward` or an `include`.

3.2.8 Lifecycle Event Dispatch

The Java Card Platform does not mandate any Java Card-specific behavior on lifecycle event dispatching.

3.2.9 Container-managed Object Lifetime and Persistence

When a web application is instantiated, the web application container creates an instance of `ServletContext` (an instance of a Java Card RE defined class that implements the `javax.servlet.ServletContext` interface) to manage the newly instantiated web application. This instance **MUST** be promoted to become a persistent object by virtue of being reachable from a persistent web container managed object. This instance of the `ServletContext` serves as the logical root of persistence for the newly instantiated application. This instance of `ServletContext` is deleted only when the application is deleted.

When executing a web application, the container **MUST** instantiate container-managed objects, meaning objects created by the container as per its operating principles (see *Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition*). Container-managed objects include instances which define the application's business logic such as servlets (instances of application-defined classes that implement the `javax.servlet.Servlet` interface), filters (instances of application-defined classes that implement the `javax.servlet.Filter` interface),

and event listeners (instances of application-defined classes that implement the `java.util.EventListener` interface). Container-managed objects also include instances which are used to encapsulate HTTP sessions and HTTP messages, meaning session objects (instances of a Java Card RE defined class that implement the `javax.servlet.http.HttpSession` interface), request objects (instances of a Java Card RE defined class that implement the `javax.servlet.http.HttpServletRequest` interface) and response objects (instances of a Java Card RE defined class that implement the `javax.servlet.http.HttpServletResponse` interface). In addition, container-managed objects include event objects (instances of Java Card RE defined classes that implement the `java.util.EventObject` interface), which are created by the container when a particular condition needs to be handled.

All container-managed objects which are instances of classes defined by the application **MUST** be bound to the owner context of the application itself. All container-managed objects which are instances of classes or interfaces defined in the `javax.servlet` package and its sub-packages **MUST** be bound to the owner context of the application itself. All other container-managed objects which are instances of classes defined by the Java Card RE **MAY** be owned by the Java Card RE.

Specific lifetime and persistence requirements of some container-managed objects are described in the following sections. Any container-managed object which is not called out specifically **MAY** have implementation assigned lifetime and persistence characteristics.

3.2.9.1 Servlet, Filter and Listener Lifetime

The following container-managed objects **MUST** be persistent objects. They are created when the application is instantiated and can be garbage collected only when the application instance is deleted.

- **The servlet context:** The servlet context is created when the application is instantiated. This is the root of persistence for the application.
- **The filters and event listeners:** these objects are created when the application is instantiated. References to these objects are maintained directly or indirectly in the servlet context by the container.
- **The servlets configured in the web application deployment descriptor as load-on-startup:** these objects are created when the application is instantiated. References to these objects are maintained directly or indirectly in the servlet context by the container.

The following container-managed objects **MAY** be volatile objects:

- **The servlets not configured in the web application deployment descriptor as load-on-startup:** these objects are created and initialized when there is a request to be serviced. These objects MAY be reused by the container to handle subsequent requests.

3.2.9.2 Session Lifetime

The following container-managed objects MUST be volatile objects unless promoted to become a persistent object by the application:

- **The HTTP sessions:** these objects are created when a new interaction session is started between an off-card application and the card. References to these objects are maintained by the servlet container until the session times out or until card tear (or reset).

Note – Container-managed volatile session objects SHOULD be referenced via reachability disrupting objects from the application's servlet context object or the container object itself.

3.2.9.3 Request, Response and Event Object Lifetime

The following container-managed objects MUST be volatile objects unless promoted to become a persistent object by the application:

- **The requests and responses:** these objects are created when new requests need to be serviced and are maintained while the requests are being serviced or until card tear (or reset).
- **The events:** these objects are created to encapsulate specific conditions that need to be handled. References to these objects are maintained while the events are being handled or until card tear (or reset).

3.2.9.4 Behavior Upon Platform Reset

Upon card reset, all volatile objects MUST be garbage collected. All references to volatile objects in reachability disrupting objects MUST be reset to `null`.

3.3 Lifecycle and Entry Point Method Invocation

3.3.1 Servlet, Filter and Listener Lifecycle Methods

Servlet, filter and listener lifecycle methods are invoked by the web container. These methods may also get called indirectly by an application itself in the following situations:

- When an event notification is triggered by an action performed by an application. An example of this is an application that adds a servlet context attribute and servlet context attribute listeners are notified of the addition in the very same thread which performed the triggering action.
- When a request is dispatched by a call to the request dispatcher by an application. The service method of the targeted servlet is invoked in the same thread which called the request dispatcher.

Note that as per the recommendation in [Section 3.3.4, “Multithreading Issues” on page 3-17](#), using request and response objects outside the scope of the request handling thread and the call of the `service` method is not advisable.

Any uncaught exception thrown from within a servlet, filter or listener lifecycle method is propagated back to the original caller. If the original caller is the web container, these exceptions must be handled specifically depending on the context of the invocation, see [Section 3.3.1.1, “Handling of Errors and Exceptions During Request Processing” on page 3-14](#) and [Section 3.3.1.2, “Handling of Errors and Exceptions Outside of Request Processing” on page 3-16](#).

3.3.1.1 Handling of Errors and Exceptions During Request Processing

When an exception is thrown by a servlet, filter or listener while processing a request, the container **MUST** take appropriate measures to clean up the request, depending on the error or exception. The type of request includes not only exceptions and errors thrown by the `Servlet.service` or `Filter.doFilter` methods, but also by other lifecycle methods that may be invoked to create the component to service a request, such as the `Servlet.init` method. The subsequent request to a component that generated such an error or exception, also depends on the error or exception:

- An `UnavailableException` signals that the servlet or filter is unable to handle requests either temporarily or permanently. A temporary unavailability is indicated at construction time by providing an estimate of how long the resource will be unavailable. A zero or negative estimate indicates that the component cannot make an estimate. No estimate indicates a permanent unavailability. The web container *MUST remove the component from service and release the component* and then handle the resource that caused such exceptions in one of the following ways:
 - In the case of a permanent unavailability, the resource *MUST* remain unavailable across the card session until the application instance is deleted and recreated. The container *MUST* respond to the request that initially caused the error and to any subsequent request for such permanently unavailable resource with an HTTP status code 404 (Not Found).
 - In the case of a determined temporary unavailability, the resource *MUST* remain unavailable at least for the expected duration. If the duration exceeds that of the card session, the container must do its best to track the expected duration across card sessions. The container *MUST* respond to the request that initially caused the error and to any subsequent request for such temporarily unavailable resource with an HTTP status code 503 (Service Unavailable), along with a `Retry-After` header indicating when the unavailability will terminate.
 - In the case of undetermined temporary unavailability, the resource *MUST* remain unavailable no more than the duration of the current card session, that is, at most until the next platform reset. The container *MUST* respond to the request that initially caused the error and to any subsequent request for such temporarily unavailable resource with an HTTP status code 503 (Service Unavailable).
 - The container *MAY* choose to ignore the distinction between a permanent and temporary unavailability and treat all `UnavailableExceptions` as permanent.
- Any other error or exception *MUST* only result in the container responding to the request that initially caused the error with an HTTP status code 500 (Internal Server Error) when the error page mechanism has not been configured to handle that error or exception. See the *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition* for more details on the error page mechanism.

3.3.1.2 Handling of Errors and Exceptions Outside of Request Processing

An error or exception may be thrown by a servlet, filter or listener lifecycle method outside the scope of processing a request by a specific component, such as during initialization at application start-up when creating listeners, filters and load-on-startup servlets, or during handling the notification of a request initialization. This MUST be considered an internal application error:

- If the application is being instantiated, the container MUST abort the application instance creation.
- If the application is already in service, the container MUST respond with an HTTP status code 500 (Internal Server Error) to any subsequent request to the application. The container MAY remove all the disposable application components from service and release them. The whole application MUST remain unavailable across card sessions until the application instance is deleted and recreated.

3.3.2 SIO, Event and Restartable Task Entry Point Methods

Shareable interface methods, `create` methods of SIO-based service factories, `notify` methods of event listeners and `run` methods of restartable tasks constitute as many additional entry points into web applications:

- Shareable interface methods and `create` methods of SIO-based service factories are typically called by other applications but may also be called by the Java Card RE, such as in the case of SIO-based authenticator services which may be called directly by the web container prior to dispatching the request for processing by the web application. See [Section 7.3.3, “SIO-based Service Lookup” on page 7-13](#).
- `notify` methods of event listeners and `run` methods of restartable tasks are called by the Java Card RE. See [Section 7.4.3, “Event Notification” on page 7-23](#) and [Section 2.10.3, “Task Execution” on page 2-56](#).

See [Section 3.3.4, “Multithreading Issues” on page 3-17](#).

3.3.3 Use of Volatile and Persistent Objects

The use by a web application of the `CLEAR_ON_DESELECT` flag when calling one of the `makeTransient...Array()` methods MUST result in a `javacard.framework.SystemException` with reason code `ILLEGAL_TRANSIENT` being thrown.

See [Section 2.8.2.1, “Reachability Disrupting Objects” on page 2-44](#) for more details.

3.3.4 Multithreading Issues

A web application’s code may be concurrently executed by multiple threads. These threads can be categorized as follows:

- Java Card RE-managed threads, which may start executing the web application’s code from one of the following *entry point methods*:
 - One of the lifecycle methods of the web application’s components (servlets, filters, listeners) and especially the `service` method of one of its servlets.
 - The `notify` method of one of its registered event notification listeners. See [Section 7.4, “Events” on page 7-16](#).
 - The `run` method of one of its registered, restartable background tasks. See [Section 2.10, “Restartable Tasks” on page 2-55](#).
- Application-managed threads, which are threads that the web application itself created.
- Other application-managed or Java Card RE-managed threads, which may start executing a web application’s code from one of its shareable interface object methods or from one of its SIO-based service factory `create` methods (see [Section 7.3, “Shareable Interface Object-based Services” on page 7-7](#)) and which originated from an application in another group context. Each such method constitutes an additional *entry point method*.

The web application container MAY use multiple threads to concurrently manage the lifecycle of web application components (servlets, filters and listeners) and to service incoming HTTP requests concurrently. In this specification, these threads are referred to as Java Card RE-managed threads or container-managed threads.

Some web application container implementations MAY limit the number of concurrent container-managed threads, thereby limiting the number of HTTP requests which may be handled concurrently. Some web application container implementations MAY rely on a pool of reusable threads to allow for both limiting the number of threads and allowing threads to be reused for several requests.

Web applications may create new threads. A Java Card Platform implementation MAY restrict the creation of such application-managed threads to ensure that other components of the platform are not negatively impacted. Web applications are responsible for managing the lifecycle of threads they create. They must especially account for restarting such threads, if needed, after platform resets.

3.3.4.1 Thread Safety

The servlet container **MUST** handle concurrent requests to the same servlet by concurrently executing the `service` method on different threads. The servlet container **MUST**, nevertheless, ensure that the re-dispatch of the request to another servlet, such as per an include or forward, occurs in the same thread as the original request.

The servlet container **MAY** not guarantee that the notification of attribute changes to `ServletContext` and `HttpSession` objects be synchronized.

The servlet container **MAY** not guarantee container-managed objects such as request and response objects be thread safe.

These requirements dictate that web application developers handle the related thread safety issues explicitly:

- The handling of concurrent requests to a web application generally requires that servlets be designed to deal with multiple threads executing within the `service` method at a particular time. In particular, access must be synchronized to any shared resources, such as files, network connections, and the servlet's class and instance variables, servlet context and HTTP session objects.
- Multiple threads executing within the `service` method of different servlets (of the same web application) may concurrently access the same `ServletContext` or `HttpSession` object. The developer has the responsibility for synchronizing access to servlet context and HTTP session resources, such those stored as attributes, as appropriate.
- Listener classes handling the notification of attribute changes to `ServletContext` and `HttpSession` objects that maintain state are responsible for the integrity of the data and should handle synchronization explicitly.
- Request and response objects should only be used within the scope of the request handling thread and the call of the `service` method. No reference to such objects should be given to objects executing in other threads or be stored in any manner, such as in instance variables, because the resulting behavior may be nondeterministic.

As mentioned above, a web application's code may be concurrently executed by multiple threads, which may not all be request handling threads. Web application developers must, therefore, explicitly handle thread safety issues on application objects that may be concurrently accessed.

3.3.4.2 Thread Ownership

All web container-managed threads MUST be owned by the Java Card RE. This includes all request handling threads and other web application component lifecycle handling threads. See [Section 2.7.4, “Thread Ownership” on page 2-40](#) for more details on the restrictions that apply to Java Card RE-owned threads.

Threads created by a web application MUST be owned by that application.

3.4 Default Container Behavior and Default Servlet

When the URL of a requested resource cannot be mapped to any of the declared servlets as per the URL path mapping rules described in the *Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition*, in the section on the use of URL paths, the container MUST attempt to serve content appropriate for the resource requested. If a *default servlet* is defined for the application, it MUST be used. This allows for serving the static resources of a web applications in response to requests from web clients.

Note – A servlet may be configured as the default servlet for a web application by being mapped to the “/” URL pattern in its web application deployment descriptor.

In addition to the standard behavior described in the *Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition*, the default container behavior when no default servlet is explicitly configured for an application in its web application deployment descriptor MUST be the following:

- It MUST serve static resources properly requested via HTTP GET or HEAD requests. These static resources MUST be served without imposing any security constraints on their access. See details on the restrictions on security-constrained static web resource below.
- It MUST reject HTTP TRACE or OPTIONS requests and MUST return an HTTP status 501 (Not implemented).
- It MUST deny the access to or modification of static web resources via HTTP POST, DELETE or PUT requests and MUST return an HTTP status 403 (Forbidden).
- It MUST NOT produce directory listings when no welcome file matches a request and MUST return an HTTP status 404 (Not found) instead.

A container **MUST** define a *default default servlet* implementing this default container behavior that is mapped to the `"/"` URL pattern. This servlet **MUST** be implemented by the `javacardx.servlet.http.DefaultServlet` class.

The Java Card Platform version of the specification does not support security constraints on static web resources serviced by the web container as per its default behavior. It imposes the following restriction on the use of `url-pattern` for filter mapping and security-constrained web resource collections: the `url-pattern` value for filter mapping and security-constrained web resource collections **MUST** exactly correspond to the `url-pattern` of one of the `servlet-mapping` elements defined for mapping request URL to servlets in the deployment descriptor.

An application developer who wants to define security constraints or filters for static content served as per the default container behavior must, therefore, explicitly redeclare the mapping to the default servlet as follows:

```
<servlet>

    <description>default servlet</description>

    <servlet-name>default</servlet-name>

    <servlet-class>javacardx.servlet.http.DefaultServlet</servlet-c
lass>

</servlet>

<servlet-mapping>

    <servlet-name>default</servlet-name>

    <url-pattern><path></url-pattern>

</servlet-mapping>
```

`<path>` may be a specific path to an application's static content or it may be `'/'` and may, therefore, designate all that application's static content.

3.5 Secure Hosting of Web Applications

A web application that has requirements for content integrity and confidentiality declares at least one user data security constraint with a transport guarantee value of `INTEGRAL` or `CONFIDENTIAL` in its web application deployment descriptor. See the "Security" chapter of the *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition* for more details on specifying security constraints.

Note – The *overall requirements* of a web application for content integrity and confidentiality corresponds to the transport guarantee requirements declared overall on its web resources, that is whether at least one user data security constraint has a transport guarantee value of INTEGRAL and whether at least one other user data security constraint has a transport guarantee value of CONFIDENTIAL.

On the Java Card Platform, the web container **MUST** implement an application's requirements for content integrity and confidentiality by only accepting requests for that application to which these constraints apply, over HTTPS connections set up accordingly.

Additionally, a web application may define in its runtime descriptor a specific requirement for secure port allocation, see [Web-Secure-Port-Number Attribute](#) in [Chapter 8](#). A web application may also define in its runtime descriptor a requirement for exclusive secure access of all its content, see [Web-Secure-Access-Only Attribute](#) in [Chapter 8](#).

The Java Card Platform **MUST** host on an application-dedicated HTTPS port each web application that has requirements for transport guarantee or that has a requirement for secure port allocation, see [Section 3.5.1, "Port-based Virtual Hosting" on page 3-21](#). The security characteristics of connections from web clients on that application-dedicated HTTPS port **MUST** be negotiated using the credentials retrieved from the `javacardx.security.CredentialManager` instances set for the web application instance, see [Section 3.5.3, "Retrieving a Web Application Instance's Security Requirements and Credentials" on page 3-26](#).

3.5.1 Port-based Virtual Hosting

The Java Card Platform **MUST** host each web application instance according to the transport guarantee requirements expressed in its web application descriptor and according to the secure port allocation requirement expressed in its runtime descriptor. A web application instance **MUST** be hosted as follows:

- **Exclusively on the default plain port for all the content** - If a web application does not have any transport guarantee requirements and is not requesting to be exclusively accessed through secure connections, the web container **MUST** exclusively host this web application instance on the default plain port, by convention, port 80.
- **Exclusively on a dedicated secure port for all the content** - If a web application is requesting to be exclusively accessed through secure connections, the web container **MUST** exclusively host this web application instance on a secure port, regardless of the transport guarantee requirements expressed in its web application descriptor. The web container **MUST** determine the secure port on

which to host all of the web application instance’s content as described in [Determination of a Dedicated Secure Port](#). The web container MUST exclusively host all this web application instance’s content on the dedicated secure port.

- **On the default plain port for unprotected content and on a dedicated secure port for the protected content** - If a web application has one or more transport guarantee requirements and is not requesting to be exclusively accessed through secure connections, the web container MUST determine the secure port on which to host this web application instance’s protected content as described in [Determination of a Dedicated Secure Port](#). The web container MUST exclusively host this web application instance’s protected content on the dedicated secure port. The content of the web application instance that is not constrained by transport guarantees, if any, MUST be equally serviced on the secure port and on the default plain port. Requests on the plain port for content constrained by transport guarantees MUST be rejected and redirected to the secure port, see [Section 3.5.2, “Request Dispatching and Redirection” on page 3-23](#).

TABLE 3-1 summarizes how a web application instance MUST be hosted depending on its overall transport guarantee requirements and its secure port allocation requirement.

TABLE 3-1 Port-based Hosting of Web Application Instances

		Overall Transport Guarantee Requirements Defined	No Transport Guarantee Requirements Defined
Exclusive Secure Access Requested	Secure Port Requested	Exclusively on a dedicated secure port, statically allocated	
	No Secure Port Requested	Exclusively on a dedicated secure port, dynamically allocated	
No Exclusive Access Requested	Secure Port Requested	On the default plain port for unprotected content and on a dedicated secure port, statically allocated, for the protected content	Exclusively on the default plain port (secure port requirement ignored)
	No Secure Port Requested	On the default plain port for unprotected content and on a dedicated secure port, dynamically allocated, for the protected content	Exclusively on the default plain port

Note – A Java Card Platform implementation is not required to support a default secure port because this requires card-wide credentials.

3.5.1.1 Determination of a Dedicated Secure Port

The web container **MUST** determine the secure port on which to host a web application instance's content as follows:

- **Static port allocation** - If the web application requests a specific dedicated secure port number, the web container **MUST** allocate the requested port to the web application instance. If the requested port number is already in use, the instantiation of the web application **MUST** fail, see [Section 3.2.3, "Application Instance Creation" on page 3-5](#).
- **Dynamic port allocation** - If the web application does not request a specific dedicated secure port number, the web container **MUST** select an unused port and allocate this port to this web application instance. If no port can be selected, the instantiation of the web application **MUST** fail, see [Section 3.2.3, "Application Instance Creation" on page 3-5](#).

3.5.2 Request Dispatching and Redirection

A statically or dynamically allocated secure port **MUST** be dedicated to a single web application instance. The web container **MUST** implement the following step to host a web application instance on the secure port allocated to a web application instance:

- It **MUST** listen for HTTPS connections on that port number. The characteristics of each of these connections (the cipher suite used for the connection) **SHOULD** satisfy the highest transport guarantee requirement of all the security constraints expressed by the web application.
- The characteristics of the connection (the cipher suite used for the connection) on which a request is received **MUST** satisfy at least one of the transport guarantee values defined by the security constraint on the queried resource. If none is satisfied, the container **MUST** reject the request with an HTTP status code 403 (Forbidden). This case accounts for web clients connecting on the secure port but not being able to negotiate a secure connection with the level of transport guarantee required for the queried resource, such as being able to negotiate for content integrity but not for confidentiality.
- It **MUST** only accept requests for that web application instance on this port number. Requests for other web application instances **MUST** be rejected with an HTTP status code 404 (Not Found).

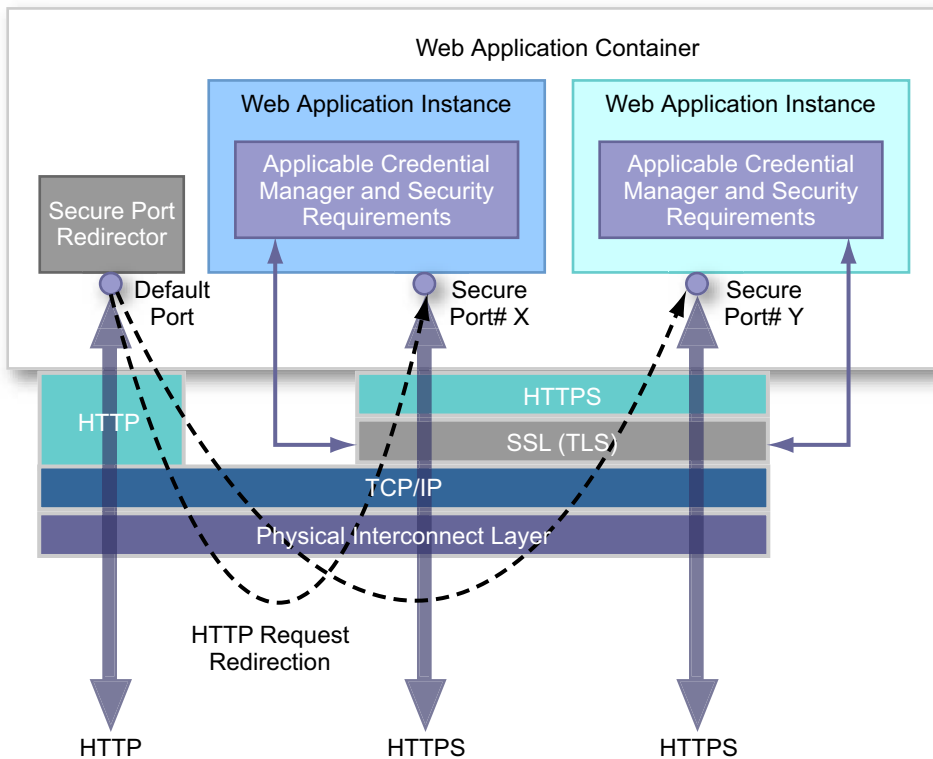
The default plain port is not dedicated to a single web application instance; several web application instances **MAY** be hosted fully or partially on the default plain port. The web container **MUST** implement the following step to host a web application instance or only its unprotected content on the default plain port:

- It **MUST** listen and accept HTTP connections on that port number.

- It MUST reject requests for web application instances that are hosted on a different secure port. It MUST redirect such requests to the dedicated secure port of the queried application instance, see [Section 3.5.2.1, “HTTP Request Redirection”](#) on page 3-25.
- It MUST accept requests for unprotected content of web application instances that are hosted on that port, content for which there is no transport guarantee requirement.
- It MUST reject requests for protected content of web application instances that are hosted on that port, content for which there are transport guarantee requirements. It MUST redirect such requests to the dedicated secure port of the queried application instance, see [Section 3.5.2.1, “HTTP Request Redirection”](#) on page 3-25.

FIGURE 3-1 depicts the secure hosting of web applications and the redirection of requests to their secure ports.

FIGURE 3-1 Request Dispatching and Redirection to Web Applications’ Secure Ports



A web container implementation MAY use the default secure port 443 to host web application instances instead of hosting these applications on the plain default port. Using the default secure port requires that the credentials to be used to establish secure connections with web clients be managed at the web container level, that is, not on a per-application basis. Such a configuration may be specific to an operating environment where cards may be issued with card credentials.

3.5.2.1 HTTP Request Redirection

When performing an HTTP redirection the web container MUST construct an HTTP response with the following characteristics:

- It MUST include a `Location` header field with a value indicating the proper location of the protected content. This location MUST be expressed as a rewritten version of the initially queried URL where the port number has been replaced with the dedicated secure port number of the targeted web application instance.
- It MUST have one of the following HTTP status code:
 - HTTP status code 301 (Moved Permanently) - if the secure port has been statically allocated. A statically assigned port does not change across card sessions and is guaranteed to be the same across cards in the case of a mass deployment of the same application.
 - HTTP status code 307 (Temporary Redirect) – if the secure port has been dynamically assigned. A dynamically assigned port may change across card sessions and moreover is not guaranteed to be the same across cards in the case of a mass deployment of the same application.

Note – The HTTP status code 307 implies that the web client cannot use the new location on the dedicated secure port as a permanent or session-wide substitute for the queried resource on the default plain port. A regular web client, such as a web browser, will typically not retain the new location and will always query the resource on the default port; it will have to be redirected for every single request.

Note – A network-traffic-conscious web client application may avoid HTTP redirection for each request by retrieving the actual dedicated secure port of the queried application from the very first HTTP redirection response and rewriting the subsequent request accordingly. Or more generically, such a web client application may query the web application's root URL on the default port and get redirected to the application's root URL on the application-dedicated secure port. The web client application may then use this URL as the base URL for its communication with the web application.

3.5.3 Retrieving a Web Application Instance's Security Requirements and Credentials

The hosting of a web application on an application-dedicated secure HTTPS port relies on security requirements and credentials, private to the application, to be retrieved from the application itself or its group context during the SSL/TLS protocol handshakes. The SSL/TLS protocol handler **MUST** retrieve the security requirements and the required credentials from the `javacardx.security.CredentialManager.SecurityRequirements` and `javacardx.security.CredentialManager` instances set for the web application instance, either by the card manager application or by the application itself. The `SecurityRequirements` and `CredentialManager` instances to be used **MUST** be the ones that are set for `CredentialManager.MODE_WEB_SERVER` mode of operation.

The `SecurityRequirements` instances are used to determine if client authentication, integrity and confidentiality of the data transmitted are required for connections from any web client on the application-dedicated secure port.

The `CredentialManager` instances are used to retrieve the key and trust material of the web application instances and to make trust decisions, that is, deciding to trust or not a particular web client attempting to connect on the application-dedicated secure port.

A `SecurityRequirements` or `CredentialManager` instance may be set in the *group context* of a web application instance by the card manager and may, therefore, apply to all application instances in that *group context*, or it may be set by a specific web application instance for itself and may, therefore, apply to that application instance only. When a `SecurityRequirements` or `CredentialManager` instance has been set for a mode of communication by a web application, it is retrieved in priority to the default instance set in the group context by the card manager application when establishing a secure communication in that mode for that web application.

The SSL/TLS protocol handler **MUST** invoke methods of the `SecurityRequirements` and `CredentialManager` instances set for `CredentialManager.MODE_WEB_SERVER` mode of operation with at least the following mandatory parameters:

- The `CredentialManager.MODE_WEB_SERVER` mode of operation.
- The connection endpoint URL in the following form `https://:<port number><context path>`; where `<port number>` is the secure port number allocated to the web application instance and `<context path>` corresponds to the context path assigned to the web application instance. Using the connection endpoint URL parameter, methods of the `SecurityRequirements` or `CredentialManager`

instances can discriminate between application instances, allowing each application instance to have its own security requirements settings and its own key and trust material.

See [Section 6.6, “Security Requirements and Credential Management of Secure Communications”](#) on page 6-49 for more details on the use of credential managers.

The web container **MUST** negotiate the security characteristics of HTTPS connections using the `SecurityRequirements` instance applicable for the web connection being established:

- The web container **MUST** negotiate for and require the confidentiality of an HTTPS connection if the `isConfidentialityRequired` method of the `SecurityRequirements` instance applicable for the web connection being established returns `true`.
- The web container **MUST** negotiate for and require the integrity of an HTTPS connection if the `isIntegrityRequired` method of the `SecurityRequirements` instance applicable for the web connection being established returns `true`.
- The web container **MUST** perform client authentication for HTTPS connections to the web application if the `isClientAuthRequired` method of the `SecurityRequirements` instance applicable for the web connection being established returns `true`.

If no applicable `SecurityRequirements` instance can be determined for the web application instance, the requirements **MUST** default to the requirement for client authentication and the overall requirements for content integrity and confidentiality of the web application, as declared in its runtime descriptor and deployment descriptor, respectively.

Note – If the security characteristics of an HTTPS connection negotiated as per the information returned by the applicable `SecurityRequirements` instance do not conform to the transport guarantee requirements for a request to a resource over that connection, the web container **MUST** reject the request with an HTTP status code 403 (Forbidden).

The default implementation of the `isConfidentialityRequired`, `isIntegrityRequired` and `isClientAuthRequired` methods of the base `SecurityRequirements` class **MUST** indicate - for the `CredentialManager.MODE_WEB_SERVER` mode of operation - the requirements of a web application for content confidentiality, content integrity and client authentication, respectively, as declared in the deployment descriptor and runtime descriptor of that web application:

- A web application that has requirements for content integrity and confidentiality declares at least one user data security constraint with a transport guarantee value of `INTEGRAL` or `CONFIDENTIAL` in its web application deployment descriptor.
- A web application may define in its runtime descriptor a specific requirement for web client authentication, see [Web-Client-Auth-Required Attribute](#) in [Chapter 8](#).

An application-defined credential manager may use this information to never request for a security level that would be lower than what the web application's deployment and runtime descriptors are requiring; this, in order to avoid requests to certain of its resources - those that require stronger transport guarantees - to be rejected by the web container.

3.5.3.1 Retrieving a Connection's Specific Security Characteristics

The web container **MUST** provide the following means for a web application to retrieve the security characteristics of the connection over which a request was received:

- The `ServletRequest.isSecure` method indicates whether the request was received on a secure connection.
- The `ServletRequest` attribute `javax.servlet.request.cipher_suite` indicates the cipher suite used for establishing the secure connection.
- The `ServletRequest` attribute `javax.servlet.request.key_size` indicates the key size used for establishing the secure connection.
- The `ServletRequest` attribute `javacardx.security.request.X509Certificate` can be used to retrieve the certificate the client provided for establishing the secure connection, if any.
- The `ServletRequest` attribute `javacardx.security.request.PSKIdentity` can be used to retrieve the Pre-Shared Key Identity the client provided for establishing the secure connection, if any.

See the "The Request" chapter of the *Java Servlet Specification for the Java Card Platform, v3.0.1, Connected Edition* for more details on the SSL connection attributes associated with a request.

APDU-based Application Environment

This chapter describes the interactions and dependencies between the platform, the applet container and APDU-based applet applications:

- [Applet Application Overview](#)
- [Applet Application Lifecycle](#)
- [Lifecycle and Entry Point Method Invocation](#)
- [Classic Applet Application Support](#)

4.1 Applet Application Overview

The two applet application models, classic and extended, support an application programming environment suitable for smart card programs which interact with off-card client applications using the ISO 7816-4 specification defined Application Protocol Data Unit (APDU) synchronous communication paradigm. An applet application receives a command APDU, processes the command, and sends a response APDU containing the result of the processing back to the off-card client. The processing of APDU commands is sequential, meaning, a subsequent APDU command may only be processed from the same off-card client across the I/O interface by an applet application after the first APDU response is returned.

The execution lifecycles of the applet application models are designed to correspond to the specification *Identification Cards - Integrated circuit cards - Part 4: organization, security and commands for interchange* (ISO 7816-4). An applet application **MUST** be selected before it can receive APDU commands. An applet application is selected when the ISO 7816-4 defined SELECT command containing the applet application's Application Identifier (AID) is processed by the applet container or implicitly as a default application. See [Section 4.3.1, "Applet Lifecycle Methods"](#) on page 4-9 and the "Logical Channels and Applet Selection" chapter of the *Runtime Environment*

Specification, Java Card Platform, v3.0.1, Classic Edition. The applet application is deselected when another application is selected to replace it or when the I/O communication interface to the off-card client is reset or a power loss occurs.

The applet container manages two types of applet applications:

- Classic applet applications (See [Section 4.4, “Classic Applet Application Support” on page 4-16](#)):
 - The applet container MUST ensure that classic applet applications execute in a single threaded environment.
 - Classic applet applications may only be programmed to reference the Java Card RE System classes defined in *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Classic Edition*.
 - Classic applet applications may also reference shareable interfaces and classic library classes ([Section 6.7.1, “Class Loader Delegation Hierarchy” on page 6-58](#)).
- Extended applet applications:
 - Extended applet applications must be programmed to execute in a multithreaded environment. The applet container MAY dispatch an APDU command to an extended applet application even when another thread is actively executing the application.
 - Extended applet applications may be programmed to reference the Java Card RE System classes defined in *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition*.
 - Extended applet applications may also reference shareable interfaces and extension library classes ([Section 6.7.1, “Class Loader Delegation Hierarchy” on page 6-58](#)). Extended applet applications MUST NOT reference classic library classes.

An applet that implements the `javacard.framework.Multiselectable` interface is deemed *multi-selectable* and allows itself to be selected when another instance from the same application group is already selected on the card. Either all the applets within an application group MUST be multi-selectable or none.

4.2 Applet Application Lifecycle

Applet application modules are the logical units of assembly of applet applications. All the components of an applet application MUST be assembled into an applet application module. See [Section 8.4.1.2, “Extended Applet Application Module Distribution Format” on page 8-6](#), and [Section 8.4.1.3, “Classic Applet Application Module Distribution Format” on page 8-7](#) for details on the structure of applet application modules.

When being deployed, an applet application will typically go through the following lifecycle:

1. Loading of the applet application module, see [Section 4.2.1, “Application Module Loading” on page 4-3](#).
2. Creation of an instance of the applet application, see [Section 4.2.2, “Application Instance Identification” on page 4-4](#) and [Section 4.2.3, “Application Instance Creation” on page 4-5](#). An applet application instance corresponds to the instantiation of one of the applet classes from the applet application module.
3. Execution of an applet application instance:
 - Dispatch of APDU Commands, see [Section 4.2.7, “Dispatching APDU Commands” on page 4-7](#).
 - Invocation of registered SIO-based services and event listeners, see [Section 4.3.2, “SIO, Event and Restartable Task Entry Point Methods” on page 4-12](#).
 - Execution of registered restartable tasks, see [Section 4.3.2, “SIO, Event and Restartable Task Entry Point Methods” on page 4-12](#).
4. Restart upon platform reset, see [Section 4.2.6, “Restart Upon Platform Reset” on page 4-7](#).
5. Deletion of the applet application instance, see [Section 4.2.4, “Application Instance Deletion” on page 4-6](#).
6. Unloading of the applet application module, see [Section 4.2.5, “Applet Application Module Unloading” on page 4-6](#).

The applet container MUST support multi-instantiation of applet applications. Multiple applet classes defined in the an applet application module may be instantiated and coexist concurrently. In addition, the same applet class defined in an applet application module may be instantiated multiple times and coexist concurrently.

4.2.1 Application Module Loading

When an applet application module is being loaded, the following requirements MUST be implemented by the Java Card Platform:

- The description of the applet application MUST be retrieved from its applet application descriptor located at `APPLET-INF/applet.xml` and from the Java Card Platform-specific application descriptor located at `META-INF/javacard.xml`.
- The runtime configuration of the applet application MUST be retrieved from the applicable runtime descriptor.

- Classes in the module's `APPLET-INF/classes` directory **MUST** be made available to the application class loader assigned for loading the module. See [Section 6.7.1, “Class Loader Delegation Hierarchy” on page 6-58](#) for more details on application class loaders.
- Consistency of the descriptors (especially references to platform resources) and between descriptors (especially cross-references), and consistency between descriptors and code (such as class name references) **MUST** be checked. Note that formal validation of these descriptors against their respective schema is not required on-card, see [Section 8.5, “Descriptor Formats” on page 8-10](#).

Any error during loading, especially module format errors, descriptor format errors, descriptor and code consistency errors, **MUST** result in the loading of the application module to fail and the application module to be rejected.

Refer to [Section 8.6, “Loading Application Modules” on page 8-25](#) for the other generic requirements that **MUST** be implemented by the Java Card Platform when an application module is loaded.

4.2.2 Application Instance Identification

Every applet instance **MUST** be identified by its Application Identifier (AID). The ISO 7816-5 specifications define the AID format and its use for the unique identification of card applications (and files in card file systems). The AID format used in Java Card technology for the identification of an applet instance (as well as applet classes) uses the Category ‘D’ format.

The AID format used by the Java Card platform is an array of bytes that can be interpreted as two distinct pieces, as shown in TABLE 4-1. The first piece is a 5-byte value known as a RID (resource identifier). The second piece is a variable length value known as a PIX (proprietary identifier extension). A PIX can be from 0 to 11 bytes in length. Thus an AID can be from 5 to 16 bytes in total length.

TABLE 4-1 AID Format

RID (5 bytes)	PIX (0-11 bytes)
---------------	------------------

ISO controls the assignment of RIDs to companies, with each company obtaining its own unique RID from the ISO. Companies manage assignment of PIXs for AIDs using their own RIDs.

Applet instances are selected for APDU communication using the ISO 7816-4 defined SELECT command containing AID information to identify the applet instance. Once selected, APDU commands may be routed to the applet instance by the applet container.

Internal to the card, every applet instance is named with a relative URI - its *application URI* - that uses the `aid` registry-based authority and whose path component corresponds exactly to its AID (See [Section 2.3.1, “Unified Naming Scheme” on page 2-5](#)). This application URI defines the root of a dedicated namespace within which all its resources, including SIO-based services, events and files, **MUST** be named.

Note – To maximize portability, an extended applet application developer should name or refer to the applet application’s resources both in code and descriptors using relative URIs.

4.2.3 Application Instance Creation

When an applet application instance is being created, the following requirements **MUST** be implemented by the Java Card Platform:

1. **The applet class’ static `install` method **MUST** be called with installation parameter data to initiate application instance creation.**

The format of the installation parameter data passed to the target applet’s `install` method **MUST** be in the format described in the API description of the method in *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition*.

2. **The applet instance which is created by the applet’s `install` method must be registered by the application via the applet’s `register` method.**

The registered applet instance **MUST** be rooted at the application-URI corresponding to the applet instance AID in the applet container’s namespace. This instance of the applet **MUST** serve as the logical root of persistence for the application instance, see [Section 4.2.8, “Container-managed Object Lifetime and Persistence” on page 4-7](#).

3. **The class-path resources in the module’s `APPLET-INF/classes` directory are shared among all applet instances from the applet application module and **MUST** be made available to the application’s code using the `Class.getResourceAsStream` method.**

4. **The applet container **MAY** start dispatching APDU commands as per [Section 4.2.8, “Container-managed Object Lifetime and Persistence” on page 4-7](#).**

Note – Because an applet application may be instantiated multiple times from the same loaded module code, the developer must account for static fields and initializers not being reset or re-run at each instantiation.

Refer to [Section 8.8, “Creation of Application Instances”](#) on page 8-31 for the other generic requirements, such as assignment to the proper group context and protection domain, that **MUST** be implemented by the Java Card Platform when an application instance is created.

4.2.4 Application Instance Deletion

When an applet application instance is being deleted, the following requirements **MUST** be implemented by the Java Card Platform:

1. **Before firing the application instance deletion request event,** `event:///standard/app/deleting`, **the Java Card RE *MUST* ensure that the applet instance being deleted is not currently selected for APDU communication on the card. If so, deletion *MUST NOT* be attempted and the operation fails.**
2. **Before firing the application instance deletion request event, the Java Card RE *MUST* inform each of the applets of potential deletion by invoking, if implemented, the applet instance's `uninstall` method of the `AppletEvent` interface.**
3. **Upon successful deletion:**
 - a. The applet instance's AID corresponding to its application URI **MUST** be released.
 - b. The applet application's temporary storage directory and its content **MUST** be removed.

The applet instance's AID **MUST** only be reassigned when the application instance is considered effectively deleted, see [Section 8.9, “Deletion of Application Instance”](#) on page 8-33.

Refer to [Section 8.9, “Deletion of Application Instance”](#) on page 8-33 for the other generic requirements that **MUST** be implemented by the Java Card Platform when an application instance is deleted.

4.2.5 Applet Application Module Unloading

When an applet application module is being unloaded, the following requirements **MUST** be implemented by the Java Card Platform:

- Any instance of the application **MUST** first have been deleted as per [Section 4.2.4, “Application Instance Deletion”](#) on page 4-6.
- Classes and resources in the module's `APPLET-INF/classes` directory **MUST** no longer be available to the class loaders that loaded the classes from the module.

Refer to [Section 8.10, “Unloading of Deployment Units”](#) on page 8-36 for the other generic requirements that MUST be implemented by the Java Card Platform when an application module is unloaded.

4.2.6 Restart Upon Platform Reset

There are no applet application environment-specific steps required before any APDU command can be dispatched to an applet application instance after a platform reset.

Refer to [Chapter 5](#) for the other generic requirements that MUST be implemented by the Java Card Platform after a platform reset.

4.2.7 Dispatching APDU Commands

The applet container is responsible for managing the selection of applet instances and dispatching APDU commands to selected applet instances.

The Java Card Platform, v3.0.1, Connected Edition MAY support more than one I/O interface, across which APDU commands are received. Each of these I/O interfaces MUST support the ISO 7816-4 specification-defined APDU communication protocol. Over each of these interfaces, the platform MAY support up to 20 ISO 7816-4 defined logical channels of APDU communication. The applet container MUST either configure a supported logical channel to have a default applet instance or MUST designate the logical channel as having no associated default applet instance.

The process of management of logical channels and applet selection over an I/O interface is described in the “Logical Channels and Applet Selection” chapter of the *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*.

The applet container MAY manage multiple APDU command dispatcher threads. The multithreading issues around managing separate dispatcher threads for each I/O interface is described in [Section 4.3.4, “Multithreading Issues For Applets”](#) on page 4-13.

4.2.8 Container-managed Object Lifetime and Persistence

When an applet instance of an applet application is instantiated, the APDU application container MUST invoke the `Applet.install()` method, which in turn creates an instance of applet (instance of application-defined subclass of the `javacard.framework.Applet` class). This instance MUST be promoted to become

a persistent object by virtue of being reachable from a persistent APDU container managed object. This instance of the `Applet` serves as the logical root of persistence for the newly instantiated applet instance of the applet application. This instance of `Applet` is deleted only when the applet instance of the applet application is deleted.

When executing an applet application, the container **MUST** instantiate container-managed objects, that is, objects created by the container as per its operating principles. Container-managed objects include instances of the `javacard.framework.APDU` class. These instances are Java Card RE entry point object instance, see [Section 2.4.2.1, “Java Card RE Entry Point Objects” on page 2-20](#).

All container-managed objects created on behalf of the application **MUST** be Java Card RE entry point objects (see [Section 2.4.2.1, “Java Card RE Entry Point Objects” on page 2-20](#)).

Specific lifetime and persistence requirements of some container-managed objects are described in the following sections. Any container-managed object which is not called out specifically **MAY** have implementation assigned lifetime and persistence characteristics.

4.2.8.1 Applet Lifetime

The following container-managed object **MUST** be a persistent object. It is created when an applet instance of the applet application is instantiated and can be garbage collected only when the applet instance of the applet application is deleted.

- **the applet:** The context is created when the applet instance of the applet application is instantiated. This is the root of persistence for the application.

4.2.8.2 Command/Response Object Lifetime

The following container-managed objects **MUST** be volatile objects:

- **The APDU object:** an instance of this temporary Java Card RE entry point object (see [Section 2.4.2.1, “Java Card RE Entry Point Objects” on page 2-20](#)) is created for each I/O interface, across which APDU commands are received, and reused over multiple APDU dispatch cycles.
- **The APDU buffer array:** an instance of this global array object (see [Section 2.4.2.2, “Global Arrays” on page 2-24](#)) is created for each I/O interface, across which APDU commands are received, and reused over multiple APDU dispatch cycles.

Note – Container-managed volatile command/response objects **SHOULD** be referenced via reachability disrupting objects from the container object itself.

4.2.8.3 Behavior Upon Platform Reset

Upon card reset, all volatile objects **MUST** be garbage collected. All references to volatile objects in reachability disrupting objects **MUST** be reset to `null`.

4.3 Lifecycle and Entry Point Method Invocation

4.3.1 Applet Lifecycle Methods

The applet container invokes the applet's methods to manage all aspects of the applet's lifecycle. The methods correspond to application creation, selection, APDU message processing and application deletion. The following sections describe each lifecycle state.

4.3.1.1 Applet Application Instance Creation

The applet container invokes the applet's static `install` method to initiate application instance creation. The application typically processes the initialization parameter information, creates and initializes applet application resources, and finally registers the newly created applet instance with the applet container via the `Applet.register` method.

A successful completion of the `Applet.register` method results in the successful completion of the applet application instance. Upon successful instantiation, and following any additional applet instance initialization, the applet container **MUST** mark the applet instance as available for selection.

The applet container **MUST** invoke the applet's `install` method with a transaction in progress. The applet container **MUST** abort this transaction, if the instantiation is unsuccessful.

4.3.1.2 Applet Application Selection

The applet container invokes one of the applet instance's `select` methods to inform the applet instance upon becoming selected on a logical channel for APDU communication. Upon successful selection, the applet instance is deemed as being active on the card. The choice of `select` method used by the applet container is as follows:

- The applet container invokes the `Applet.select` method of the applet instance, if there are no applet instances from within the application group that are currently active.
- Otherwise, the applet container invokes the `Multiselectable.select` interface method of the applet instance, if there is at least one other applet instance from within the application group that is currently active. Applet selection fails if the applet does not implement the `Multiselectable` interface and there is at least one applet instance from within the application group that is currently active.

Applet selection fails if the `select` method returns `false` or throws an uncaught exception.

If applet selection fails, the applet container **MUST** mark the corresponding logical channel as having no applet selected. If the applet selection was initiated by the SELECT APDU Command or the MANAGE CHANNEL APDU Command, the applet container **MUST** send an APDU Response to the off-card client with status code equal to `0x6999` (`javacard.framework.ISO7816.SW_APPLET_SELECT_FAILED`).

If the applet selection is successful and the selection was initiated by the SELECT APDU command, the applet container **MUST** dispatch the SELECT APDU command to the newly selected applet for processing via the `Applet.process` method.

The applet container **MUST** abort the transaction in progress, if any, upon normal return or uncaught exception from the `process` method. When the applet container aborts the transaction, it proceeds as if an uncaught exception was thrown.

The step-by-step processing requirements of the SELECT APDU command and MANAGE APDU command are described in the sections “Opening and Closing Logical Channels” and “Applet Selection” in the chapter “Logical Channels and Applet Selection” of the *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*.

4.3.1.3 Applet Application Execution

The applet container MUST dispatch APDU commands received over a logical channel to the applet instance currently selected on that logical channel for processing. The applet container MUST call the `Applet.process` method with the APDU object parameter containing the incoming APDU command.

If there is no applet selected on the logical channel, the applet container MUST send an APDU response to the off-card client with status code equal to `0x6999` (`javacard.framework.ISO7816.SW_APPLET_SELECT_FAILED`).

Upon normal return from the `process` method, the applet container MUST send an APDU response to the off-card client with status code equal to `0x9000` (`javacard.framework.ISO7816.SW_NO_ERROR`) along with any accumulated response data in the APDU object.

If the `process` method throws an uncaught `ISOException` exception, the applet container MUST send an APDU response to the off-card client with status code equal to the reason code contained in the `ISOException` object, along with any accumulated response data in the APDU object.

If the `process` method throws any other uncaught exception, the applet container MUST send an APDU response to the off-card client with status code equal to `0x6F00` (`javacard.framework.ISO7816.SW_UNKNOWN`), along with any accumulated response data in the APDU object.

Note – In some cases of ISO 7816 protocol, partially accumulated response data must be filled out, with zeros, to account for the previously set applet response length, before being sent to the off-card client. For more details, see “API Topics - The APDU Class” section of the “Logical Channels and Applet Selection” chapter in *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*.

The applet container MUST abort the transaction in progress, if any, upon normal return or uncaught exception from the `process` method. When the applet container aborts the transaction, it proceeds as if an uncaught exception was thrown.

The specifics of APDU command decoding to determine which logical channel to dispatch an incoming APDU command is described in the sections “Forwarding APDU Commands to a Logical Channel” and “Other Command Processing” of the chapter “Logical Channels and Applet Selection” in *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*.

4.3.1.4 Applet Application Deselection

The applet container invokes one of the applet instance's `deselect` methods to inform the applet instance upon becoming deselected on a logical channel for APDU communication. The choice of `deselect` method used by the applet container is as follows:

- The applet container invokes the `Applet.deselect` method of the applet instance, if there are no applet instances from within the application group that are currently active.

Components of transient array objects of type `CLEAR_ON_DESELECT` associated with the applet instances from their application group **MUST** be cleared.

- Otherwise, the applet container invokes the `Multiselectable.deselect` interface method of the applet instance, if there is at least one applet instance from within the application group that is currently active.

The applet container **MUST** abort the transaction in progress, if any, upon normal return or uncaught exception from the `deselect` method.

The step-by-step processing requirements of applet deselection are described in the section “Applet Deselection” in the chapter “Logical Channels and Applet Selection” of *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*.

4.3.1.5 Applet Application Deletion

The applet container invokes the `AppletEvent.uninstall` interface method of the applet instance, if implemented, to inform the applet instance that there is a request to delete it. The applet typically performs clean up activities including removing any dependencies which might restrict its deletion.

The applet container **MUST** abort the transaction in progress, if any, upon normal return or uncaught exception from the `uninstall` method.

4.3.2 SIO, Event and Restartable Task Entry Point Methods

Shareable interface methods, `create` methods of SIO-based service factories, `notify` methods of event listeners and `run` methods of restartable tasks, constitute as many additional entry points into applet applications:

- Shareable interface methods and `create` methods of SIO-based service factories are typically called by other applications but may also be called by the Java Card RE. This can occur in the case of SIO-based authenticator services, which may be

called directly by the applet container prior to dispatching the request for processing by the applet application. See [Section 7.3.3, “SIO-based Service Lookup”](#) on page 7-13.

- notify methods of event listeners and run methods of restartable tasks are called by the Java Card RE. See [Section 7.4.3, “Event Notification”](#) on page 7-23 and [Section 2.10.3, “Task Execution”](#) on page 2-56.

See [Section 4.3.4, “Multithreading Issues For Applets”](#) on page 4-13.

4.3.3 Use of Volatile and Persistent Objects

Components of transient array objects of type `CLEAR_ON_DESELECT` MUST be cleared whenever an applet instance is deselected (explicitly or implicitly) and no applet instances from the application group are active on the card.

See [Section 2.8.2.1, “Reachability Disrupting Objects”](#) on page 2-44 for more details.

4.3.4 Multithreading Issues For Applets

This section describes multithreading issues that apply to the extended applet application environment. The extended applet application environment MAY be multithreaded. Extended applet applications MUST be thread-aware and MUST, therefore, account for concurrent processing.

Classic applet applications are not thread-aware. Therefore, the classic applet application environment MUST guarantee the thread-safety of classic applet applications. See [Section 4.4.2, “SIO Synchronization Proxy Classes”](#) on page 4-17.

An extended applet application’s code may be concurrently executed by multiple threads. These threads can be categorized as follows:

- Java Card RE-managed threads that may start executing the extended applet application’s code from one of the following *entry point methods*:
 - One of the lifecycle methods of one of the application’s applets, in particular the `process()` method, of one of its applets.
 - The `notify()` method of one of its registered event notification listener. See [Section 7.4, “Events”](#) on page 7-16.
 - The `run()` method of one of its registered restartable background tasks. See [Chapter 2](#).
- Application-managed threads, which are threads that the extended applet application created.

- Other application-managed or Java Card RE-managed threads that may start executing an extended applet application's code from one of its Shareable Interface Object methods or from one of its SIO-based service factory `create()` methods (see [Section 7.3, “Shareable Interface Object-based Services”](#) on page 7-7) and that originated from another application's context. Each such Shareable Interface Object method constitutes an additional *entry point method*.

The applet application container MAY use multiple threads to concurrently manage the lifecycle of applets and to process incoming APDU commands concurrently over several interfaces. These threads are referred to in this specification as Java Card RE-managed threads or container-managed threads. The applet container MAY handle APDU commands concurrently over each of the contacted and the contactless interfaces, but MUST handle only one command at a time per interface. Because thread safety must be ensured for classic applets, the applet container MUST guarantee that only one APDU command received over any of the interfaces is processed by a classic applet at a particular time. Extended applets, by contrast, may process concurrently one command over each interface. Each command is handled by a different thread.

Some applet application container implementations MAY limit the number of concurrent container-managed threads. This may, therefore, limit the number of APDU commands that may be processed concurrently.

Extended applet applications may create new threads. A Java Card Platform implementation MAY restrict the creation of such application-managed threads to ensure that other components of the platform are not negatively impacted. Extended applet applications are responsible for managing the lifecycle of threads they create. They must account for restarting such threads, if needed, after platform resets.

4.3.4.1 Thread Safety

The applet container MUST handle concurrent APDU commands to the same extended applet by concurrently executing the `process()` method in different threads.

The applet container MAY use a single instance of the APDU object and APDU buffer array per interface being concurrently serviced. The applet container MUST use a different instance of the APDU object and APDU buffer array for each of the APDU commands being concurrently handled.

APDU objects are temporary Java Card RE entry point objects (see [Section 2.4.2.1, “Java Card RE Entry Point Objects”](#) on page 2-20) and the associated APDU buffers are global array objects (see [Section 2.4.2.2, “Global Arrays”](#) on page 2-24). References to these temporary objects cannot be stored in class variables, in instance variables, or in array components (see [Section 2.4, “Context Isolation Basics”](#) on page 2-14). These objects can, therefore, only be used within the scope of the

command handling thread and the call of the `process()` method. No reference to such objects can be given to objects executing in other threads. The thread safety of such objects is, therefore, not required.

The applet container **MUST** guarantee that only one APDU command received over any of the interfaces is processed by a classic applet at a particular time.

The following requirements dictate that extended applet application developers handle the related thread safety issues explicitly:

- The handling of concurrent APDU commands to an extended applet application generally requires that applets be designed to deal with multiple threads executing within the `process()` method at a particular time. In particular, access must be synchronized to any shared resources, such as files, network connections, and the applet's class and instance variables.
- As mentioned above, an extended applet application's code may be concurrently executed by multiple threads, which may not all be APDU command handling threads. Extended applet application developers must, therefore, explicitly handle thread safety issues on application objects that may be concurrently accessed.

Classic applet application developers do not have to specifically account for thread safety.

Caution – Although the classic applet application environment is thread safe, a classic applet invoking Shareable Interface Objects methods of an extended applet application or web application, such as on a SIO-based service, may encounter unexpected behavior. For example, this could occur if the state of the service concurrently changes while within the scope of the same `process()` method invocation. This is because that same SIO-based service object may be concurrently accessed by other extended applet or web applications.

4.3.4.2 Thread Ownership

All applet container-managed threads **MUST** be owned by the Java Card RE. This includes all APDU command handling threads and other applet application component lifecycle handling threads. See [Section 2.7.4, “Thread Ownership” on page 2-40](#) for more details on the restrictions that apply to Java Card RE-owned threads.

Threads created by an extended applet application **MUST** be owned by that application.

4.4 Classic Applet Application Support

The term *classic application* is applied to an applet application that may also be executed on the Java Card 3 Platform Classic Edition. The Java Card 3 Platform Classic Edition supports a more restrictive runtime environment and a more restrictive set of core and framework libraries than the Connected Edition. However, the applet container MUST guarantee the same functional behavior of a classic application on the Java Card 3 Platform Connected Edition provided that the classic application meets the following requirements:

- The classic applet application uses only the API defined in *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Classic Edition*, which is a strict subset of the API defined in *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition*.
- The classic applet application uses only the virtual machine features defined in *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*, which is a strict subset of the features defined in *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Connected Edition*.
- The classic applet application is encapsulated in the classic applet application distribution format ([Section 8.4.1.3, “Classic Applet Application Module Distribution Format” on page 8-7](#)) for a classic applet application or the classic library distribution format ([Section 8.4.3, “Classic Library Distribution Format” on page 8-9](#)) for a classic library.
- The classic applet application shares classic libraries only with other classic applications. See [Section 6.7, “Code Isolation” on page 6-57](#).
- The classic applet application does not use nested transactions and the annotation-based model defined in [Section 2.9.2, “Transaction Demarcation” on page 2-47](#). Instead the classic application may use the transaction model defined in *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*.

4.4.1 Backward Compatibility

The Java Card 3 Platform Connected Edition, along with off-card compatibility tools, MUST ensure that Java applications programmed for the Java Card Platform, Version 2.2.2, run unchanged. These applications are also classic applications and may be executed on both the Java Card 3 Platform Connected and Classic Editions.

Classic applications on the Java Card 3 Platform Connected Edition MUST:

- execute in a single threaded environment
- execute APDU commands sequentially even when received over distinct I/O interfaces

4.4.2 SIO Synchronization Proxy Classes

The Java Card RE uses SIO synchronization proxy classes to ensure that only one application thread is executing any classic application or classic library. An SIO synchronization proxy instance is substituted in place of the actual SIO instance of the classic application whenever it becomes accessible to an extended applet or a web application. This SIO synchronization proxy instance for an SIO instance of a classic application synchronizes with the singleton *classic applet container mutex object* before entering the SIO instance of the classic application. Additionally, an SIO synchronization proxy instance is substituted in place of the actual SIO instance of the extended or web application whenever it becomes accessible to a classic application. This SIO synchronization proxy instance for an SIO instance of an extended applet or a web application ensures that any SIO instances of a classic application that are passed as a parameters to the extended applet or web application are substituted with suitable SIO synchronization proxy instances.

The SIO proxy class **MUST** be a concrete implementation class which extends the `javacardx.framework.ClassicSIOProxy` abstract class, and implements the referenced shareable interfaces. The proxy class **SHOULD** implement the delegation software programming pattern for all the shareable interface methods. The `ClassicSIOProxy` class has a `setSIO` method that takes the SIO instance as a parameter and stores it internally for delegation. The `getSIO` method returns the SIO instance.

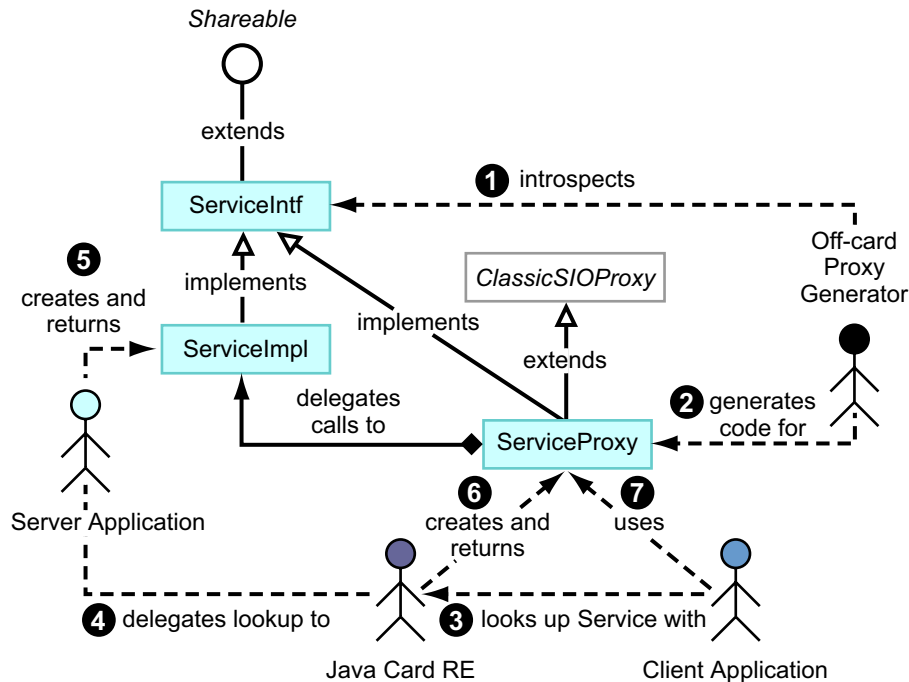
The classic application or classic library **MUST** include SIO proxy classes¹ in a subpackage named `proxy` of the classic application or classic library package for all the shareable interfaces referenced by the package. The proxy classes in the subpackage are named `Proxy1`, `Proxy2`, and so forth. For each Shareable Interface Object implementation class in the classic application or library, there **SHOULD** be a proxy class which implements exactly the same set of Shareable interfaces. Additional proxy classes **SHOULD** implement as many combinations of referenced shareable interfaces as possible without method signature conflicts or duplication. The complete set of additional proxy classes **SHOULD** span all the referenced shareable interfaces in as few classes as possible. The set of proxy classes implemented **MAY** be customized for the classic application or library module to address the programming pattern used in the module.

These proxy classes are dynamically loaded by the Java Card RE on behalf of the application by the SIO proxy generation mechanism implemented by the Java Card RE and **MUST** therefore be declared in the `dynamically-loaded-classes` element of the Java Card Platform-specific Application descriptor. See [Section 8.5.3, “Java Card Platform-specific Application Descriptor”](#) on page 8-11.

1. Off-card pre-processing tools which process classic applications--Converter and Normalizer--typically insert these SIO Proxy Classes.

FIGURE 4-1 depicts the relationships and interactions between the components and actors of the SIO synchronization proxy mechanism. The numbering indicates a typical interaction sequence. Colors indicate common ownership of interacting objects.

FIGURE 4-1 SIO Synchronization Proxy Mechanism (Collaboration Diagram)



The SIO proxy class is instantiated by the SIO Proxy generation mechanism via its default constructor. The SIO proxy class MUST invoke each corresponding shareable interface method of the SIO object within a *synchronized* block, only after obtaining a lock on the singleton *classic applet container mutex object* obtained via the static field `CLASSIC_APPLET_CONTAINER_MUTEX` of the `ClassicSIOProxy` class. If the shareable interface method returns an object, the equivalent SIO proxy class method MUST call the `ClassicSIOProxy.processReturn` method to process the object and substitute a proxy for the object, if required, before returning from the shareable interface method. If the shareable interface method has an object as a parameter, the equivalent SIO proxy class method MUST call the `ClassicSIOProxy.processParam` method to process the object and substitute a proxy for the object, if required, prior to passing it to the shareable interface method.

The SIO Proxy generation mechanism implemented by the Java Card RE MUST search for a suitable SIO synchronization proxy class corresponding to the specific SIO object, within the `Proxy` subpackage of the classic applet application package or classic library package. The SIO Proxy generation mechanism is used by the

ClassicSIOProxy.processReturn and the ClassicSIOProxy.processParam methods as well as by the ServiceRegistry.lookup and the JCSys^{tem}.getAppletShareableObject SIO discovery methods.

Two shareable interfaces defined by the classic server application, WalletSI and BalanceSI, and one shareable interface defined by the client, AccSI, as well as their associated SIO proxy classes in the proxy subpackage are shown below:

CODE EXAMPLE 4-1 SIO Synchronization Proxy Classes Example

```
package com.sun.testServer;
/*shareable interface in classic server application/library package*/
public interface WalletSI extends Shareable {
    BalanceSI getBalance(AccSI acc);
    short getVersion();
}

package com.sun.testServer;
/*shareable interface in classic server application/library package*/
public interface BalanceSI extends Shareable {
    int getHi();
    int getLo();
    short getVersion();
}

package com.sun.testClient;
/* shareable interface in client application/library package*/
public interface AccSI extends Shareable {
    short getAccNum();
    byte getVersion();
}

package com.sun.testServer.proxy;
/* First SIO Proxy Class in proxy subpackage of server application or
library implements 2 out of 3 referenced shareable interfaces*/

public class Proxy1 extends ClassicSIOProxy
    implements WalletSI, BalanceSI{
    public BalanceSI getBalance(AccSI acc) {
        try {
            synchronized (CLASSIC_APPLET_CONTAINER_MUTEX) {
                return (BalanceSI)(processReturn(((WalletSI)getSIO()).
                    getBalance((AccSI)processParam(acc))));
            }
        } catch (ClassNotFoundException e) {
            throw new RuntimeException("SIO Synchronization Proxy
Error: Class Not Found");
        } catch (InstantiationException e) {
            throw new RuntimeException("SIO Synchronization Proxy
Error: Instantiation Exception");
        }
    }
}
```

CODE EXAMPLE 4-1 SIO Synchronization Proxy Classes Example (*Continued*)

```
public int getHi(){
    synchronized (CLASSIC_APPLET_CONTAINER_MUTEX){
        return ((BalanceSI)getSIO()).getHi();
    }
}

public int getLo(){
    synchronized (CLASSIC_APPLET_CONTAINER_MUTEX){
        return ((BalanceSI)getSIO()).getLow();
    }
}

public short getVersion() {
    synchronized (CLASSIC_APPLET_CONTAINER_MUTEX){
        if (getSIO() instanceof BalanceSI){
            return ((BalanceSI)getSIO()).getVersion();
        } else if (getSIO() instanceof WalletSI){
            return ((WalletSI)getSIO()).getVersion();
        } else{
            throw new RuntimeException("SIO Synchronization Proxy
                                     Error");
        }
    }
}

package com.sun.testServer.proxy;
/* Second SIO Proxy Class in proxy subpackage of server application
or library implements 1 out of 3 referenced shareable interfaces*/
public class Proxy2 extends ClassicSIOProxy
    implements AccSI{
    public short getAccNum(){
        synchronized (CLASSIC_APPLET_CONTAINER_MUTEX){
            return ((AccSI)getSIO()).getAccNum();
        }
    }
    public byte getVersion() {
        synchronized (CLASSIC_APPLET_CONTAINER_MUTEX){
            return ((AccSI)getSIO()).getVersion();
        }
    }
}
```

Caution – An extended applet application or web application that defines a SIO implementation class must not pass the SIO instance to its classic applet client via another intermediary classic applet application which obscures visibility into the actual shareable interfaces implemented by the SIO instance by simply declaring the

SIO instance obtained from the extended applet application as a generic `javacard.framework.Shareable` Interface type. The SIO proxy class created for a SIO object class defined in an extended applet or web application may correspond only to the subinterfaces of the `javacard.framework.Shareable` Interface referenced by the classic applet application.

Caution – The SIO proxy class for a SIO object class defined in an extended applet or web application is created within the classic application client's proxy subpackage and corresponds only to the subinterfaces of the `javacard.framework.Shareable` Interface referenced by the classic application. These SIO proxy classes may implement more shareable interfaces than those implemented by the original SIO instance. In this case, the classic application may encounter a false positive indication when performing “instance of” checks on the interfaces implemented by the substituted proxy object.

4.4.2.1 Single Threaded Runtime Environment

To ensure that classic applications and classic libraries execute in a single threaded environment, the following requirements **MUST** be supported:

- The applet container **MUST** invoke an entry point method of a classic application within a *synchronized* block, only after entering the monitor for the singleton, Java Card RE owned, *classic applet container mutex object* obtained via the static field named `CLASSIC_APPLET_CONTAINER_MUTEX` of the `ClassicSIOProxy` class.
- When invoked from an extended applet application or a web application, the Java Card RE implementation of the lookup methods of the `javacardx.facilities.ServiceRegistry` class, and the `getAppletShareableInterfaceObject` method of the `javacard.framework.JCSystem` class, **MUST** substitute an instance of an SIO synchronization proxy class corresponding to the SIO object being returned from the `getShareableInterfaceObject` method of the `javacard.framework.Applet` class of the classic application. The SIO synchronization proxy instance **MUST** be owned by the classic application.

If the `getShareableInterfaceObject` method of the `javacard.framework.Applet` class of the classic application returns an SIO synchronization proxy instance for an SIO object owned by an extended applet or web application, the contained SIO object **MUST** be substituted instead.

- When invoked from a classic application, the Java Card RE implementation of the `getAppletShareableInterfaceObject` method of the `javacard.framework.JCSystem` class, **MUST** substitute an instance of the SIO synchronization proxy class corresponding to the SIO object being returned from

the `getShareableInterfaceObject` method of the `javacard.framework.Applet` class of the extended applet application. The SIO synchronization proxy instance **MUST** be owned by the classic application.

If the `getShareableInterfaceObject` method of the `javacard.framework.Applet` class of the extended application returns an SIO synchronization proxy instance for an SIO object owned by an classic applet, the contained SIO object **MUST** be substituted instead.

4.4.3 Restricted Visibility on Classic Library

Classic libraries **MUST** be loaded by the classic library class loader, which is in the delegation hierarchy of classic applet applications only ([Section 6.7.1, “Class Loader Delegation Hierarchy” on page 6-58](#)). This ensures that classic library code always executes in a single threaded environment.

4.4.4 Classic Transaction Model

Classic applications **MUST** be restricted to use the transaction model and facilities described in the chapter “Transactions and Atomicity” of the *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*.

To ensure the properties of the classic transaction model, all classes of a classic application or classic library **MUST** be annotated with the `TransactionType - SUPPORTS`². This ensures that transitions from one method to another when a classic transaction is in progress, continue the ongoing transaction.

4.4.5 Special Security Restrictions

All classic applications are associated with the *classic application platform protection domain*. The classic application platform protection domain is the security configuration for all classic applications and specifies the security permissions that are included and those that are not included ([Section 6.2.2, “Protection Domains” on page 6-13](#)).

The special security restrictions for classic applications ensure that classic applications do not encounter unexpected behavior or security attacks in the presence of thread-aware extended applet and web applications.

2. Off-card pre-processing tools which process classic applications--Converter and Normalizer--typically insert these transaction annotations.

Card Initialization and Power-up

The card initialization requirements are described in this chapter. Platform requirements during I/O interface reset are also included.

- [Card Initialization](#)
- [Power-up and Card Reset](#)
- [I/O Interface Reset](#)
- [Concurrently Active Interfaces](#)

5.1 Card Initialization

The card initialization occurs after masking and prior to card personalization and issuance. When the card is initialized, the Java Card RE is also initialized. Java Card RE initialization **MUST** be performed in the following sequence:

1. The Java Card virtual machine is started. The internal state required by the virtual machine and the Java Card RE to run Java applications is created and initialized.

In Java Card technology, the execution lifetime of the virtual machine (VM) is the lifetime of the card. The information required to run the applications that have been instantiated on the card **MUST** be preserved in persistent memory even when power is removed from the card. Because the VM and the objects created on the card are used to represent application information that is persistent, the Java Card VM appears to run forever. When power is removed, the VM only stops temporarily. When the card is powered up again, the VM starts again and recovers its previous persistent object heap from persistent memory store.

2. All Java Card RE facilities are initialized.
 - a. All the core platform classes and framework classes are created, loaded, linked and initialized

3. The platform policy (platform protection domains) is loaded
 4. The inter-application communication facility is initialized
 - a. The SIO factory registry is created
 - b. The event notification registry is created
 5. The authentication facility is initialized
 6. The restartable tasks facility is initialized:
 - a. The restartable task registry is created
 7. The card management facility is initialized:
 - a. The application registry is created
 8. The application containers are initialized:
 - a. The applet container is initialized
 - b. The web container is initialized
 9. The card manager application is loaded and instantiated
-

5.2 Power-up and Card Reset

When power is re-applied following a power down (or tear) to the card, or reset operation is performed on the card (card reset), the Java Card RE MUST ensure that the following actions are performed in sequence:

1. All volatile objects are no longer accessible.
2. The component values of transient arrays are reset to their default values - 0 or null.
3. The internal reference field of all `TransientReference` instances is reset to null.
4. All monitors for objects are unlocked.
5. Persistent thread objects, instances of the `java.lang.Thread` class reachable from a root of persistence, MUST be placed in the “terminated” state and cannot be restarted.
6. All transactions in progress when power was lost are aborted. Updates to persistent object fields, array components and static fields which occurred during the aborted transaction are rolled back.

7. All I/O interfaces are initialized:
 - a. Any I/O interface which has been activated and supports the ISO 7816-4 defined APDU protocol is initialized per [Section 5.3.1, "ISO 7816-4 Reset"](#) on page 5-4.
 - b. Any I/O interface which has been activated and supports the TCP/IP protocol is initialized per [Section 5.3.2, "Transmission Control Protocol \(TCP\) Reset"](#) on page 5-4.
8. Any persistent GCF connection objects, instances of classes implementing the `javax.microedition.io.Connection` interface, that are reachable from the persistence root MUST be placed in a "closed" state.
9. The containers are restarted and MUST take the necessary actions as per the application model they support:
 - a. The applet container MUST perform the following actions in sequence:
 - i. All applet instances that were active when power was lost are implicitly deselected.
 - b. The web container MUST perform the following actions in sequence:
 - i. All active sessions are cleared - volatile HTTP session objects are garbage collected.
 - ii. All requests and responses are cleared - volatile HTTP request and response objects are garbage collected.

Listeners registered to receive session or request destruction events MUST NOT be notified.
10. The Restartable task facility MUST create and start new threads for the tasks that were still registered when power was lost.
11. The containers may now process incoming requests over the I/O interfaces and dispatch requests to applications.

5.3 I/O Interface Reset

When an I/O Interface is logically reset, all ongoing communication sessions on that interface are terminated. In addition, specific protocol requirements of each physical interface reset must be followed.

5.3.1 ISO 7816-4 Reset

When an I/O interface which supports the ISO 7816-4 APDU protocol is reset, the applet container **MUST** perform the following actions in sequence to ready itself for accepting APDU commands:

1. The APDU object, if any, opened for that I/O interface **MUST** be placed in `STATE_ERROR_IO` state and **MUST** throw an `APDUException` exception with reason code `APDUException.IO_ERROR` upon use.
2. The applet thread, if any, dispatched to the currently selected applet instance **MUST** be terminated. The transaction in progress in the currently selected applet instance on that I/O interface, if any, **MUST** be aborted. Monitors for objects owned by the applet thread **MUST** be unlocked.
3. All applet instances that were active on that I/O interface at the time of reset are deselected.

If the I/O interface is being reset with the card fully powered up, meaning either initiated by the card itself or on programmatic request, each active instance is explicitly deselected via the appropriate `deselect` method (see [Section 4.3.1.4, “Applet Application Deselection” on page 4-12](#)). Otherwise, the active applet instances are implicitly deselected.

4. `CLEAR_ON_DESELECT` Transient data associated with each applet instance that was implicitly deselected is reset to the default value, if no applet instances from the application group are active on the card.
5. If default applet selection is implemented, the default applet for the basic logical channel (channel 0) for that I/O interface is selected as the active applet instance for the basic logical channel (channel 0) on that I/O interface, and the default applet's appropriate `select` method is called (see [Section 4.3.1.2, “Applet Application Selection” on page 4-10](#)). Otherwise, the Java Card RE sets its state to indicate that no applet is active on the basic logical channel of that I/O interface.

5.3.2 Transmission Control Protocol (TCP) Reset

When an I/O interface which supports the TCP protocol is reset, the Java Card RE **MUST** perform the following actions in sequence:

1. Any TCP/IP connections sustained over that I/O interface are terminated.
2. Any GCF connection objects opened over that I/O interface **MUST** be placed in a “closed” state.

3. If a Power Up or Card Reset is not in progress - the web container **MUST** perform the following actions in sequence to ready itself for accepting new HTTP requests on the ports of the I/O interface statically or dynamically allocated to the currently loaded web applications
 - a. Any listeners registered to receive session or request destruction events are notified.
 - b. All requests received on the “closed” connection are destroyed.
 - c. All active sessions associated with the “closed” connection objects **MUST** be deleted.

5.4 Concurrently Active Interfaces

When the card is concurrently operating over multiple interfaces, one interface may be reset, physically or logically, without affecting the other interfaces. Applications which were communicating concurrently over multiple interfaces may need to track the termination of one of the connections.

Security and Access Control Mechanisms

This chapter describes the following elements of security and access control:

- [Security Policy](#)
- [Permission-based Security](#)
- [Role-based Security](#)
- [User Authentication and Authorization](#)
- [On-card Client Application Authentication and Authorization](#)
- [Security Requirements and Credential Management of Secure Communications](#)
- [Code Isolation](#)
- [Package Access Control](#)
- [Context Isolation Enhancements](#)

6.1 Security Policy

A security policy 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.

On the Java Card Platform, a security policy may be implemented using two complementary techniques:

- **Permission-based security** - Permission-based security allows for restricting access to protected system and library resources based on some of the characteristics of the application requesting the access.

- **Role-based security** - Role-based security allows for restricting access to protected application resources including web resources, SIO-based services and events based on some of the characteristics of the application requesting the access, such as its identity and the identity of the user on behalf of whom the access is requested.

6.1.1 Permission-based Security Policy

A permission-based security policy maps some of an application's characteristics to a set of permissions granted to the application. These characteristics include the type of application model implemented by an application and may include other characteristics of an application's code, such as its signer's credentials.

Each application model supported by the Java Card Platform requires, permits or precludes access to various system resources, which may differ from one application model to the other. For example, the web application model supports multithreading and allows for an application to spawn new threads. However, in some environments, web applications may be denied permission to spawn new threads so other applications and the web container itself are not negatively impacted. Conversely, classic applet applications are typically not thread-safe and run in an environment that is single-threaded for backward compatibility reasons. Classic applet applications must not be permitted to spawn new threads.

The *platform security policy* MUST define this mapping of application models to sets of permissions granted to applications implementing these application models. The platform security policy MUST define for each of the three application models supported on the Java Card Platform the set of permissions that guarantees the consistency and integrity of the applications implementing each of these application models. The platform security policy MUST define each of these sets of granted permissions as a set of included permissions and a set of excluded permissions. The set of excluded permissions guarantees that no additional permissions can be granted that may violate the platform security policy. The platform security policy MAY be tuned for a specific environment, provided the consistency and integrity of each application model and of the platform itself is guaranteed. See [Appendix A](#) for the default set of permissions mapped to each of the application models.

In addition to permissions granted to an application based on its application model, other permissions MAY be granted in accordance with the operational environment in which the application is deployed. These permissions may be granted on a per-application basis or to applications sharing the same characteristics. Such permissions may include, for example, the permission to use another application's service. The mapping from application characteristics to sets of permissions in accordance with the operational environment in which the application is deployed MAY be defined by what is referred to as the *card management security policy*.

Permission sets defined by the platform security policy and the card management policy constitute what is referred to as *protection domains*.

6.1.2 Role-based Security Policy

An application developer may express the logical security requirements of the application either declaratively through the definition of *security constraints* on certain resources (web resources only) or programmatically through the implementation of programmatic security checks within the code that manages the access to certain resources. These security constraints and programmatic security checks name the security roles permitted to access the protected resources.

A user or client security role is a logical grouping of users or client applications defined by the application developer or assembler. When the application is deployed, user and client roles MUST be mapped by a deployer (for example, the application provider) to actual user identities and client application identities or characteristics on the targeted platform. Such a mapping is referred to as an *application security policy*. An application security policy MUST be defined by an application deployer at deployment-time in the runtime descriptor of the application.

6.1.3 Effective Application Security Policy

The overall security policy in effect for a particular application is the combination of the three security policies:

- the platform security policy
- the card management security policy
- the application security policy defined for the application when it was deployed.

The effective authorization of a client application to use services provided by a server application results from the combination of the application client's platform and card management security policies, and the server application's application security policy. That is, the client application must be granted the permission to interact with the server application by its own platform and card management security policies, plus the client application must be granted an authorized role of the server application.

6.2 Permission-based Security

6.2.1 Permissions

Permissions represent 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 `java.security.Permission` class. All permissions have a name or target name and may have an actions list. The format and interpretation of the name and actions list are specific to the permission classes.

Each permission concrete class MUST implement the `implies` abstract method of the `Permission` class to compare permissions. *Permission p1 implies permission p2* means that if an application is granted permission p1, it is naturally granted permission p2. This comparison MUST take into account all the attributes of a permission, including its name or target name and, if defined, its actions list.

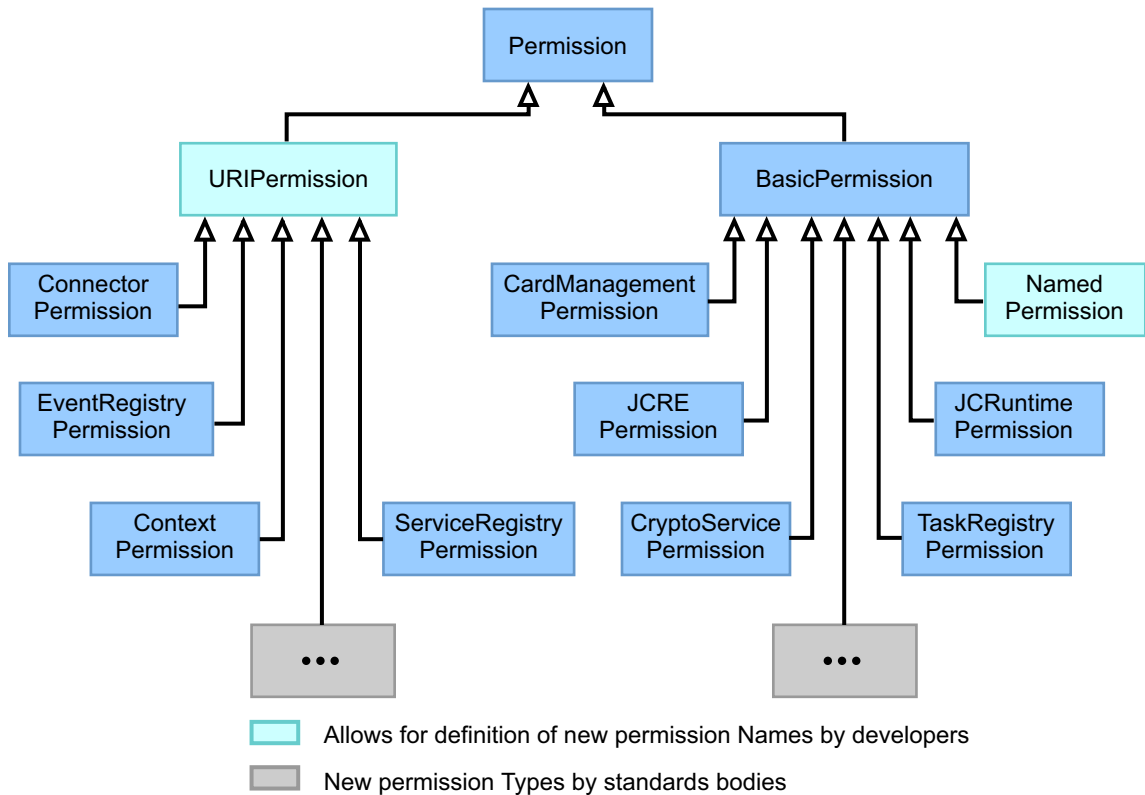
Permission objects are used to define the security policy. These permission objects are created when the platform security policy and the card management security policy are loaded and/or updated. Permission objects are also constructed whenever a permission to access a specific protected resource (the requested permission) must be checked against the permission granted by the security policies.

On the Java Card Platform, permission can be categorized into two types:

- named permissions, which have a name (one from a predefined list of names) but no actions list
- URI-named permissions, which identify the protected resource with a target URI and may additionally have an actions list identifying the permitted actions on the targeted resource

FIGURE 6-1 depicts the hierarchy of permission classes, the named permissions and the URI-named permissions as well as the two ways of defining new permissions.

FIGURE 6-1 Permission Class Hierarchy



6.2.1.1 Named Permissions

A Named Permission (or Property-named Permission) contains a name designating the resource or collection of resources to protect. A named permission has no actions list; the named permission is either granted or not. The named permission classes are concrete subclasses of the `java.security.BasicPermission` class. They follow the hierarchical property naming convention of `BasicPermission` and allow for the use of name-prefix patterns (a property-like name with a trailing asterisk such as "crypto.*").

All named permission classes **MUST** be subclasses of the `BasicPermission` class. See [Section 6.2.1.3, "Definition of New Permissions"](#) on page 6-11.

The `implies` method of named permission classes **MUST** implement the following steps:

1. Check whether the compared permission instance passed as parameter is of the same class type as the comparing permission instance (both MUST be direct instances of the same class).
2. Check whether the compared permission name is implied by the name of the comparing permission.

The permission names comparison MUST take into account wildcards so that a trailing wildcard name component in the comparing permission's name matches any trailing component of the compared permission name (including a wildcard).

TABLE 6-1 gives the list of named permission classes and their descriptions.

TABLE 6-1 Named Permission Classes

Permission Class	Permission Description
<code>javacardx.spi.framework.JCREPermission</code>	This class is for Java Card runtime environment permissions. It MUST protect the following functionality: <ul style="list-style-type: none"> • designating objects as Java Card runtime environment entry point objects (EPO) and global arrays. See Section 6.2.5.2, "Context-switch-triggered Security Checks" on page 6-20.
<code>javacardx.framework.JCRuntimePermission</code>	This class is for Java Card runtime permissions. It MUST protect the following functionality: <ul style="list-style-type: none"> • setting and getting of the credential manager(s) used for secure connections and accesses. See Section 6.6, "Security Requirements and Credential Management of Secure Communications" on page 6-49. • creating and modifying threads. See Section 2.7, "Multithreading" on page 2-35.
<code>javacardx.security.NamedPermission</code>	This class is intended for application- and library-defined named permissions. See Section 6.2.1.3, "Definition of New Permissions" on page 6-11.

TABLE 6-1 Named Permission Classes (*Continued*)

Permission Class	Permission Description
<code>javacardx.facilities.TaskRegistryPermission</code>	This class is for restartable task registry access permissions. It MUST protect the following functionality: <ul style="list-style-type: none"> • registering and unregistering of restartable tasks. See Section 2.10, “Restartable Tasks” on page 2-55.
<code>javacardx.security.CryptoServicePermission</code>	This class is for cryptographic service access permissions. It MUST protect the following functionality: <ul style="list-style-type: none"> • retrieving instances of specific cryptographic services. See the API and SPI specification for <code>javacardx.crypto</code>, <code>javacard.security</code>, <code>javacardx.security</code> and <code>javacardx.spi.security</code>.
<code>javacardx.spi.cardmgmt.CardManagementPermission</code>	This class is for card management permissions. It MUST protect the following functionality: <ul style="list-style-type: none"> • loading and unloading of application deployment units. See Section 8.6, “Loading Application Modules” on page 8-25 and Section 8.7, “Loading Libraries” on page 8-29. • creating and deleting application instances. See Section 8.8, “Creation of Application Instances” on page 8-31 and Section 8.9, “Deletion of Application Instance” on page 8-33. • retrieving the protection domain of applications. See Section 6.2.2, “Protection Domains” on page 6-13. • retrieving the credential manager of applications. See Section 6.6, “Security Requirements and Credential Management of Secure Communications” on page 6-49.

6.2.1.2 URI-named Permissions

A URI-named Permission contains a URI target name designating a resource or collection of resources to protect and an optional actions list. The URI-named permission classes are concrete subclasses of the base `javacardx.security.URIPermission` class. They follow the URI naming convention of `URIPermission` to designate resources or collections of resources and allow for the use of a path-prefix URI patterns.

The target name represents the URI of a resource, such as an application, an SIO-based service, an event, a web resource, a file or any other type of resource. See [Section 2.3, “Unified Naming and Dedicated Application Namespaces”](#) on page 2-5.

The target name may be either:

- an exact pattern, which designates exactly one resource,

- a path-prefix pattern ending with a `"/"`, which may designate a collection of resources,
- an application-generic-path pattern starting with `"/~"`, which may designate a resource or collection of resources in the current application's namespace,
- an authority pattern:
 - a registry-based authority pattern whose registry name equals `"*"`,
 - a server-based authority pattern whose DNS hostname, if one is specified, starts with `"*."` or equals `"*"` or whose port number, if one is specified, equals `"*"`,
- or any combination of the above.

For example, `sio:///transit/pos/ticketbook` designates the specific shareable interface object identified by the `sio:///transit/pos/ticketbook` URI; the `sio:///transit/pos/*` designates all the shareable interface objects identified by a URI starting with `sio:///transit/pos/`. The URI `sio:///~/ticketbook` designates the specific shareable interface object identified by the `sio:///transit/pos/ticketbook` URI when in the context of application `///transit/pos`. The URI `http://*.sun.com:*/*` designates any resource on any host in the `sun.com` DNS domain accessible via the HTTP protocol on any port.

Note – The root of the namespace designated by the path prefix URI pattern **MUST NOT** be itself included in the set of resources designated.

The actions to be granted are passed to the constructor in a string containing a list of one or more comma-separated keywords (whitespaces surrounding commas are ignored). The possible action keywords are application or library-specific, but the semantic of each action keyword **MUST NOT** overlap, because an action **MUST NOT** imply another action¹.

The URI-named permission classes **MUST** implement the following requirements:

- All resource URI matching operations **MUST** apply on canonicalized forms of the target resource URIs, that is, URIs **MUST** have first been resolved, then normalized.
- Resource URIs passed as a parameter to the permission classes' constructors may be absolute or relative. If a resource URI is relative, it **MUST** be resolved against the application's root URI that applies. For example, the relative service URI `ticketbook` may be resolved to `sio:///transit/pos/ticketbook` in the context of the application `///transit/pos`.

1. This restriction allows for uniformly managing URI-named permissions without requiring that each permission class be separately handled according to its own semantic.

- Resource URIs may contain “.” or “...” path components and MUST, therefore, be normalized. For example, `sio:///transit/pos/foo/bar/../../../../ticketbook` must be rewritten as `sio:///transit/pos/ticketbook`.
- The actions string MUST be canonicalized before processing, that is, it is converted to lowercase and sorted in lexical order.

The `implies` method of URI-named permission classes MUST implement the following steps:

1. Check whether the compared permission instance passed as parameter is of the same class type as the comparing permission instance (both permissions MUST be direct instances of the same class).
2. Check whether the compared permission's URI is implied by the URI of the comparing permission.
3. Check whether the compared permission actions list is implied by the actions list of the comparing permission.

The permission URIs comparison MUST take into account wildcards so that a trailing wildcard path component in the comparing permission's URI path matches any trailing component of the compared permission URI (including a wildcard).

[TABLE 6-2](#) gives the list of URI-named permission classes and their descriptions.

TABLE 6-2 URI-named Permission Classes

Permission Class	Permission Description
<code>javacardx.io.ConnectorPermission</code>	<p>This class is for GCF connector permissions. It MUST protect the following functionalities:</p> <ul style="list-style-type: none">• opening client connections to a designated server.• listening to client connections on designated a server connection endpoint.• accepting client connections on a designated server connection endpoint.• reading from a designated file connection input stream.• writing on a designated file connection output stream. <p>See the API specification for <code>javax.microedition.io</code> and <code>javacardx.io</code>.</p>
<code>javacardx.framework.ContextPermission</code>	<p>This class is for firewall-enforced context permissions. It MUST protect the following functionalities:</p> <ul style="list-style-type: none">• switching to another designated application context. See Section 6.2.5.2, “Context-switch-triggered Security Checks” on page 6-20.• transferring the ownership of an object to another designated application context. See Section 7.2.2, “Transferring Object Ownerships” on page 7-5.

TABLE 6-2 URI-named Permission Classes (*Continued*)

Permission Class	Permission Description
<code>javacardx.facilities.EventRegistryPermission</code>	<p>This class is for event registry access permissions. It MUST protect the following functionalities:</p> <ul style="list-style-type: none">• registering a listener for a designated event.• unregistering the listener of a designated event.• notifying event listeners of a designated event. <p>See Section 7.4, “Events” on page 7-16.</p>
<code>javacardx.facilities.ServiceRegistryPermission</code>	<p>This class is for SIO-based service registry access permissions. It MUST protect the following functionalities:</p> <ul style="list-style-type: none">• registering a factory for a designated SIO-based service.• unregistering the factory of a designated SIO-based service.• looking up a designated SIO-based service. <p>See Section 7.3, “Shareable Interface Object-based Services” on page 7-7.</p>
<code>javacardx.security.URIPermission</code>	<p>This class extends the <code>Permission</code> class and is the base class of all URI-named permission classes. It MUST be used as the base class for all permissions that use URIs (Uniform Resource Identifiers) as target names. See Section 6.2.1.3, “Definition of New Permissions” on page 6-11.</p>

6.2.1.3 Definition of New Permissions

New permissions may be defined either by defining new permission names for already defined permission classes or by defining new permission types extending existing permission base classes.

Defining New Permission Names

An application or library developer may define new named or URI-named permissions by defining new names, target name URIs and actions lists for `NamedPermission` and `URIPermission` objects.

The name attribute of the `NamedPermission` class is not constrained and allows any name to be used. The semantic of the names used depends on the context where they are defined and especially on the resources protected. Names of

NamedPermission objects should uniquely designate the resources they protect. It is suggested that the names of NamedPermission objects be prefixed in accordance with the reverse domain name convention for package naming suggested by *The Java Programming Language Specification*. The prefixes `javacard` and `javacardx` are reserved by this specification.

The URI name and actions list attributes of the `URIPermission` class are not constrained and allow for any URI and URI scheme, and any actions list, to be used. The URI target name **MUST** uniquely designate the resource or collection of resources to protect. The semantic of the actions used **SHOULD** reflect permissible operations on these resources.

[TABLE 6-3](#) lists the naming convention of application- and library-defined permissions a Java Card Platform **MAY** support.

TABLE 6-3 Application- and Library-defined Permissions

Permission Description	Permission Name	Permission Actions List	Permission Class
Named permission	Property name format with optional trailing wildcard (name pattern), for example: <ul style="list-style-type: none">• <code>transit.pos.credit</code>• <code>transit.pos.*</code>	None	<code>javacardx.security.NamedPermission</code>
URI-named permission	URI format with optional trailing wildcard (path-prefix URI pattern), for example: <ul style="list-style-type: none">• <code>sio:///transit/pos/ticketbook</code>• <code>sio:///transit/pos/*</code>	Free-form names: <code>credit</code> , <code>debit</code>	<code>javacardx.security.URIPermission</code>

Defining New Permission Types

New permission types **MAY** be defined by standards bodies when defining standard application or framework libraries for the Java Card Platform. These new permission classes **MUST** be subclasses of the `BasicPermission` or `URIPermission` classes. It is suggested that these permissions:

- be defined in the Java programming language package of the resource or functionality they protect
- define actions whose semantics do not overlap
- be declared `final`
- follow the same naming conventions as described in [TABLE 6-3](#).

Additionally, these new permission classes **MUST** be immutable.

See [Section 6.2.2, “Protection Domains” on page 6-13](#) for the implications of defining new permission types on the protection domain.

6.2.2 Protection Domains

A protection domain is a set of permissions granted to an application or group of applications. A protection domain is bound to a group context whose applications are granted the set of permissions. The set of permissions of the protection domain is determined according to the security policy that applies to the application or group of applications.

An *application’s effective protection domain*, or *application protection domain* for short, is the combination of:

- **Permissions granted by the platform security policy** - The *platform security policy* defines a set of permissions granted to an application according to some of the application’s characteristics, such as the application model it implements. This set of permissions is named a *platform protection domain*. It is common to all applications to which it applies and remains unchanged throughout the lifetime of these applications.
- **Permissions granted by the card management security policy** - the set of permissions additionally granted to the application according to the *card management security policy* in effect. The card management security policy may define an additional set of permissions granted to an application in accordance with the operational environment in which the application is deployed. These permissions may be granted on a per-application basis or to applications sharing the same characteristics. Such permissions may include, for example, the permission to use another application’s service. This set of permissions may be updated by the card manager applications throughout the lifetime of the application to account for changing privileges.

Each application’s group context **MUST** be bound to one and only one protection domain, its application protection domain.

An application protection domain **MUST** be bound to a platform protection domain. The effective set of permissions granted by an application protection domain **MUST** account for the permissions granted as well as the permissions explicitly denied by the platform protection domain it is bound to, as per the procedure for checking permissions described in [Section 6.2.4, “Checking of Permissions” on page 6-17](#).

An application protection domain **MUST** be updatable; that is, permissions may be added to or removed from the set of granted permissions during the lifetime of an application.

Updates to an application protection domain **MUST NOT** affect the protection domain of other applications in a different group context.

Updates to an application protection domain MUST NOT result in a set of permissions that violate the platform security policy that applies to that application (see also [Section 6.2.4, “Checking of Permissions” on page 6-17](#)):

- Any attempt to add to an application protection domain permissions that are *implied* by the set of excluded permissions defined for the platform protection domain of that application MUST result in a security exception being thrown.
- Removal of permissions from an application protection domain MUST be limited to permissions of that application protection domain and MUST NOT result in any alteration of the definition of the platform protection domain of that application.

An application protection domain MUST support being irreversibly locked to prevent further update of its permission set.

A protection domain MUST only allow for the following permission objects to be added to its permission set:

- direct instances of the following `Permission` subclasses:
 - `javacardx.framework.JCRuntimePermission`
 - `javacardx.security.NamedPermission`
 - `javacardx.facilities.TaskRegistryPermission`
 - `javacardx.io.ConnectorPermission`
 - `javacardx.framework.ContextPermission`
 - `javacardx.facilities.EventRegistryPermission`
 - `javacardx.facilities.ServiceRegistryPermission`
 - `javacardx.security.URIPermission`
 - `javacardx.security.CryptoServicePermission`
 - `javacardx.spi.cardmgmt.CardManagementPermission`
 - `javacardx.spi.framework.JCREPermission`
- direct instances of permission classes, subclasses of `BasicPermission` and `URIPermission`, that MAY be defined by standards bodies when defining standard application or framework libraries for the Java Card Platform as defined in [Section 6.2.1.3, “Definition of New Permissions” on page 6-11](#) and that a Java Card Platform implementation MUST explicitly allow for.

The use of `JCREPermission` MUST be restricted to the definition of platform protection domains, see [Section 6.2.2.1, “Platform Protection Domains” on page 6-16](#).

Any attempt to add to an application protection domain instances of permission classes, other than the one listed above or indirect instances of the classes listed above, MUST result in a security exception being thrown. Any attempt to add to an application protection domain an instance of `JCREPermission` MUST result in a security exception being thrown.

An application protection domain may be assigned a name. Protection domain names are not constrained and may not be unique, but **MUST** be different from the names reserved for protection domains defined by the platform security policy, see [Section 6.2.2.1, “Platform Protection Domains” on page 6-16](#). The name of a protection domain may be used by context switch permissions and role-based security for inter-application communications, see [Section 6.2.5.2, “Context-switch-triggered Security Checks” on page 6-20](#) and [Section 6.5, “On-card Client Application Authentication and Authorization” on page 6-46](#).

A Java Card Platform implementation **MAY** internally create Java Card RE-owned copies² of the permission objects to be added to a protection domain and only add these copies to the protection domain. In such a case, subsequent management operations (add, remove, enumerate, check) **MAY** be performed on these Java Card RE-owned copies.

A Java Card Platform implementation **MAY** internally use any optimized representations to manage (add, remove, enumerate, check) permissions in a protection domain, but **MUST** manage permission sets as collections of permission objects (allowing duplicates) with respect to adding and removing permissions³. Add and remove operations on a collection of permission objects **MUST** be implemented in terms of the `Permission.equals()` method⁴. Especially, permissions **MUST** be removed with the same granularity in which they were added, regardless of the possible compositions of permissions of the same types. For example, if the two following permissions:

- `ServiceRegistryPermission("/transit/*", "lookup")`
- `ServiceRegistryPermission("/transit/pos/ticketbook", "register, unregister")`

had been added to a protection domain, attempting to remove the permission `ServiceRegistryPermission("/transit/pos/ticketbook", "register")` or `ServiceRegistryPermission("/transit/pos/ticketbook", "lookup")` must not result in the protection domain's set of permissions being changed because it does not actually contain any of the permission objects to be removed.

Access to the class or classes encapsulating the characteristics of protection domains **MUST** be protected by package access control builtin checks as described in [Section 6.8.1, “Built-in Checks” on page 6-63](#) and **MUST** be restricted to card management applications. Retrieving the protection domain of an application from its group context **MUST** be subject to access control: a security check **MUST** ensure that the permission `javacardx.spi.cardmgmt.CardManagementPermission` with the target name `protectionDomain.get` is granted.

2. Equivalent in terms of the `Permission.equals()` method.

3. Permission sets defined by the platform security policy cannot be updated, therefore, they may not be subject to this constraint.

4. Not in terms of the `Permission.implies()` method.

6.2.2.1 Platform Protection Domains

The platform security policy defines for each of the three application models supported on the Java Card Platform, a platform protection domain, which is a set of permissions that guarantees the consistency and integrity of the applications implementing each of these application models. Each of these platform protection domains defines the minimum set of permissions granted to an application of the corresponding type. Additionally, the platform security policy defines a platform protection domain for the card management applications.

Each of these platform protection domains MUST be uniquely identified with a name.

Each of these platform protection domains MUST be defined with a set of *included permissions* and a set of *excluded permissions*.

The default platform protection domains defined below MAY be tuned for specific environments according to the following rules:

- Permissions MUST NOT be added to or removed from the excluded permission set defined for the corresponding default protection domain.
- Permissions MAY be added to or removed from the included permission set defined for the corresponding default protection domain.

Platform protection domains MUST be locked to prevent any post-issuance update.

Default Platform Protection Domain for Web Applications

The default protection domain for web applications MUST be named “Web”. It MUST include the set of included permissions listed in [Appendix A, TABLE A-1](#) and the set of excluded permissions listed in [Appendix A, TABLE A-2](#).

Default Platform Protection Domain for Extended Applet Applications

The default protection domain for extended applet applications MUST be named “Extended”. It MUST include the set of included permissions listed in [Appendix A, TABLE A-3](#) and the set of excluded permissions listed in [Appendix A, TABLE A-4](#).

Default Platform Protection Domain for Classic Applet Applications

The default protection domain for classic applet applications MUST be named “Classic”. It MUST include the set of included permissions listed in [Appendix A, TABLE A-5](#) and the set of excluded permissions listed in [Appendix A, TABLE A-6](#).

Default Platform Protection Domain for Card Management Applications

The default protection domain for card management applications MUST be named “CardManagement”. It MUST include the set of included permissions listed in [Appendix A, TABLE A-7](#) and the set of excluded permissions listed in [Appendix A, TABLE A-8](#).

6.2.3 Assigning Permissions

A group context is bound to one application protection domain. All the applications in that group context are granted the same set of permissions.

Prior to the first application being instantiated in a group context, the assignment of permissions to the application protection domain MUST proceed as follows:

1. The platform protection domain MUST be determined, based on the type of applications to be instantiated, and bound to the group context.
2. The application protection domain MUST be created with an optional name and bound to that platform protection domain.
3. The application protection domain MAY be assigned additional permissions as per the card management policy in effect.
4. The group context MUST be bound to the application protection domain.

Permissions MAY be added and removed from the set of permissions initially added as per the card management policy at any time, provided the application protection domain is not locked.

6.2.4 Checking of Permissions

Checking if a permission is granted to an application by its application protection domain MUST follow the following steps:

1. Check if the permission is *implied* for all of its actions, if any are specified, by the set of permissions resulting from the union⁵ of:
 - the set of included permissions defined for the platform protection domain
 - the set of permissions added to the application protection domain, as per the card management in effect.

⁵Checking of permissions must account for permissions spanning the platform protection domain’s set of included permissions and the set of permissions added to the application protection domain, as per the card management security policy in effect.

2. Check if the permission is **not implied** for any of its actions, if any are specified, by the set of excluded permissions defined by the platform protection domain.

6.2.4.1 Permission Composition

When checking if a requested permission is implied by a set of permissions, the requested permission may not correspond to a single permission object in the permission set but may be implied by a composition of several permissions.

A URI-named permission object may not only include a URI target name but also a list of actions. When a permission set contains several permissions of the same type which protect the same resource either by directly designating the resource or by designating a namespace that encompasses the resource, the actions of all the permissions that protect the resource **MUST** be taken into account to determine the effective permission on that resource.

For example, if a permission set contains the two following permissions:

- `ServiceRegistryPermission("/transit/*", "lookup")`
- `ServiceRegistryPermission("/transit/pos/ticketbook", "register, unregister")`

and the permission `ServiceRegistryPermission("/transit/pos/ticketbook", "register, lookup")` is being checked, then the effective permission against which the checked permission must be compared is `ServiceRegistryPermission("/transit/pos/ticketbook", "register, unregister, lookup")`.

Note – Using the `implies` method of a permission object is not adequate for checking a permission with multiple actions against a permission set when the permission actions to be checked span several permission objects in the set. In the previous example, none of the permissions in the permission set individually implies the permission being checked; therefore, the `implies` method of an individual permission object is not adequate. A simple solution that allows for using the `implies` method of permission objects is to break the permission being checked into an equivalent set of single-action permissions (for example, `ServiceRegistryPermission("/transit/pos/ticketbook", "register")`, `ServiceRegistryPermission("/transit/pos/ticketbook", "lookup")`). All these single-action permissions must be implied by at least one permission in the set of permissions for the permission being checked to be granted by that set.

6.2.5 Security Policy Enforcement

The security policy is enforced by two types of security checks:

- **context-switch-triggered security checks** - which are automatically initiated, see [Section 6.2.5.2, “Context-switch-triggered Security Checks” on page 6-20](#).
- **programmatically security checks** - which are programmatically initiated, see [Section 6.2.5.3, “Programmatically Security Checks” on page 6-22](#).

Each of these two types of security checks implements the same principle for deciding whether to grant access to protected resources.

6.2.5.1 Access Control Decision Principle

An access control decision **MUST** be made based on the group context of the immediate application requesting the access:

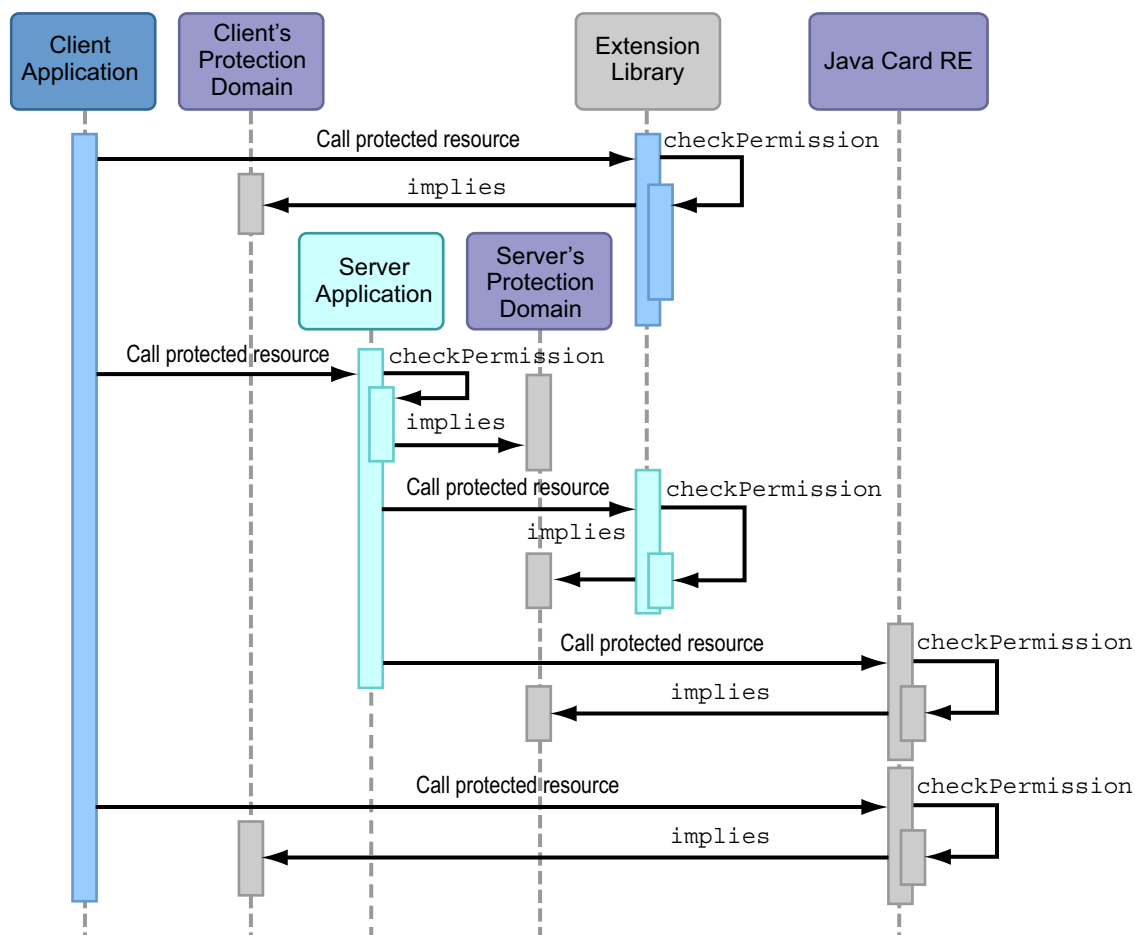
- When the currently active context (see [Section 2.7.2.2, “Per-Thread Active Context and Active Namespace” on page 2-38](#)) is a group context, the access control decision **MUST** be made based on--and solely based on--the protection domain associated with the currently active group context.
- When the currently active context is the Java Card RE context, the access control decision **MUST** be made based on--and solely based on--the protection domain associated with the previously active context (the calling application’s group context). If there is no previously active context, the access **MUST** be granted.

The protection domains of the other group contexts in the chain of calls **MUST NOT** be taken into account during an access control decision.

FIGURE 6-2 depicts the interaction between the different actors involved in a access control decision. Objects colors indicate common ownership of interacting objects. The colors of focus of control boxes⁶ indicate the currently active contexts as the results of specific calls.

6. In UML sequence diagrams, a focus of control is a rectangle that lies on an object lifeline and which depicts the duration of a particular call to that object.

FIGURE 6-2 Access Control Decision Principle (Sequence Diagram)



6.2.5.2 Context-switch-triggered Security Checks

When a thread calls a method of an object owned by a group context different from its active context, the Java Card virtual machine performs a context switch. During the context switching method invocation, the current active context is saved and the new context (group context or Java Card RE context) becomes the active context of the currently executing thread.

After the Java Card virtual machine has determined that a context switch must be performed, but prior to performing it, the Java Card virtual machine **MUST** perform one of two security checks depending on the new context:

- if the new context is the Java Card RE context, a security check **MUST** decide whether the application protection domain associated with the current active context grants the permission to call the Java Card RE-owned object, that is, if the object being called is an authorized Java Card RE entry point object (EPO) or a global array.
- if the new context is not the Java Card RE context, that is, the new context is an application's group context, a security check **MUST** decide whether the application protection domain associated with the current active context grants the permission to switch to the new group context.

Java Card RE Entry Point Object Security Check

This security check **MUST** ensure that the permissions in the applicable protection domain (as defined in [Section 6.2.5.1, “Access Control Decision Principle” on page 6-19](#)) *imply* the permission `javacardx.spi.framework.JCREPermission` with a name designating the class name of the temporary (respectively permanent) Java Card RE EPO or global array object according to the following format `callTempJCREEPO.{class name}` (respectively `callPermJCREEPO.{class name}`).

[TABLE 2-3](#) gives the *Classic* set of permanent Java Card RE EPO classes that is designated by the permission `javacardx.spi.framework.JCREPermission` with the special name `callPermJCREEPO.CLASSIC`.

[TABLE 2-4](#) gives the *Classic* set of temporary Java Card RE EPO classes that is designated by the permission `javacardx.spi.framework.JCREPermission` with the special name `callTempJCREEPO.CLASSIC`.

[TABLE 2-5](#) gives the *Extended* set of permanent Java Card RE EPO classes that is designated by the permission `javacardx.spi.framework.JCREPermission` with the special name `callPermJCREEPO.EXTENDED`.

[TABLE 2-7](#) gives the *Classic* set of *global array objects* that is designated by the permission `javacardx.spi.framework.JCREPermission` with the special name `callTempJCREEPO.CLASSIC`.

Context Switch Security Check

This security check **MUST** ensure that the permissions in the protection domain associated with the currently active context *imply* the permission `javacardx.framework.ContextPermission` with a target identifying the new group context and with the action `switch`.

The target identifying the new group context **MUST** designate one of the following:

- the name assigned to the protection domain associated with that new group context

- the name of the platform protection domain bound to the protection domain associated with that new group context
- the URI identifying an application in that new group context
- a standard service URI. This allows for indirectly designating an application that registers a standard application service without specifically naming the designated application.

No such security check **MUST** happen when switching to the Java Card RE context or switching from the Java Card RE context.

6.2.5.3 Programmatic Security Checks

Security checks may be programmatically initiated by the Java Card RE, libraries, and also applications to protect access to sensitive resources or check by anticipation if access to protected resources would be granted.

To programmatically check if an application is granted the permission to access a specific protected resource prior to access, the resource, an instance of the corresponding permission class, **MUST** be constructed and the `checkPermission` method of the `java.security.AccessController` class **MUST** be invoked. The `checkPermission` method of the `AccessController` class **MUST** determine whether the access request indicated by the permission should be granted or denied, based on the protection domain associated with the group context of the application. The specified permission **MUST** be granted if the access controller determines that the permissions of the protection domain *imply* the requested permission.

The permissions required for access are specified for each protected resource or functionality by the associated API specification. These **MUST** be enforced by an implementation of the Java Card Platform.

6.2.5.4 Platform Protection Domain Security Check Optimizations

Platform protection domains are not updatable post issuance. An implementation **MAY** implement checks for permissions granted by platform protection domains, such as for `javacardx.spi.framework.JCREPermission`, in any optimized way, including as built-in checks, as long as the result is semantically equivalent to that described in [Section 6.2.4, “Checking of Permissions” on page 6-17](#).

6.3 Role-based Security

Role-based security is used to restrict access to resources to only those users and client applications which have been granted a particular security role:

- For a user to be granted a particular security role, or, less formally, to belong to a particular user role, the user's identity must be determined by authentication with the platform. See [Section 6.4, "User Authentication and Authorization" on page 6-28](#) for more details on user authentication.
- A client application may be granted a particular security role, or, less formally, may belong to a particular client role, based on its identity as determined by authentication with the server application it wants to access or based on some of its characteristics, such as its identification on the platform (its application identifier). See [Section 6.5, "On-card Client Application Authentication and Authorization" on page 6-46](#) for more details on client application authentication.

An application's role-based security relies on the following:

- the definition, in terms of roles, of the application's security requirements through the definition of declarative security constraints on certain resources, externally to the application's code, and/or the implementation of programmatic security checks within the code which manages access to certain resources
- the mapping of these user and client roles to user identities and client application identities or characteristics, respectively.

A user or client security role is a logical grouping of users or clients defined by the application developer or assembler. When the application is deployed, roles **MUST** be mapped by a deployer (for example, the application provider) to user identities and client identities or client characteristics. If a role is not mapped, that role cannot be granted to any user or application client and, therefore, a resource exclusively protected by that role will not be accessible.

6.3.1 User Role-based Security

User role-based security may be used to control access both to SIO resources, such as SIO-based services and events, and to web resources:

- User role-based security for SIO-based services and events may only be implemented using programmatic security⁷.
- User role-based security for web resources may be defined using both declarative security and programmatic security:

7. Declarative means of specifying user authorization constraints on SIO resources in an application's descriptor are not supported at this time

- A web application developer may use declarative security to express security constraints on certain resources of a web application. Web security constraints are declared in the deployment descriptor of web applications for constraints on web resources.
- Programmatic security is used by security-aware web applications when declarative security alone is not sufficient to express the security model of the application.

6.3.1.1 Declarative User Role-based Security

A web application developer may use declarative security to express security constraints on certain resources of a web application. A security constraint designates:

- the constrained resources, meaning the resources to which the security constraint applies
- and an authorization constraint which establishes a requirement for authentication and names the user security roles permitted to access the constrained resources.

Security constraints are declared in the deployment descriptor of web applications for constraints on web resources.

See [Section 6.4.5.1, “Container Checking of User Authorization” on page 6-35](#) for more details on how the web container determines whether a user is authorized to access a protected web resource.

6.3.1.2 Programmatic User Role-based Security

Programmatic user role-based security consists of the following methods:

- the `isUserInRole` method of the `JCSystem` class
- the `isUserInRole` method of the `HttpServletRequest` interface

The `JCSystem.isUserInRole` method determines if an authenticated identity is in a specified security role as defined in the Java Card Platform-specific application descriptor for SIO and event resources. On a call to `JCSystem.isUserInRole`, the following steps MUST be followed:

1. If the authentication is session-scoped, that is, if there is an authenticated identity associated with the current thread, check that the URI of the authenticator associated with that authenticated identity matches one of the authenticator URIs to which the role was mapped. See [Section 6.3.1.3, “Role to User Mapping” on page 6-25](#) for details on the matching rules.

2. Otherwise, if the authentication is global, that is, there is no authenticated identity associated with the current thread, or if the authentication is session-scoped but the URI of the authenticator associated with the authenticated identity does not match any of the authenticator URIs to which the role was mapped, check that one of the authenticator URI to which the role was mapped corresponds to a globally authenticated user.

The `HttpServletRequest.isUserInRole` method determines if an authenticated identity is in a specified security role as defined in the web application's deployment descriptor for web resources. On a call to `HttpServletRequest.isUserInRole`, the following steps MUST be followed:

1. If there is an authenticated identity associated with an incoming request, check that the URI of the authenticator associated with that authenticated identity matches one of the authenticator URIs to which the role was mapped.
2. Otherwise, if there is no authenticated identity associated with the incoming request, false is returned.

Caution – The `HttpServletRequest.isUserInRole` method applies to roles defined for web resources, while the `JCSysstem.isUserInRole` method applies to roles defined for SIO and event resources.

6.3.1.3 Role to User Mapping

User security roles MUST be declared in the Java Card Platform-specific application descriptor and/or in the deployment descriptor of web applications. These roles MUST be mapped to authenticator URIs in the runtime descriptor of the application. See [Section 6.4.1, “Scheme-specific Authenticators” on page 6-30](#) for details on authenticator URIs.

The use of path prefix URIs is allowed under certain restrictions for designating a collection of session authenticator URIs. For example
`sio:///standard/auth/user/session/admin/*` may be used to designate
`sio:///standard/auth/user/session/admin/admin1/password`,
`sio:///standard/auth/user/session/admin/admin2/fingerprint` and
`sio:///standard/auth/user/session/admin/admin3/iris-scan`. Such path prefix authenticator URI patterns can only be used to designate authorized users where a user authorization can be checked by matching the path prefix URI pattern against the authenticator URI representing a user's identity.

The platform MUST enforce programmatic security for the authenticated identity associated with the client of an SIO-based service or the consumer of an event based on the corresponding authenticator URI. The web container MUST enforce declarative and programmatic security for the authenticated identity associated with an incoming request based on the corresponding authenticator URI.

The following rule must be enforced by the Java Card RE:

- If a deployer has mapped a security role to an exact authenticator URI (one which is not a path-prefix URI pattern), the authenticated identity is in the security role only if its associated authenticator URI matches exactly the authenticator URI to which the security role has been mapped by the deployer.
- If a deployer has mapped a security role to a session authenticator path-prefix URI pattern, the authenticated identity is in the security role only if its associated authenticator URI is matched by the session authenticator path-prefix URI pattern to which the security role has been mapped by the deployer.

The use of path-prefix URI pattern for global authenticator is not permitted. The platform **MUST** reject any application mapping security roles to a global authenticator path-prefix URI pattern.

The use of an authenticator path-prefix URI pattern is not permitted for security constraints on web resources defined in a web applications's deployment descriptor. The platform **MUST** reject any web application mapping security roles defined for security constraints on web resources to authenticator path-prefix URI patterns.

Note – The use of path-prefix URI pattern for global authenticator and for security constraints on web resources is restricted because it would require searching all matching authenticators in the registry, either to attempt to authenticate or check authentication.

If a user security role declared in the Java Card Platform-specific application descriptor or in the deployment descriptor of a web application is not mapped to any authenticator URI, no user can authenticate in that role and therefore access to resources protected by that role can not be authorized.

If a web application declares a user security role in its deployment descriptor but no login configuration or authentication method (`login-config` and `login-config/auth-method` elements), active authentication in that role by the container on behalf of that application will not be performed. Similarly, if the user security role is mapped to an authenticator URI with a `<scheme>` or `<realm>` path component that does not match the authentication method or realm name⁸ (`login-config/auth-method` and `login-config/realm-name` elements) declared for that web application, active authentication in that role by the container on behalf of that application will not be performed. If the user security role is mapped to a global card holder identity, access to resources protected by that role may still be authorized regardless of the authentication scheme or realm name, provided the global card holder identity has already been authenticated either programmatically by the application or, by or on behalf of another application. This behavior is enforced by the algorithm described in [Section 6.4.5.1, “Container Checking of User Authorization”](#) on page 6-35.

8. When the `login-config/realm-name` element is not present an empty realm name is assumed.

Note – If the user security role is mapped to a **session-scoped authenticator URI** with a *<scheme>* or *<realm>* path component that does not match the authentication method or realm name declared for that web application, not only active authentication in that role by the container on behalf of that application will never be performed but access to resources protected by that role will never be authorized neither. Tools used to package web applications SHOULD flag this kind of role mapping as an error.

See [Chapter 8](#) for more details on how to map roles to users in the runtime descriptor of the application.

6.3.2 Client Role-based Security

Client role-based security may be used to control access to SIO resources such as SIO-based services and events only. Client role-based security for SIO-based services and events may only be implemented using programmatic security⁹.

6.3.2.1 Programmatic Client Role-based Security

Programmatic client role-based security consists of the `isClientInRole` method of the `JCSysSystem` class.

The `JCSysSystem.isClientInRole` method determines if an application client is in a specified security role. A call to this method MUST check that one of these conditions exist:

- if the client role is mapped to one or several application URIs or application path-prefix URI patterns, the client application URI is matched by one of the URIs to which the role is mapped
- or, if the client role is mapped to one or several protection domain names, the client application's application protection domain name or platform protection domain name is matched by one of the protection domain names to which the role is mapped
- or, if the client role is mapped to one or several credential aliases, the client application is authenticated using one of the credentials corresponding to the aliases to which the role is mapped. See [Section 6.5, “On-card Client Application Authentication and Authorization”](#) on page 6-46 for more details on client application authentication.

9. Declarative means of specifying client authorization constraints on SIO resources in an application's descriptor are not supported at this time

Client role-based security **MUST** not apply among applications within the same group context. If the client application (the caller of the SIO) is in the same group context as the server application `isClientInRole` **MUST** return `true`.

6.3.2.2 Role to Client Mapping

Client security roles **MUST** be declared in the Java Card Platform-specific application descriptor. These roles **MUST** be mapped in the runtime descriptor of the application to one of the following:

- One or several application URIs or application path-prefix URI patterns, designating specific authorized client applications or collections of authorized client applications, respectively.
- One or several protection domain names, designating the platform protection domains or the application protection domains of authorized client applications.
- One or several credential aliases, designating the credentials of authorized application clients.

See [Chapter 8](#) for more details on how to map roles to clients in the runtime descriptor of the application.

6.4 User Authentication and Authorization

User authentication is the process by which a user proves his or her identity to the card. This authenticated identity is then used by the platform to perform authorization decisions for accessing protected resources such as web resources and SIO-based and event-based services.

Users can be categorized into two types:

- card holder, who is the primary user of the card
- other user, such as a remote card administrator

To both these types of users may correspond different user identities on the card:

- card holder user identities, noted as *card-holder-users* in the rest of this chapter
- other user identities, noted as *other-users* in the rest of this chapter

Additionally, applications can be categorized into two types:

- locally accessible application, which may directly prompt the card holder for authentication when needed
- remotely accessible application, which cannot directly prompt the card holder for authentication when needed

See [Section 6.4.6, “Card Holder Authorization For Remotely Accessible Applications” on page 6-40](#) for how the container MUST distinguish between these two types of applications.

The scope of user authentication can be categorized into two types:

- global authentication, restricted to card-holder-users
- session-scoped authentication of both card-holder-users and other-users

Card-holder-user authentication can be tracked globally (card-wide). Authorization to access resources protected by a globally authenticated card-holder-user identity is granted to all users. This may be viewed as the card holder opening the card for a certain usage, thereby allowing certain protected resources to be accessed by any users regardless of their actual identities.

Because several conversational sessions can be established simultaneously between on-card applications and web clients, web user authentication can also be 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. Such a situation would, otherwise, arise if global authentication were used to authorize access from different web clients, even though different identities were used.

The Java Card RE implementation MUST support tracking the state of card-holder-user authentication globally (card-wide) and on a per-session basis (session-scoped), whereas the Java Card RE implementation MUST support tracking the state of other-users' authentication only on a per-session basis. In both cases, authentication is delegated to application-level authenticators for performing the actual authentication against a user's credentials according to a specific scheme (such as PIN-based, biometric, or password-based).

Additionally, an application may either manage (meaning initiate, check and reset) authentication of users directly, interacting directly with the user authentication facility, or may delegate authentication of users to the application container. Container-managed authentication is only supported by the web application container¹⁰. A web application may manage user authentication directly if the web application needs to manage authentication at a different granularity than the one provided by the web container.

Container-managed authentication of a card-holder-user MUST only be supported for applications identified in their Java Card Platform-specific application descriptor as locally accessible, see [Section 6.4.6, “Card Holder Authorization For Remotely Accessible Applications” on page 6-40](#). Application-managed authentication of a card-holder-user is not subject to this restriction, but application developers and application providers must be aware of the potential risks.

10. The applet-application container does not support container-managed authentication.

6.4.1 Scheme-specific Authenticators

The authentication facility relies on authenticators, a set of authentication services, to handle authentication according to different schemes. These authenticators **MUST** expose service interfaces which implement the `javacardx.framework.Authenticator` interface and which itself implements `javacard.framework.Shareable` interface. The corresponding SIOs **MUST** be registered in the SIO-based service registry under the standard SIO namespace. The authenticator SIO namespace is partitioned to discriminate between card-holder-users and other-users, and between global and session-scoped authentication. URIs in this namespace additionally distinguish both specific users being authenticated, as well as specific schemes used for authentication of these users. Authenticators may be registered and unregistered in and from the SIO-based service registry by any application having the required privileges, such as a card management application.

An authenticator URI **MUST** be one of three types:

- A global card-holder-user authenticator URI:

```
sio:///standard/auth/holder/global/[<realm>/]<user>/<scheme>
```

- A session-scoped card-holder-user authenticator URI:

```
sio:///standard/auth/holder/session/[<realm>/]<user>/<scheme>
```

- A session-scoped user authenticator URI:

```
sio:///standard/auth/user/session/[<realm>/]<user>/<scheme>
```

The `<user>`, `<scheme>` and `<realm>` path components of these URIs are place holders for specific user names, authentication schemes and authentication realm names respectively. The `<realm>` path components of these URIs may designate an arbitrary path.

For example, an authenticator URI for the global card-holder-user name `owner`, the authentication scheme `pin` and the realm name `/transit/pos` could be:

```
sio:///standard/auth/holder/global/transit/pos/owner/pin
```

Each authenticator URI represents a unique identity on the platform.

The uniqueness of user names (`<user>` path component of the authenticator URIs) across the different authenticator sub-namespaces is not enforced by the platform, meaning the same user name may be used for a global card-holder-user identity, a session card-holder-user identity, or even an other-user's identity. It is recommended, though, that the user management policy enforces the uniqueness of user names between card-holder-user identities and all other-user identities.

Note – Application-defined users are not supported. The platform **MUST** reject the application if an application’s declarative security policy references an authenticator URI outside the three namespaces for global card-holder-user, session-scoped card-holder-user and other-user authentication defined above. The platform **MUST** reject the application especially if an application’s declarative security policy references an authenticator URI inside its own namespace or another application’s namespace. This requirement centralizes the representation and management of identities on the platform.

Each authenticator **MUST** provide the following functionality:

- authenticate a user with provided credentials
- check whether or not a user is authenticated
- reset the authentication status of an authenticated user

For PIN-based authentication, the authenticator SIO and its URI **MUST** have the following properties:

- the `<scheme>` path component of the authenticator URI **MUST** be `pin`
- the authenticator SIO **MUST** implement the `javacardx.framework.SharedPINAuth` interface

For password-based authentication, the authenticator SIO and its URI **MUST** have the following properties:

- the `<scheme>` path component of the authenticator URI **MUST** be `password`
- the authenticator SIO **MUST** implement the `javacardx.framework.SharedPasswordAuth` interface

For biometric authentication, the authenticator SIO and its URI **MUST** have the following properties:

- the `<scheme>` path component of the authenticator URI **MUST** be one of the strings listed in the first column of [TABLE 6-4](#)
- the authenticator SIO **MUST** implement the `javacardx.framework.SharedBioTemplateAuth` interface; its `getBioType` method **MUST** return the bio type constant corresponding to the `<scheme>` path component of its URI as listed in the second column of [TABLE 6-4](#)

Additional schemes **MAY** be defined as long as they do not use these reserved names.

Since authenticators are plain SIOs, registration and invocation of authenticators are subject to the same access control policies as plain SIOs. For example, authentication as a particular identity may only be granted to a specific application and, then, only if a card-holder-user has already been globally authenticated. Authentication of the admin user `sio:///standard/auth/user/session/admin/password`, for

example, may only be authorized through the /admin web application if the owner card-holder-user `sio:///standard/auth/holder/global/owner/pin` is already authenticated.

TABLE 6-4 Biometric Scheme Name to BioBuilder Biometric Type Constant Mapping

Biometric Scheme Name	Biometric Type Constant
body-odor	BODY_ODOR
dna-scan	DNA_SCAN
ear-geometry	EAR_GEOMETRY
facial-feature	FACIAL_FEATURE
finger-geometry	FINGER_GEOMETRY
fingerprint	FINGERPRINT
gait-style	GAIT_STYLE
hand-geometry	HAND_GEOMETRY
iris-scan	IRIS_SCAN
keystrokes	KEYSTROKES
lip-movement	LIP_MOVEMENT
palm-geometry	PALM_GEOMETRY
retina-scan	RETINA_SCAN
signature	SIGNATURE
thermal-face	THERMAL_FACE
thermal-hand	THERMAL_HAND
vein-pattern	VEIN_PATTERN
voice-print	VOICE_PRINT

6.4.2 Global Authentication of Card Holders

For global authentication, the authenticator SIO instances returned by the SIO-based service registry for a particular URI MUST represent a single authentication state. The authenticator SIO may typically be implemented as a singleton.

Authenticators registered under the namespace `sio:///standard/auth/holder/global/` MUST support global card-holder-user authentication.

Upon successful authentication using a global authenticator, the authenticated identity represented by the URI of the global authenticator **MUST** remain authenticated until explicitly reset or invalidated by a failed re-authentication attempt or until a platform reset.

In order to determine if a card-holder-user is globally authenticated, the corresponding authenticator **MUST** be located and the authentication status **MUST** be checked. For example, to determine if the card-holder-user owner identified by the URI `sio:///standard/auth/holder/global/owner/pin` is authenticated, the PIN authenticator must be located and its `Authenticator.isValidated` method must be invoked.

An application developer may determine if a card-holder-user is globally authenticated by directly implementing this procedure or may use the `JCSysm.isUserInRole` method to determine if an authenticated card-holder-user identity is in a specified security role as defined in the Java Card Platform-specific application descriptor. See [Section 6.3.1.2, “Programmatic User Role-based Security” on page 6-24](#) for more details.

6.4.3 Session-scoped Authentication of Web Users

For session-scoped authentication, the authenticator SIO instance returned by the SIO-based service registry for a particular URI **MUST** represent a different authentication state for each session. This is typically achieved by returning different instances at each call. While the authentication state they manage must be different, all authenticator instances representing the same identity **SHOULD** share a common *try limit counter* so that failed attempts may be counted globally.

Upon successful authentication using a session authenticator, the authenticated identity represented by the URI of the session authenticator **MUST** be associated with the current authentication session and bound to the current thread. The same authenticated identity **MUST** be bound to any subsequent request-dispatching thread when entering any web application within the scope of that same authentication session.

Note – The exact definition of an authentication session depends on the authentication scheme but may be roughly defined as being demarcated by the prompting for the user’s credentials during the same conversational session between a client and the web application container.

Note – As per the *Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition*, the web application container is required to track authentication information at the container level (rather than at the web application level) in order to allow users authenticated for one web application to access other web resources managed by the container permitted to the same security identity. Therefore, the association between an authentication session and a session authenticator should be managed at the container level.

Session-scoped authentication **MUST** be supported for authenticators registered under the namespaces `sio:///standard/auth/holder/session/` and `sio:///standard/auth/user/session/`.

Authenticators registered under the namespace `sio:///standard/auth/holder/session/` **MUST** support session-scoped card-holder-user authentication.

Authenticators registered under the namespace `sio:///standard/auth/user/session/` **MUST** support session-scoped authentication for other-users.

In order to determine if a user (a card-holder-user or an other-user) is authenticated for the current session, the corresponding session authenticator URI and the authenticator URI bound to the current thread **MUST** be compared. For example, to determine if the user `admin` identified by the URI `sio:///standard/auth/user/session/admin/password` is authenticated, this URI must be compared with the authenticator URI bound to the current thread.

An application developer may not have direct access to the authenticator URI bound to the current thread. An application developer may use the `JCSystem.isUserInRole` method to determine if the currently authenticated identity is in a specified security role as defined in the Java Card Platform-specific application descriptor. See [Section 6.3.1.2, “Programmatic User Role-based Security”](#) on page 6-24 for more details.

6.4.4 Application-managed Authentication

The platform **MUST** allow an application to manage (validate, check and reset) user authentication by the application interacting directly with the user authentication facility.

Applications may locate authenticators in the SIO-based service registry using their URIs and may invoke the methods exposed by the authenticators to:

- authenticate a user with provided credentials
- check whether or not a user is authenticated

- reset the authentication status of an authenticated user

It is the responsibility of the application to properly demarcate the authentication session.

6.4.5 Web Container-managed Authentication

When attempting to access a protected web resource, a web application client can authenticate a user to the web application container using one of the following mechanisms:

- HTTP Basic Authentication (standard support)
- HTTP Digest Authentication (standard support)
- Form-based Authentication (standard support)
- Java Card Platform Authentication (introduced as an extensible set of vendor or container-specific mechanisms)

The authentication mechanism to be used when accessing a protected web resource **MUST** be specified in the application's deployment descriptor (web application deployment descriptor). The platform **MUST** reject a web application which declares in its deployment descriptor an authentication method that is not supported.

These authentication mechanisms can be used for both global authentication of card-holder-users and session-scoped authentication of card-holder-users and other-users.

6.4.5.1 Container Checking of User Authorization

When a user attempts to access a protected web resource, the container **MUST** determine whether the user is authorized to access the resource by checking if any of the following conditions is true:

1. There is an authenticated identity associated with the current authentication session (a session authenticator URI) and that authenticated identity corresponds to one of the roles permitted to access the resource.
2. One of the roles permitted to access the resource corresponds to a globally authenticated identity (a global authenticator URI).

If the user is authorized, the requested web resource is activated and a reference to it is returned.

If the user is authenticated but not authorized, the request **MUST** be rejected as forbidden and a 403 status code **MUST** be returned.

If the user is not authenticated and if there is no login configuration or authentication method (`login-config` and `login-config/auth-method` elements) defined for the web application, the request **MUST** be rejected as forbidden and a 403 status code **MUST** be returned.

If the user is not authenticated and if there is a login configuration and authentication method (`login-config` and `login-config/auth-method` elements) defined for the web application, the container **MUST** attempt to authenticate the user as follows:

1. The container **MUST** query the user for his credentials (including his user name) through the requesting client according to the authentication scheme identified in the `login-config/auth-method` element of the web application deployment descriptor.
2. For the first¹¹ role permitted to access the resource (which is done in the order by which they are listed in the deployment descriptor) mapped to an authenticator whose URI's `<scheme>` and `<realm>` path components match the values of the `login-config/auth-method` and `login-config/realm-name` elements¹², respectively, of the web application deployment descriptor, and whose URI's `<user>` path component matches the user name provided by the user, the container **MUST** proceed as follows:
 - a. The container **MUST** locate the authenticator. If the container fails to locate the authenticator, the authentication **MUST** fail.
 - b. The container **MUST** attempt to authenticate the user with the provided credentials.
3. If none of the roles permitted to access the resource is mapped to an authenticator URIs matching the provided user name, the authentication **MUST** fail.
4. If authentication fails, therefore the user is not authorized, and the status code of the response is set to 401 (Unauthorized).
5. If authentication succeeds, the user is authorized and the requested resource is returned to the client.

This process is a refinement of the generic process described in the *Java Servlet Specification for the Java Card Platform* where the container first attempts to authenticate the user and then determines if the user's authenticated identity is permitted to access the resource. For security reasons, the validation of the credentials provided by the user during authentication must only happen if the user's claimed identity is known to be permitted to access the resource.

11. It is recommended that the user management policy enforces the uniqueness of user names across card-holder-users and other-users authenticator namespaces

12. When the `login-config/realm-name` element is not present an empty realm name is assumed.

Some authentication schemes, such as the standard form-based authentication mechanism and the Java Card Platform authentication mechanism defined below, may use means other than setting the status code of the response to 401 (Unauthorized) for triggering authentication and/or handling authentication failure.

The `getRemoteUser` method of the `HttpServletRequest` interface returns the user name the client used for authentication. This value also corresponds to the `<user>` path component of the authenticator URI used for authentication. Note that if access to a protected resource is granted on the basis of some (other) globally authenticated card holder identity rather than on the actual authenticated identity of the user, the `getRemoteUser` method **MUST** still return the user name provided by the remote user if he already authenticated, or `null` if the remote user has not already been authenticated.

6.4.5.2 HTTP Basic, HTTP Digest and Form-based Authentication Schemes

HTTP Basic, HTTP Digest and Form-based Authentications are all based on a username and password.

These three schemes **MUST** be mapped to either PIN-based or password-based authenticators. Typically, for card-holder-user authentication, these three schemes will be mapped to PIN-based authenticators, while for other-users they will be mapped to password-based authenticators. The platform **MUST** reject web applications for which these three schemes are not mapped to PIN-based or password-based authenticators.

When performing PIN-based authentication using one of these three schemes, the password value **MUST** be interpreted as the PIN, and the user name either explicitly queried or implicitly set¹³ **MUST** be used to determine the URI of the authenticator.

Caution – On the Java Card Platform, when an application is configured for container-managed HTTP Basic or Digest authentication, the web container **MUST** filter out `Authorization` request headers so that they are not accessible to the application. When an application is not configured for container-managed HTTP Basic or Digest authentication, `Authorization` request headers are not filtered out. This allows for application-managed HTTP BASIC and DIGEST authentication.

13. When using form-based authentication for PIN-based authentication, a dynamically generated login form may not display a user name field but may include a hidden field set with the user name to be used for determining the authenticator's URI.

When HTTP DIGEST authentication is being used, the web container MUST check the credentials of the user by calling the `check` method of the `javacardx.servlet.http.HttpDigestAuthentication` interface on the authenticator. If the authenticator does not implement the `HttpDigestAuthentication` interface the authentication MUST fail.

To facilitate the implementation by application and library developers of authenticators that support HTTP DIGEST authentication, instances of `javacardx.biometry.OwnerBioTemplate` created for the type `BioBuilder.PASSWORD` SHOULD implement the `HttpDigestAuthentication` interface.

6.4.5.3 Java Card Platform Authentication Schemes

In addition to the standard authentication schemes, the web application container MUST support the use of the Java Card Platform authentication mechanism for container-managed authentication. This mechanism is intended to be extensible and to support various authentication schemes such as biometric which have requirements beyond those of simple user name and password.

The Java Card Platform authentication mechanism MUST be identified in the `auth-method` element of web application deployment descriptors with names in the form `JC-<scheme>`, where the `<scheme>` name component must match the scheme path component of registered authenticator's URIs. For example, an `auth-method` element value of `JC-fingerprint` constrains the container to use fingerprint-based authenticators that must be registered with URIs ending in `/fingerprint`, for example, `sio:///standard/auth/holder/global/transit/pos/owner/fingerprint`.

The Java Card Platform authentication mechanism behaves similarly to the standard form-based authentication mechanism and reuses its deployment descriptor entries for login and error pages: `form-login-page` and `form-error-page`.

The login form MUST collect the user's name and scheme-specific credential and MUST pass them as parameters of an HTTP request to the `jc_security_check` URL. This URL is relative to the base URL of the login form. The parameters for the user's name and scheme-specific credential MUST be named `jc_username` and `jc_<scheme>` (for example, `jc_pin`, `jc_password` or `jc_fingerprint`), respectively. For a biometric authentication scheme, the `jc_<scheme>` parameter will contain the data of the candidate template. When the scheme-specific credentials are too large, are binary data or are text containing non-ASCII characters, the content type `"multipart/form-data"` may be used for the request. If implemented in HTML, the login form may contain fields for entering a user name and a scheme-specific credential. These fields must be named `jc_username` and `jc_<scheme>`, respectively. The action of the login form must be `jc_security_check`.

Note that a Java Card Platform authentication mechanism can also be used for PIN-based and password-based authentication, in which case the `auth-method` element of the web application deployment descriptor must be set to `JC-pin` and `JC-password`, respectively.

When a user attempts to access a protected web resource, the container checks the user's authentication. If the user is authenticated and possesses authority to access the resource, the requested web resource is activated and a reference to it is returned. If the user is not authenticated, the web application container **MUST** perform the following steps:

1. The login form associated with the security constraint is sent to the client and the URL path (and parameters) triggering the authentication is stored by the container.
2. The user is asked to fill out the form.
3. The client posts the form back to the server.
4. The container determines if there is an authorized role for accessing the resource that matches the user's claimed identity and then attempts to authenticate the user using the information from the form and the configured scheme.
5. If there is no role matching the user's claimed identity, or if authentication fails, the error page is returned using either a forward or a redirect, and the status code of the response is set to 200.
6. If the user is authorized, the client is redirected to the resource using the stored URL path and parameters.

The error page sent to a user that is not authenticated **MUST** contain information about the failure.

If the user is authenticated, the method `getAuthType` of the `HttpServletRequest` interface may be used to retrieve the name of the scheme used for authentication. The returned value **MUST** be the exact authentication scheme name as declared in the `auth-method` element of the web application deployment descriptor, for example, `"JC-fingerprint"`.

An example of configuration for PIN-based authentication of a web application is given in [CODE EXAMPLE 6-1](#) and [CODE EXAMPLE 6-2](#).

CODE EXAMPLE 6-1 HTML Form For PIN-based Authentication

```
<form method="POST" action="jc_security_check">
  <input type="text" name="jc_username" />
  <input type="password" name="jc_pin" />
</form>
```

CODE EXAMPLE 6-2 Deployment Descriptor Configuration For PIN-based Authentication

```
<login-config>
  <auth-method>JC-pin</auth-method>
  <form-login-config>
    <form-login-page>pin-based-login.html</form-login-page>
    <form-error-page>error.html</form-error-page>
  </form-login-config>
</login-config>
```

An example of configuration for fingerprint authentication of a web application is given in [CODE EXAMPLE 6-3](#) and [CODE EXAMPLE 6-4](#).

CODE EXAMPLE 6-3 HTML Form For Fingerprint Authentication

```
<form method="POST" action="jc_security_check"
  enctype="multipart/form-data">
  <input type="text" name="jc_username" />
  <input type="file" name="jc_fingerprint" />
</form>
```

CODE EXAMPLE 6-4 Deployment Descriptor Configuration For Fingerprint Authentication

```
<login-config>
  <auth-method>JC-fingerprint</auth-method>
  <form-login-config>
    <form-login-page>fingerprt-based-login.html</form-login-page>
    <form-error-page>error.html</form-error-page>
  </form-login-config>
</login-config>
```

6.4.6 Card Holder Authorization For Remotely Accessible Applications

Clients of web applications can be categorized into two types:

- card holder-facing clients, which may directly and safely interact with the card holder. These clients are typically local, co-hosted on the card-hosting device, or in close proximity to the card. The determination of whether a client is facing the card holder or can otherwise be trusted as directly and safely interacting with the card holder is not in the scope of this specification. But such a determination can, for example, be made based on credentials (such as the client's certificate) or on the client's IP address.

- non-card holder-facing clients, which do not directly interact with the card holder, but interact with some other-users such as remote administrators. These clients are typically remote systems that may interact with the card through the network to which the card-hosting device itself is connected.

Access to locally accessible web applications (meaning applications that may be expected to interact with the card holder) MUST only be allowed from card holder-facing web clients. When using container-managed authentication, the web container MUST NOT proceed with a card-holder-user authentication through a web client that is not facing the card holder or considered otherwise trusted.

Access to remotely accessible applications (meaning, applications that are not expected to interact with the card holder but with other-users) requires card holder authorization when accessed from non-card holder-facing clients. Web applications requiring card holder authorization MUST include in their Java Card Platform-specific application descriptor a `card-holder-authorization` element and one or more `role-name` sub-elements designating role names mapped to card-holder-users, see [card-holder-authorization Element](#) in [Chapter 8](#). Access to such a web application MUST only be granted if the card holder is already authenticated in one of the roles listed. This applies, in addition, to the authorization constraints on remote users that the web application's deployment descriptor may define for specific portions of the web application (web resources). If no role is listed, a web client MUST NOT be allowed access to any of the resources of the web application.

[FIGURE 6-3](#) depicts the various interactions when accessing locally or remotely accessible web applications from a card holder-facing client. The numbering indicates typical interaction sequences.

FIGURE 6-3 Access from a Card Holder-Facing Client (Collaboration Diagram)

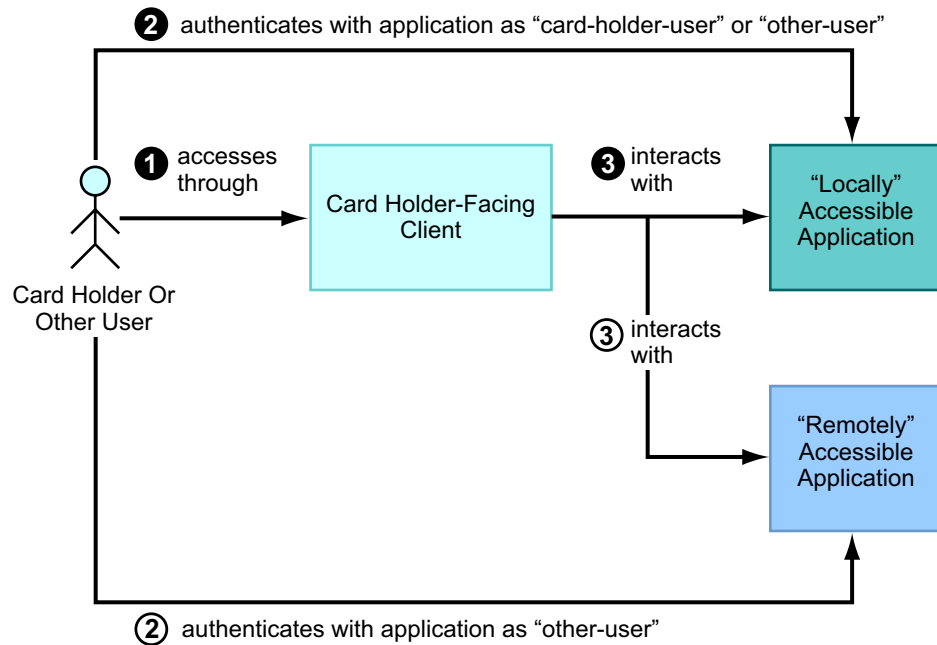


FIGURE 6-4 depicts the various interactions when accessing a remotely accessible web application from a non card holder-facing client. The numbering indicates typical interaction sequences.

FIGURE 6-4 Access from a Non Card Holder-Facing Client (Collaboration Diagram)

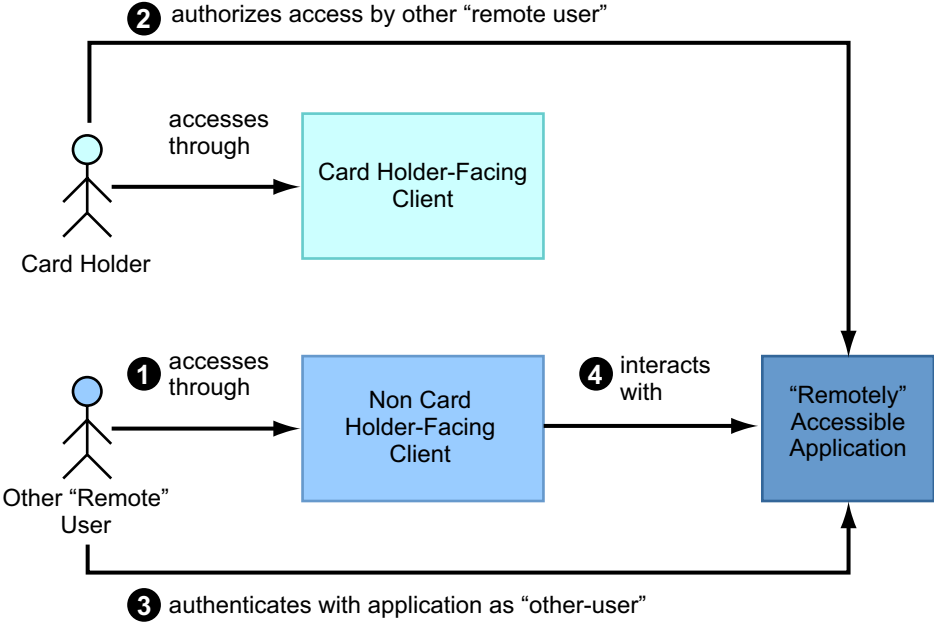


TABLE 6-5 summarizes when card holder authorization is required for granting access to a web application depending on the types of clients requesting the access and the types of applications being accessed.

TABLE 6-5 Card Holder Authorization Requirements for Access by User

Access From:	Access To:	
	Locally Accessible Application	Remotely Accessible Application
Card Holder Facing Client	Not Required	Not Required
Non Card Holder Facing Client	REJECTED	Required

The verification of whether the card holder is already authenticated in one of the roles listed for authorizing a remote access to an application **MUST** be done by locating the card-holder-user authenticators these roles are mapped to and invoking their `Authenticator.isValidated` method.

If none of the card-holder-users the roles listed are mapped to is already authenticated, the web container **MUST NOT** attempt to authenticate any of these card-holder-users through the requesting web client, and it **MUST** reject the request as forbidden. A 403 status code **MUST** be returned to the requesting web client.

For example, to allow access from remote web clients if the card holder is authenticated in the ADMIN role, the application's Java Card Platform-specific descriptor would contain:

```
<card-holder-authorization>
    <role-name>ADMIN</role-name>
</card-holder-authorization>
```

The platform MUST allow for these role names (card holder authorization role names) to be mapped to either global or session-scoped card-holder-user authenticator URIs. The card holder authorization, if granted for an application, is granted for the duration of the HTTP session between the remote client and that application. Card holder authorization is granted on per HTTP session and per application basis. When during the same interaction session between the remote client and the card, the remote client access another web application that requires card holder authorization, the authorization MUST be checked for that new application.

The platform MUST reject web applications that require card holder authorization and that map a card holder authorization role name to an authenticator URI that is not in the card-holder-user authenticator namespace. Mapping of roles other than card-holder-authorization roles to card holder authenticators is allowed in remotely accessible application but active authentication in these roles using these card holder identities MUST only be allowed when accessed from trusted-card-holder-facing terminal.

TABLE 6-6 summarizes the role-to-user mapping that must be supported (or rejected by the platform) for both locally accessible applications and remotely accessible applications depending on the role (card holder authorization role or other role), the type of user identity to which that role is mapped.

TABLE 6-6 Supported Role-to-User Mapping For Locally Accessible and Remotely Accessible Applications

Role	User Identity	Locally Accessible Application	Remotely Accessible Application
Card Holder Authorization Role	Card Holder User	N/A	Supported
	Other User		Rejected
Other Role	Card Holder User	Supported	Supported
	Other User	Supported	Supported

Caution – Relying on global card-holder-user authentication for card holder authorization to access remotely accessible web applications may create vulnerabilities because this authorization is implicitly granted to all web clients the web application trusts.

Note – Application developers and/or providers may use this mechanism to delegate card-holder-user authentication and authorization to a locally accessible application that registered itself as an authentication service (with a custom authenticator).

6.4.6.1 Container Checking of Card Holder Authorization

When a user attempts to access a web application's resource, the container **MUST** proceed as follows:

1. If the web client is a trusted card holder facing client, the container **MUST** proceed with the regular checking of user authorization as per the process described in [Section 6.4.5.1, "Container Checking of User Authorization" on page 6-35](#).
2. If the web client is not a trusted card holder facing client, the container **MUST** proceed as follows:
 - a. If the session was already authorized by the card holder, the container **MUST** proceed with the regular checking of user authorization as per the process described in [Section 6.4.5.1, "Container Checking of User Authorization" on page 6-35](#).
 - b. If the session was not already authorized by the card holder and if the application's Java Card Platform-specific descriptor contains a `card-holder-authorization` element with one or more `role-name` sub-elements designating role names, then for each of the card-holder-users these roles are mapped to, the container **MUST** proceed as follows:
 - i. The container **MUST** locate the card-holder-user authenticator. If the container fails to locate the authenticator, the card-holder-user **MUST** be skipped.
 - ii. If the card-holder-user is already authenticated as per a call to the authenticator's `Authenticator.isValidated` method, the container **MUST** grant the access for the duration of the HTTP session with this web application and the container **MUST** proceed with the regular checking of user authorization as per the process described in [Section 6.4.5.1, "Container Checking of User Authorization" on page 6-35](#).
 - iii. If none of the card-holder-users the roles are mapped to are already authenticated, then the request **MUST** be rejected as forbidden and a 403 status code **MUST** be returned.

- c. If the application's Java Card Platform-specific descriptor does not contain a `card-holder-authorization` element or if it has no `role-name` sub-elements, then the request MUST be rejected as forbidden and a 403 status code MUST be returned.

6.5 On-card Client Application Authentication and Authorization

Client application authentication is the process by which a client application proves its identity to a server application. This authenticated identity is then used by the server application to perform authorization decisions for accessing protected resources, such as SIO-based and event-based services it exposes.

A server application may initiate and check authentication of a client application programmatically by invoking the client role-based security API described in [Section 6.3.2.1, “Programmatic Client Role-based Security”](#) on page 6-27, and passing a client security role name that has been mapped to credential aliases, designating the credentials of authorized application clients.

Client application authentication is tracked on a per-server application basis. That is a client application may authenticate with several different server applications and the authentication of the client application with each of these server applications must be tracked independently.

The Java Card RE implementation MUST support tracking the state of client application authentication on a per-server application basis.

6.5.1 On-card Client Application Authentication

When a server application invokes the `JCSystem.isClientInRole` method with a role name argument that has been mapped to one or several credential aliases, and with the URI of the protected resource the client application is requesting access to, the Java Card RE MUST proceed as follows:

1. The `CredentialManager` instance applicable to the server application for the mode `MODE_SIO_SERVER` is retrieved, see [Section 6.6.2, “Retrieving Security Requirements and Credential Managers For Establishing Connections”](#) on page 6-51.
 - If none can be retrieved, the authentication fails and false is returned.

2. The `CredentialManager` instance applicable to the client application for the mode `MODE_SIO_CLIENT` is retrieved, see [Section 6.6.2, “Retrieving Security Requirements and Credential Managers For Establishing Connections”](#) on page 6-51.
 - If none can be retrieved, the authentication fails and `false` is returned.
3. The server’s `CredentialManager` instance `getTrustedCredentials` method is invoked with the list of credential aliases to which the client security role is mapped and with the URI of the protected resource the client application is requesting access to.
 - The `getTrustedCredentials` method must return the list of actual credentials (secret keys, public keys or public key certificates) associated with the aliases.
 - If no credential is returned, the authentication fails and `false` is returned.
4. The client’s `CredentialManager` instance `getCredentials` method is invoked with the list of credential types requested by the server and with the URI of the protected resource the client application is requesting access to.
 - The `getCredentials` method must return a list of one or more credentials (secret keys or private keys) of the requested types and corresponding to the protected resource URI¹⁴.
 - If no credential is returned, the authentication fails and `false` is returned.
5. The server’s trusted client credentials and the client’s credentials are checked two-by-two, according to their respective pairing property¹⁵, using a challenge-response authentication algorithm until a matching pair is found.
 - If one of the client’s credentials matches - as per the cryptographic challenge-response authentication algorithm - one of the server’s trusted client credentials, the authentication succeeded and `true` is returned.
 - Otherwise, `false` is returned.

The Java Card RE MUST use a cryptographic challenge-response authentication algorithm with randomly generated challenges to authenticate the client application with the server application using the retrieved server application’s trusted client credentials and client application’s credentials.

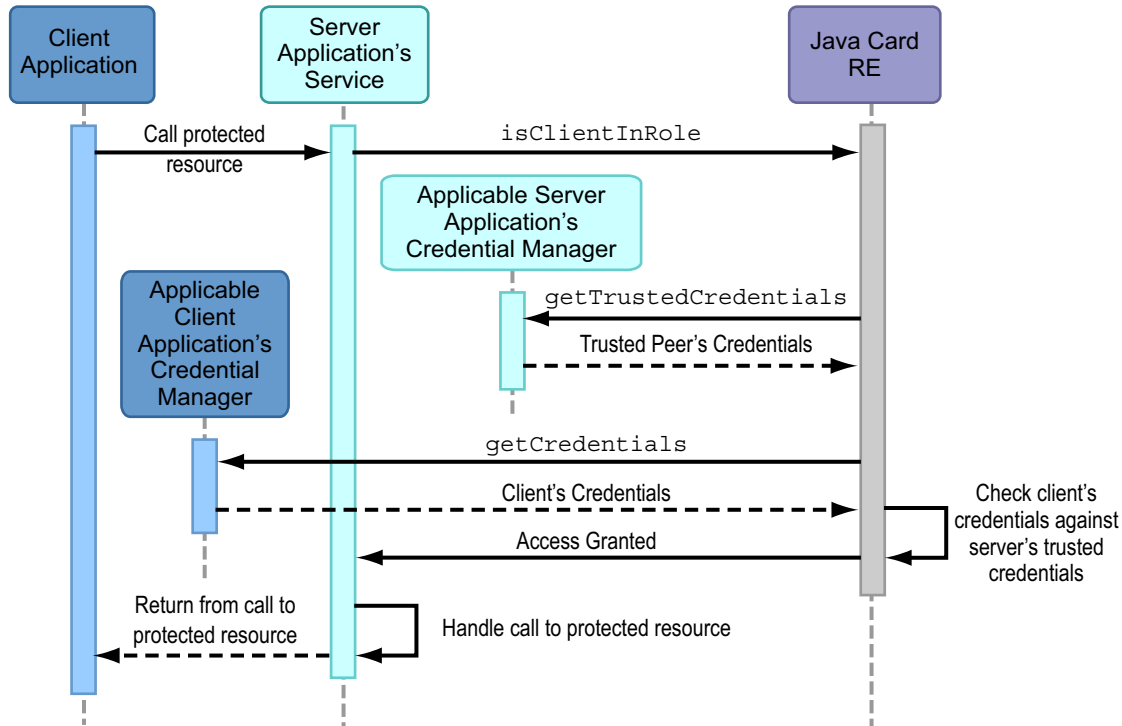
This process MUST execute within the Java Card RE context and the ownership of the credential objects MUST be retained by their original owner at all times.

14. An application client may return several matching credentials when it may be granted several security roles by the same server application.

15. Secret keys against secret keys, private keys against public keys or subject’s public keys of certificates

FIGURE 6-5 depicts the interaction between the different components and actors of on-card client application authentication. Objects colors indicate common ownership of interacting objects. The colors of focus of control boxes¹⁶ indicate the currently active contexts as the results of specific calls.

FIGURE 6-5 On-card Client Application Authentication (Sequence Diagram)



6.5.2 Authentication Session Duration

The authentication status of a client application with a server application is valid for the duration of the authentication session. The duration of authentication session may be of one of the following types:

- **Access-bound:** authentication in a particular role **MUST** be performed for each invocation of the `JCSystem.isClientInRole` method.

¹⁶In UML sequence diagrams, a focus of control is a rectangle that lies on an object lifeline and which depicts the duration of a particular call to that object.

- **Card session-bound:** authentication in a particular role MUST be performed once for a specific client application and specific server application during the card session. The result of the first successful authentication of the client application in a particular role MUST be cached until the platform is reset.
- **Application lifetime-bound:** authentication in a particular role MUST be performed once for a specific client application and specific server application during their common lifetime. The result of the first successful authentication of the client application in a particular role MUST be cached until either of the server application or client application instance is deleted (uninstalled).

The type of the authentication session duration MUST be set in the runtime descriptor of the application, either globally for all of its client security roles or individually for each of its client security roles, see [On-Card-Clients-Credential-Auth-Duration Attribute](#) in [Chapter 8](#). The choice of the type of the authentication session duration may depend on the sensitivity of the security role, as well as on whether a client application's or server application's credentials are managed dynamically (may be changed during their lifetime) or not.

6.6 Security Requirements and Credential Management of Secure Communications

On the Java Card Platform, applications may interact with on-card or off-card peers through the following secure communications:

- Secure network connections over Secure Sockets Layer (SSL) or Transport Layer Security (TLS) such as:
 - Web container-managed HTTPS server connections between an off-card client and a web application.
 - Application-managed secure stream socket server GCF connections between an application and an off-card client.
 - Application-managed HTTPS or secure stream socket client GCF connections between an application and an off-card server.
- SIO-based inter-application communications between a server application and a client application.

The security characteristics of a secure communication depend on the type of connection:

- confidentiality, integrity and endpoint authentication for SSL or TLS-based network connections,
- endpoint authentication for SIO-based inter-application communications.

The security requirements - meaning 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, **MUST** be retrieved from settings that **MUST** be either specific to that application or common to all applications in the same group context. Similarly, the management of the credentials and trust decisions required to establish the secure communication **MUST** be delegated to a credential manager that **MUST** be either specific to that application or common to all applications in the same group context.

Application developers may configure the security requirements for secure communication with peers. The security requirements for a secure communication may cover the followings:

- peer authentication,
- integrity of the data transmitted,
- confidentiality of the data transmitted.

Security requirements are encapsulated by an instance of the `javacardx.security.CredentialManager.SecurityRequirements` class.

Application developers may use a credential manager to manage the credentials used to establish secure communication with peers. A credential manager is used both for managing the key material which is used to authenticate with peers and for managing the trust material that is used when making trust decisions such as deciding whether credentials presented by a peer should be accepted. A credential manager addresses the following requirements:

- Retrieving and handling an application's private credentials securely within the application's group context.
- Retrieving credentials in any required application-specific way.
- Retrieving and handling credentials that may be dependent on the type of communication, meaning the role, or mode, the application plays in the communication such as client or server, and on the security characteristics of the communication.

A credential manager is encapsulated by an instance of the `javacardx.security.CredentialManager` class.

6.6.1 Assignment of Security Requirements and Credential Managers

Credential managers and security requirements may be assigned as follows:

- A security requirements or credential manager instance may be assigned to an application's group context by the card manager application when that application's group context is created. The security requirements or credential

manager instance set by the card manager application MUST apply to all modes by default. The security requirements or credential manager instance set by the card manager application MUST NOT be accessible to applications. The security requirements or credential manager instance set in a group context MUST apply to all applications in that group context.

- Security requirements or credential manager instances may be assigned to an application instance by that application instance itself. An application may set security requirements or credential manager instances using the `CredentialManager`'s methods `setSecurityRequirements` and `setCredentialManager`, respectively. An application may retrieve the application-assigned security requirements or credential manager instances using the `CredentialManager`'s methods `getSecurityRequirements` and `getCredentialManager`, respectively. The security requirements or credential manager instances set by an application instance MUST apply to that application instance only.

Setting and getting of credential managers by applications MUST be subject to access control. When the `CredentialManager.setCredentialManager` and `CredentialManager.getCredentialManager` methods are invoked, a security check MUST ensure that the permission `javacardx.framework.JCRuntimePermission` with the target name `credentialManager.set` and `credentialManager.get`, respectively, is granted.

Note that setting and getting of security requirements by applications is not subject to access control.

Retrieving the card manager-assigned credential manager and security requirements of an application from its group context MUST be protected by package access control builtin checks as described in [Section 6.8.1, “Built-in Checks” on page 6-63](#) and MUST be restricted to card management applications. Retrieving the card manager-assigned credential manager and security requirements of an application from its group context MUST additionally be subject to access control: a security check MUST ensure that the permission `javacardx.spi.cardmgmt.CardManagementPermission` with the target name `credentialManager.get` is granted.

6.6.2 Retrieving Security Requirements and Credential Managers For Establishing Connections

The security requirements and the credential manager retrieved for establishing a secure communication MUST be the ones set for the corresponding type and mode of communication:

- `CredentialManager.MODE_WEB_SERVER`, for web container-managed secure server connections.
- `CredentialManager.MODE_GCF_CLIENT` and `MODE_GCF_SERVER`, for application-managed GCF secure client connections and application-managed GCF secure server connections, respectively.
- `CredentialManager.MODE_SIO_CLIENT` and `MODE_SIO_SERVER`, for each ends of SIO method invocations.
- `CredentialManager.MODE_DEFAULT`, when no security requirements or credential manager instance can be retrieved for the specific mode corresponding to the communication being established.

When a security requirements or credential manager instance has been set for a mode of communication by an application, it **MUST** be retrieved in priority to the instance set in the group context by the card manager application when establishing a secure communication in that mode for that application. That is the security requirements or credential manager set by the application overrides the security requirements or credential manager set by the card manager application:

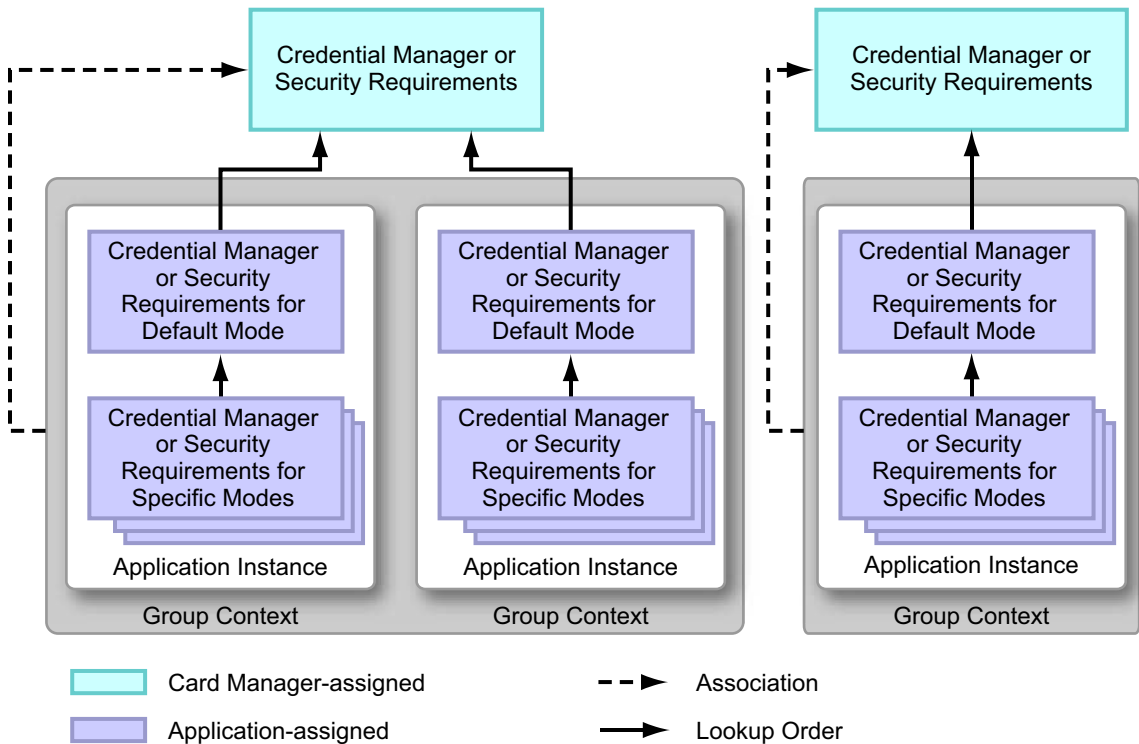
- If a security requirements or credential manager instance has been set for a particular mode or for the default mode by an application, that instance **MUST** be used to establish the secure communication. A security requirements or credential manager instance set for the default mode by an application overrides the instance set by the card manager in all modes.
- If no security requirements or credential manager instance has been set for that particular mode or for the default mode by an application, the security requirements or credential manager instance assigned to the group context by the card manager application **MUST** be used to establish the secure communication.

If no applicable credential manager instance can be retrieved, the secure communication **MUST** fail.

If no applicable security requirements can be retrieved, the security requirements **MUST** default to false, meaning there is no specific requirement, unless the application is a web application and the mode of communication is `CredentialManager.MODE_WEB_SERVER`, in which case the security requirements **MUST** correspond to the requirement for client authentication and the overall requirements for content integrity and confidentiality of the web application as declared in its runtime descriptor and deployment descriptor, respectively. See [Section 3.5.3, “Retrieving a Web Application Instance’s Security Requirements and Credentials”](#) on page 3-26.

FIGURE 6-6 depicts the assignment of security requirements or credential manager instances by the card manager and application instances, and the order in which they are retrieved.

FIGURE 6-6 Security Requirements and Credential Manager Assignment and Lookup Order



6.6.3 Invocation of Security Requirements and Credential Managers

The `SecurityRequirements` and `CredentialManager` class defines the following methods

- Methods that **MUST** be invoked by the SSL/TLS protocol handler during the SSL/TLS handshake procedure that allows the SSL/TLS client and server to negotiate the characteristics of the connection's security such as confidentiality, integrity and client authentication requirements.
- Methods that **MUST** be invoked by the Java Card RE during the on-card client application authentication procedure that allows a client application to prove its identity to the server application from which it wants to get services, such as through an SIO-based service or event, see [Section 6.5.1, "On-card Client Application Authentication"](#) on page 6-46.

See the API specification for a detailed description of these methods.

A Java Card Platform implementation MUST NOT cache credentials or trust decisions returned or performed by credential managers.

For secure network communications, the security requirements and credential manager instance MUST be retrieved for each SSL/TLS handshake. The same security requirements and credential manager instance MUST be used throughout a SSL/TLS communication session, such as when required for SSL/TLS re-handshake.

In the case of inter-application SIO-based communications, the credential manager instance MUST be retrieved for each on-card client authentication according to the authentication session duration, see [Section 6.5.2, “Authentication Session Duration” on page 6-48](#).

The callers of the `SecurityRequirements` and `CredentialManager` methods, the SSL/TLS protocol handler or the Java Card RE itself, MUST invoke these methods with at least the following mandatory parameters:

- The mode of operation corresponding to the type and mode of the connection or access.
- The connection or access endpoint URI in its canonical form as defined in [Section 2.3.3.1, “Canonicalization of URIs” on page 2-8](#).

[TABLE 6-7](#) describes the endpoint URI formats that MUST be used in conjunction with the different modes of operation.

TABLE 6-7 Connection or Access Endpoint URIs and Modes of Operations

Mode of Operation	Endpoint URI
<code>CredentialManager.MODE_WEB_SERVER</code>	Connection endpoint URL in the following form <code>https://:<port number><context path></code> ; where <i><port number></i> is the secure port number allocated to the web application instance and <i><context path></i> corresponds to the context path assigned to the web application instance; or in other terms its application URI.

TABLE 6-7 Connection or Access Endpoint URIs and Modes of Operations (Continued)

Mode of Operation	Endpoint URI
CredentialManager.MODE_GCF_CLIENT	Connection endpoint URL as passed to a <code>javax.microedition.io.Connector.open</code> method. Examples are: <code>ssl://<host name>:<port number></code> <code>https://<host name>:<port number><path></code> where <code><host name></code> represents the peer's, meaning server's, host name and <code><port number></code> is the secure port number for the client connection.
CredentialManager.MODE_GCF_SERVER	Connection endpoint URL as passed to a <code>javax.microedition.io.Connector.open</code> method. Examples are: <code>ssl://:<port number></code> where the secure port number for the server connection.
CredentialManager.MODE_SIO_CLIENT CredentialManager.MODE_SIO_SERVER	SIO-based access endpoint URL or resource URL as passed to the <code>javacardx.framework.JCSystem.isClientInRole</code> method. Examples are: <code>sio://<application path>/<sio path></code> <code>sio://<reserved path>/<sio path></code> <code>event://<application path>/<event path></code> <code>event://<reserved path>/<event path></code> where <code><application path></code> represents the application's URI path, <code><reserved path></code> represents one of the reserved path /platform or /standard, and <code><sio path></code> and <code><event path></code> represent specific SIO-based resource paths.

FIGURE 6-7 illustrates the invocation sequence of `CredentialManager` methods during a TLS Handshake with asymmetric, or public key, cryptography authentication. See *RFC 5246 The Transport Layer Security (TLS) Protocol Version 1.2* for details on the TLS Handshake Protocol.

FIGURE 6-7 Invocation of CredentialManager Methods During a TLS Handshake with Asymmetric, or Public Key, Cryptography Authentication

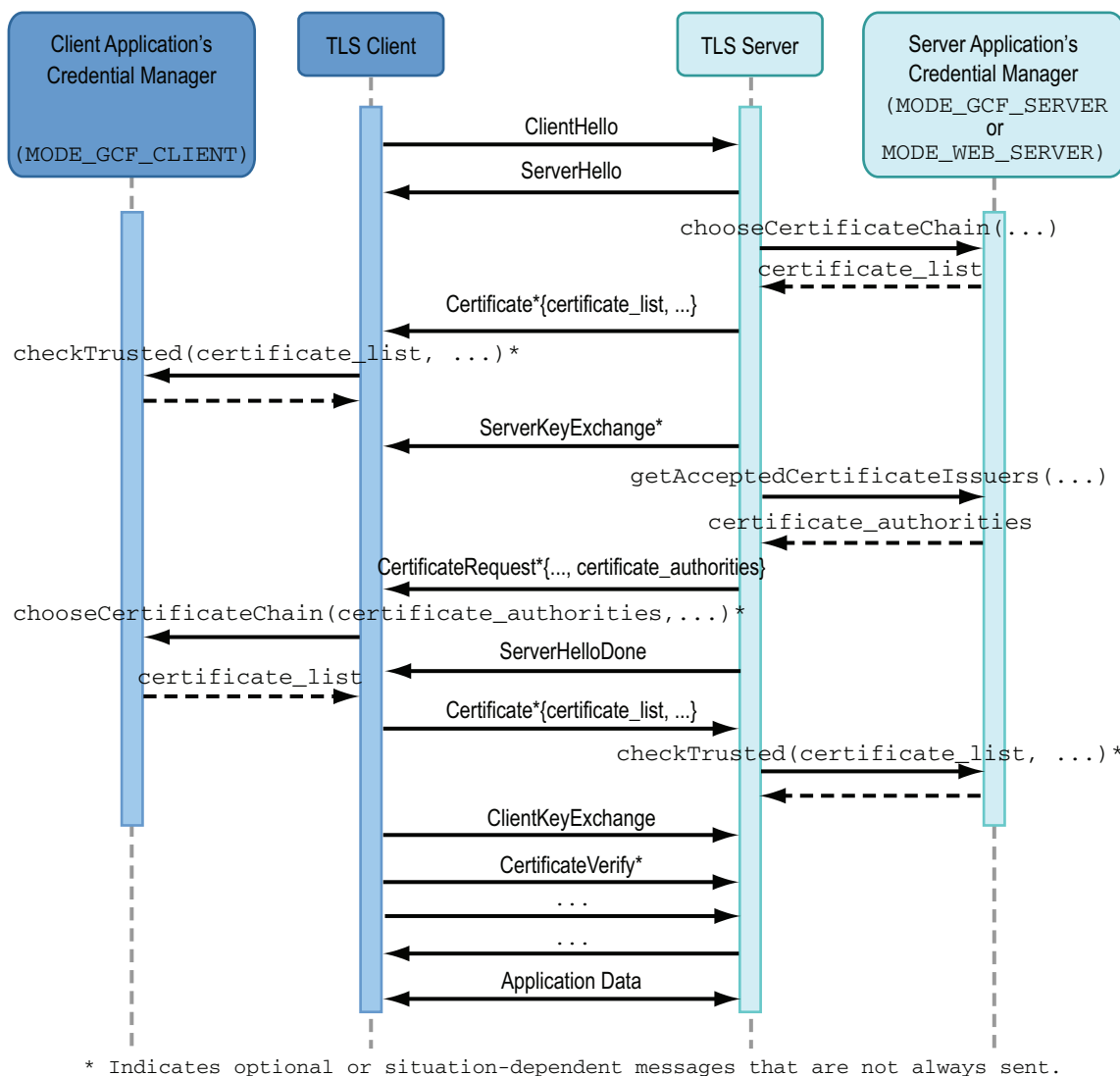
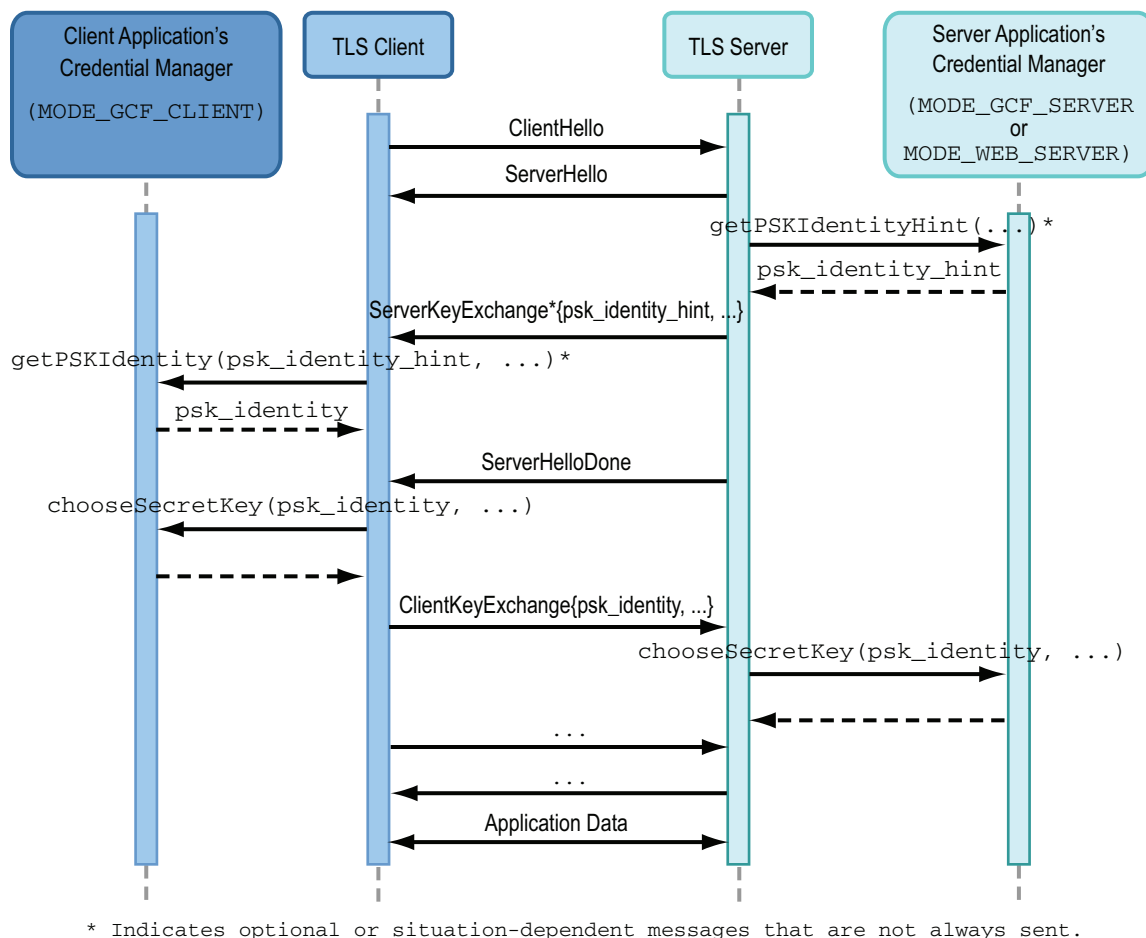


FIGURE 6-8 illustrates the invocation sequence of CredentialManager methods during a TLS Handshake with symmetric, or pre-shared secret key, cryptography authentication. See *RFC 4279 Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)* for details on the TLS Handshake Protocol.

FIGURE 6-8 Invocation of CredentialManager Methods During a TLS Handshake with Symmetric, or Pre-shared Secret Key, Cryptography Authentication



6.7 Code Isolation

Code isolation ensures that code loaded from one application does not interfere with code of other applications. Code loaded in this manner cannot override or directly access code of other applications. This is implemented by defining and enforcing different class namespaces for each loaded application code.

On the Java Card Platform, code isolation **MUST** be implemented with a *class loader delegation hierarchy*, which enforces isolation of application code by default and allows for explicitly sharing code such as libraries and public interfaces between cooperating applications.

6.7.1 Class Loader Delegation Hierarchy

In order to enforce separate code isolation spaces (or class namespaces) for applications while still allowing them to communicate, the class loader delegation hierarchy **MUST** ensure the following:

- Java Card RE system classes are visible to all libraries and applications.
- Shareable interfaces used to communicate between applications from different group contexts are visible to all applications.
- Extension libraries are only visible to all extended applet applications and web applications.
- Classic libraries are only visible to all classic applet applications. Because classic applet applications are not thread-aware and to specifically avoid concurrent access to static fields and methods of classic library classes (classes in Java programming language packages without any applet), a classic library **MUST** be packaged individually and loaded so that the visibility on classic library packages is restricted to classic applet applications only.
- Application groups are isolated from each other, that is: classes of an application group are not visible to other application groups. Moreover, applications inside an application group are isolated from each other, that is: classes of an application are not visible to other applications in the same group. However, classes from group libraries that are shared among applications in an application group are visible to all applications in the same application group. Only web application groups and extended applet application groups may include group libraries.

Every class loader with the exception of the bootstrap class loader **MUST** have one and only one parent class loader. The class loader delegation hierarchy **MUST** be laid out as follows:

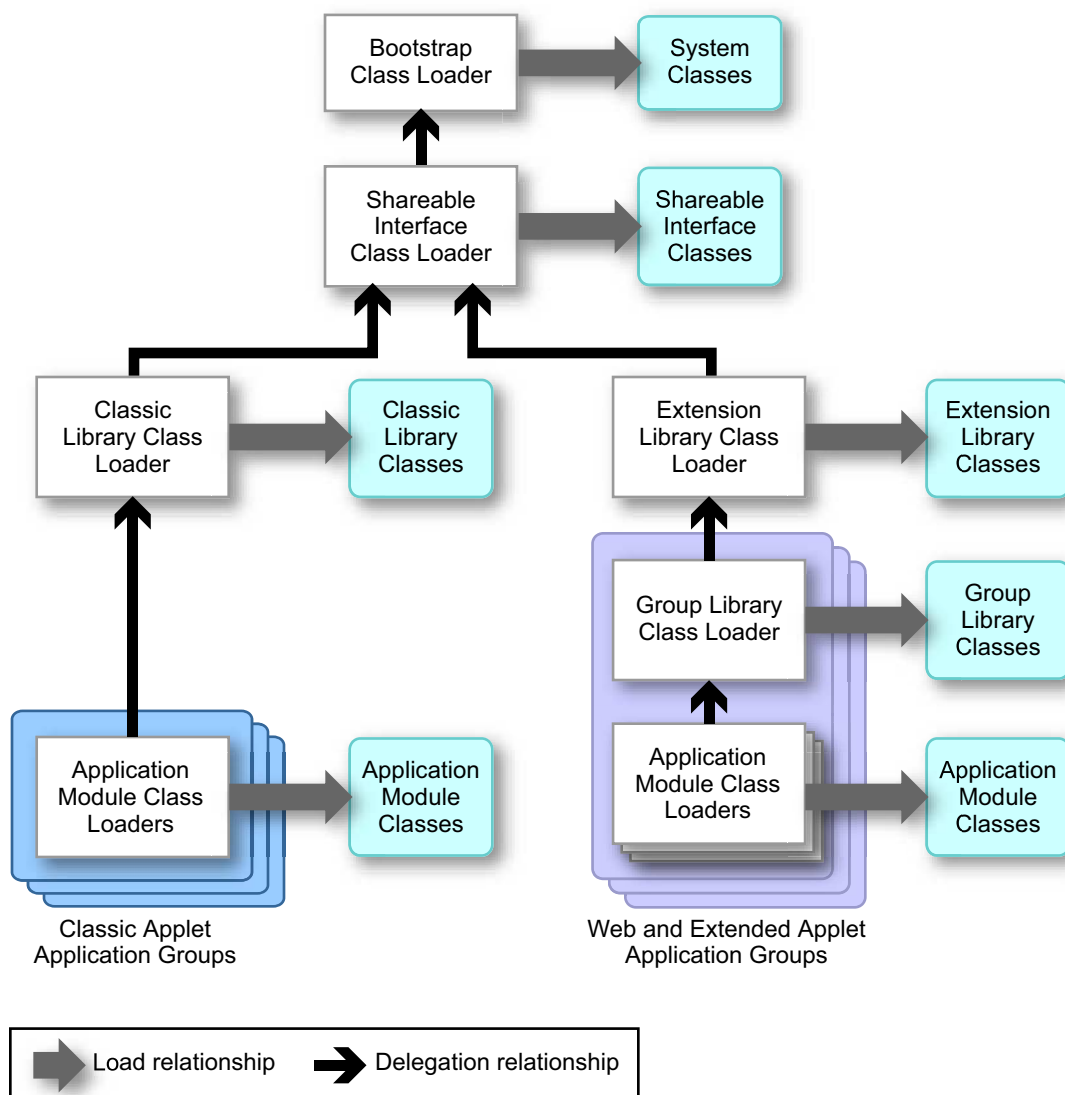
- The *bootstrap class loader* is the root of the delegation hierarchy. The bootstrap class loader **MUST** load the Java Card RE system classes.
- The *shareable interface class loader* is the direct child of the bootstrap class loader. The shareable interface class loader **MUST** load publicly exposed shareable interfaces.
- The *extension library class loader* is a direct child of the shareable interface class loader. The extension library class loader **MUST** load extension libraries.
- The *classic library class loader* is a direct child of the shareable interface class loader. The classic library class loader **MUST** load classic libraries.

- *Group-library class loaders* are direct children of the extension library class loader. Libraries, private to different application groups, MUST be loaded by distinct group library class loaders, one per web or extended applet application group.
- *Application module class loaders* are either direct children of the group library class loaders or direct children of the classic library class loader, depending on the type of application model. Application modules MUST be loaded by distinct application module class loaders, one per application module. Application module class loaders MUST be leaves of the class loader delegation hierarchy.

The class loader delegation hierarchy MAY be enhanced with *application framework class loaders* in order to load standard bodies-defined framework libraries:

- **Framework libraries shared among all classic applet applications and extended applet and web applications** - Such an application framework class loader MUST be inserted in the class loader hierarchy between the bootstrap class loader and the shareable interface class loader. These framework libraries must account for the limitations of classic applet applications. In particular, they must be thread-safe.
- **Framework libraries shared among a restricted set of application groups** - Such application framework class loaders MUST be direct children of the extension library class loader. In such a configuration, some group library class loaders MAY be direct children of application framework class loaders.

FIGURE 6-9 Class Loader Delegation Hierarchy



6.7.2 Class Loading Delegation Principle

The class loading on the Java Card Platform **MUST** address the following requirements:

- The same class cannot be loaded more than once by the same class loader or be loaded again by any of its children class loaders, thereby preserving type safety.
- All classes loaded by a class loader and its ancestors are visible to all its children class loaders but not vice versa, thereby allowing for restricted code sharing.
- A class is uniquely identified in the class loader delegation hierarchy (and therefore on the platform) by its name and the class loader that effectively loaded it, thereby allowing for code isolation.

To address these requirements, the class loading delegation **MUST** be implemented as follows:

1. When a loaded class references a class type and that reference must be resolved, the class loader of the loaded class **MUST** be requested to load the new class.
2. When the `java.lang.Class.forName` method is called to dynamically load a class, the class loader to be requested to load the class **MUST** be the defining class loader of the calling class.
3. When the `java.util.ResourceBundle.getBundle` method is called to dynamically load a subclass of `ResourceBundle`, the class loader to be requested to load the class **MUST** be the defining class loader of the calling class.
4. If a class loader is requested to load a class and, if the class has not already been loaded by that class loader, the class loader **MUST** first delegate to its parent class loader (unless it is the bootstrap class loader).
5. If a class cannot be loaded by its parent class loader, a class loader **MUST** attempt to locate the class file and define the class itself (convert it from its binary representation into an instance of `java.lang.Class`).
6. The same class **MUST NOT** be loaded more than once by the same class loader.
7. If a class cannot be loaded by a class loader a `ClassNotFoundException` **MUST** be thrown.

As a consequence, a class may be loaded by a class loader (its *defining class loader*) which may be different from the class loader (its *initiating class loader*) initially invoked to load the class. The defining class loader is either the initiating class loader or a parent of the initiating class loader. Since a class is uniquely identified by its name and its defining class loader, two classes are deemed the same (equals) if they have the same name and have the same defining class loader. Each class loader thereby defines a separate class namespace.

6.7.2.1 Loading of Array Classes

The `Class` instance internally created by the Virtual Machine to represent an array class **MUST** be assigned the defining class loader of its component type. If the array is an array of primitive types, the defining class loader of the array class **MUST** correspond to the bootstrap class loader.

6.7.3 User-defined Class Loaders

The class loader delegation hierarchy **MUST NOT** be exposed to application developers, and especially, there **MUST NOT** be support for user-defined class loaders.

6.7.4 Class-Path Resource Lookup

Class-path resources designate those resources that are packaged along with classes and that can be accessed using the `getResourceAsStream` method of the `java.lang.Class` class¹⁷.

Lookup of class-path resources **MUST** be performed by class loaders in the class loader delegation hierarchy according to the same delegation principle as that of class loading with the only restriction that Java Card RE system resources **MUST** not be accessible to applications or libraries. The class-path resource lookup delegation **MUST** be implemented as follows:

1. When the `getResourceAsStream` method is invoked on a `Class` instance, the class loader of that class **MUST** first delegate to its parent class loader (unless the parent is the bootstrap classloader¹⁸).
2. If the named resource cannot be located by its parent class loader, a class loader **MUST** attempt to locate the resource itself among the resources it loaded. The extension library class loader or a group library class loader **MUST** look up a resource in the libraries it loaded in the exact same order as these libraries were loaded (meaning, starting from the first loaded library).
3. If a resource cannot be found by a class loader, `null` **MUST** be returned.

As a result, the “visibility” of class-path resources is identical to that of classes, as described in the preamble of [Section 6.7.1, “Class Loader Delegation Hierarchy” on page 6-58](#), with the exception that Java Card RE system resources are not visible to libraries and applications.

¹⁷.These resources must not be confused with static web resources.

¹⁸.As per the above-mentioned restriction.

6.8 Package Access Control

The Java Card Platform provides different complementary mechanisms to control the definition and access to classes in packages.

To be allowed to define a new class in a package, all of the following MUST NOT fail:

- built-in checks for that package, see [Section 6.8.1, “Built-in Checks” on page 6-63](#)
- package sealing checks, see [Section 6.8.2, “Package Sealing Checks” on page 6-64](#)

6.8.1 Built-in Checks

The Java Card Platform MUST prevent loaded code, applications or libraries, from overriding or extending the set of the system classes provided in the following packages and sub-packages thereof:

- `java.*`
- `javax.*`
- `javacard.*`
- `javacardx.*`
- platform implementation-specific packages (including `com.sun.javacard.*` and `com.sun.javacardx.*`).

A Java Card Platform implementation MUST ensure that application or library developers cannot override, modify, or add any classes to these protected system packages. An attempt by a class loader other than the *bootstrap class loader* to define a class in one of these protected system packages MUST result in a `SecurityException` being thrown.

The Java Card Platform MUST prevent an extended applet or web application’s code from overriding or extending the set of classes in packages initially defined by extension libraries. An attempt by a class loader other than the *extension library class loader* to define a class in a package initially defined by an extension library MUST result in a `SecurityException` being thrown.

The Java Card Platform MUST prevent a classic applet application’s code from overriding or extending the set of classes in packages initially defined by classic libraries. An attempt by a class loader other than the *classic library class loader* to define a class in a package initially defined by a classic library MUST result in a `SecurityException` being thrown.

The Java Card Platform MUST prevent loaded code, applications or libraries, other than resident libraries and card management applications from accessing (linking) directly with system classes provided in the following packages and sub-packages thereof:

- `javacardx.spi.*`
- platform implementation-specific packages (including `com.sun.javacard.*` and `com.sun.javacardx.*`).

6.8.2 Package Sealing Checks

The Java Card Platform MUST support the JAR file package sealing mechanism. A package sealed within a JAR file specifies that all classes defined in that package MUST originate from the same JAR file. Otherwise, a `SecurityException` MUST be thrown.

A package may be sealed according to the optional `Sealed` attribute in the JAR Manifest file, defined for that package or defined by default for all packages in the JAR file. After a class loader loads a class from a sealed JAR file, classes in the same package can only be loaded from that JAR file.

6.8.3 Restriction on the Use of the `Class.forName` Method

The use of the `Class.forName` method by application code to dynamically load classes MUST be restricted. Only the classes declared in the Java Card Platform-specific descriptor of an application MUST be allowed to be dynamically loaded by that application, see [dynamically-loaded-classes Element](#) in [Chapter 8](#).

An application module class loader MUST implement the following security check: an attempt to dynamically load a class that is not declared in the Java Card Platform-specific descriptor of the application module loaded by that application module class loader MUST result in a `SecurityException` being thrown.

A group-library class loader MUST implement the following security check: an attempt to dynamically load a class that is not declared in the Java Card Platform-specific descriptor of any of the application modules loaded by one of its child application module class loader MUST result in a `SecurityException` being thrown.

This restriction MUST NOT apply to extension library code. The *extension library class loader* MUST allow for unrestricted use of the `Class.forName` method to dynamically load classes from other extension libraries or allowed system libraries.

6.9 Context Isolation Enhancements

This section describes the enhancements to the basic firewall-enforced context isolation mechanism on the Java Card Platform, Connected Edition:

- Additional permission-based security checks that apply when context switching to allow for a finer-grained access control during sharing across group contexts, see [Section 6.9.1, “Context Switches” on page 6-65](#).
- Application namespace enforcement based on tracking the active namespace associated with the current thread within and across group and owner contexts, [Section 6.9.2, “Application Namespace Enforcement” on page 6-65](#).
- Assignment of proper ownership to new objects created by objects whose ownership had been transferred to a new owner context using the *object ownership transfer mechanism*, [Section 6.9.3.1, “Ownership of Objects Created by Transferred Objects” on page 6-66](#).

Refer to [Section 2.4, “Context Isolation Basics” on page 2-14](#) for details on the basic firewall-enforced context isolation mechanism.

6.9.1 Context Switches

Before performing a context switch to another group context or to the Java Card RE context, the Java Card RE enforces security checks based on the protection domain of the currently active context, meaning the application requesting the context switch. See [Section 6.2.5.2, “Context-switch-triggered Security Checks” on page 6-20](#).

6.9.2 Application Namespace Enforcement

An application URI defines the root of a dedicated namespace within which all its resources are named. All resources of an application are named relatively to that application’s URI, see [Section 2.3.2, “Dedicated Application Namespaces” on page 2-6](#).

Additionally, each thread of control maintains distinct *active context* and *active namespace* references. See [Section 2.7.2.2, “Per-Thread Active Context and Active Namespace” on page 2-38](#).

The Java Card Platform MUST enforce the separation of application namespaces as follows:

- Relative URIs MUST be resolved relative to the current thread’s active namespace, meaning the currently executing application.

- Attempts to create or register resources under a namespace other than the current thread's active namespace, meaning under another application's namespace, MUST result in a security exception.

6.9.3 Ownership of Transferable Objects

The basic firewall-enforced context isolation mechanism (see [Section 2.4, “Context Isolation Basics” on page 2-14](#)) is enhanced with the *object ownership transfer mechanism*. This mechanism allows an application to transfer the ownership of objects it owns to applications in other contexts. See [Section 7.2, “Object Ownership Transfer Mechanism” on page 7-3](#) for more details on the ownership transfer mechanism.

In order to facilitate application instance deletion and to prevent dependencies on simple objects referenced by static fields of shared classes from blocking the deletion of the application instance that owns these objects, instances of explicitly transferable object classes, namely arrays and `String` objects, MUST be owned by the group context of the application that created them, not by the application itself. That is, such objects are not bound to any owner context but solely to a group context. Similarly, when transferred using the transfer of ownership mechanism, these objects are owned by the group context of the targeted recipient. Note that implicitly transferable objects are not bound to any context (group contexts or Java Card RE context) and, therefore, do not interfere with application instance deletion.

6.9.3.1 Ownership of Objects Created by Transferred Objects

When an explicitly transferable object is transferred from one owner context to another, the object is bound to the new owner context and any object created subsequently by the transferred objects MUST be bound to the same owner context as the transferred object's.

Implicitly transferable objects are not bound to any context (group contexts or Java Card RE context). Any object created and returned by an implicitly transferable object as a result of a call from an object bound to an owner context MUST be bound to that same owner context. For example, a call to an interned `String` object's `substring` method must return a `String` object bound to the owner context of the caller. The only exception is the `String.intern` method, which returns a canonical representation for the `String` object that is not bound to any context (group context or Java Card RE context). See [Section 2.7.2.2, “Per-Thread Active Context and Active Namespace” on page 2-38](#).

An implicitly transferable object MUST NOT hold in its state any reference to objects it created as a result of a call from an object bound to an owner context as this will break the immutability principle of such objects and prevent safe sharing.

Inter-application Communication

This chapter describes issues related to how applications communicate in the Connected Edition.

- [Security Containment Mechanisms](#)
- [Object Ownership Transfer Mechanism](#)
- [Shareable Interface Object-based Services](#)
- [Events](#)

7.1 Security Containment Mechanisms

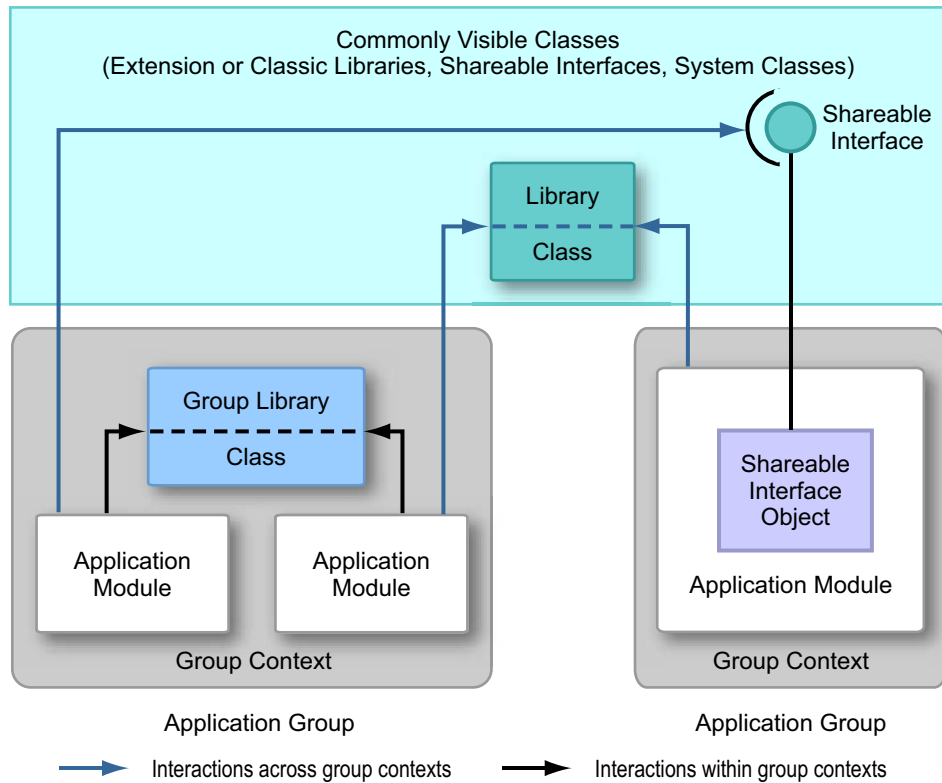
On the Java Card Platform, Connected Edition, communication between applications (web applications or applet applications) is constrained by two orthogonal security containment mechanisms:

- **Context isolation:** an application, along with the objects it creates (and, therefore, owns), are bound to a separate, protected object space called a *group context*. Applications from the same application group share the same group context. Applications from the same group context can directly communicate by invoking and passing commonly owned objects. Applications from different group contexts can only directly communicate using the shareable interface mechanism. See [Section 2.4, “Context Isolation Basics”](#) on page 2-14.
- **Code isolation:** classes from an application group are loaded by separate *class loaders*, which define a different *namespace* for each application group. Classes from one application group are not visible to another application group, but system classes and extension library classes are visible to all application groups. For applications to communicate directly through shareable interfaces, these interfaces must be commonly visible and must, therefore, have been loaded by a common parent class loader in the class loader delegation hierarchy. See [Section 6.7, “Code Isolation”](#) on page 6-57.

Applications from different application groups can communicate indirectly through static fields and methods of commonly visible classes or by the means of Java Card RE entry point objects and global arrays that are also, by definition, commonly visible. Context isolation also applies to objects exchanged by these means. See [Section 2.4, “Context Isolation Basics”](#) on page 2-14.

FIGURE 7-1 depicts the orthogonal security containment mechanisms and different means of inter-application communications.

FIGURE 7-1 Context Isolation, Code Isolation and Inter-Application Communications



To enhance communications across group contexts, the Java Card Platform, Connected Edition, introduces the *object ownership transfer mechanism*. This mechanism allows an application to transfer the ownership of objects it owns to other applications.

An application can communicate with another application using the following two inter-application communication facilities:

- **Services:** an application can publish shareable interface object-based services it wants to provide to other applications.

- Events: an application can notify other applications of particular conditions. These conditions are encapsulated in shareable interface object-based events.

These two inter-application communication facilities build on the shareable interface mechanism and the mechanism for transfer of objects' ownership.

7.2 Object Ownership Transfer Mechanism

The object ownership transfer mechanism allows an application to transfer the ownership of objects it owns to other applications. These objects can then be passed as parameters or return values of SIO methods calls. Furthermore, these objects can also be passed from the former owner to the new owner through static fields or as parameters or return values to static methods of commonly visible classes or by the means of Java Card RE entry point objects and global arrays.

Once an application has transferred the ownership of an object it owns to another application, the application that originally owned the object can no longer access it, even though it may still hold a reference to it. When an object's ownership is transferred to an application bound to a group context, all other applications from that same group context have access to the object.

7.2.1 Transferable Classes

Object ownership transfer MUST only apply to direct instances of a well-defined set of classes called *Transferable Classes*. Transferable classes can be categorized into two types:

- **Implicitly transferable (shared) classes** - 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. These classes are all immutable objects¹ - that is instances of these classes cannot be modified. These objects may be shared with other applications without any risk of concurrent and/or unexpected modifications.
- **Explicitly transferable classes** - Classes whose instances must have their ownership explicitly transferred to another application's group context in order to be accessible to that other application. These classes are typically mutable objects - that is instances of these classes can be modified after creation. These objects cannot be shared with other applications without risk of concurrent and/or unexpected modifications.

1. With the exception of `javacard.framework.CardException` and subclasses thereof. These classes support a setter method `setReason(short)` for backward compatibility reasons.

See [Section 6.9.3, “Ownership of Transferable Objects”](#) on page 6-66 for more details on the ownership assignment of instances of implicitly and explicitly transferable classes.

The Java Card Platform MUST support, and only support, the following list of implicitly transferable classes:

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.Class`
- `java.lang.Throwable` and API-defined subclasses thereof
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Short`
- `java.lang.String` (only literal strings and interned `String` objects).

The set of implicitly transferable classes that subclass the `Throwable` class MAY be extended by standards bodies provided these classes respect the contract of implicitly transferable classes on immutability defined above.

The Java Card Platform MUST support, and only support, the following list of explicitly transferable classes:

- `java.lang.String` (newly created strings, strings computed at run time)
- Arrays.

The ownership of the objects referenced by the components of an array MUST NOT be affected when the ownership of the array is transferred. Especially, since multi-dimensional arrays are arrays of arrays, when the ownership of a multi-dimensional array is transferred the ownership of the nested arrays MUST NOT be affected.

Strings computed at run time are not mutable objects but may encapsulate sensitive information such as passwords which must not be shared by default. Strings computed at run time MUST NOT be, by default, implicitly transferable. Such newly created strings MUST stay bound to their creator's group context until they are explicitly interned by calling their `intern()` method, in which case they MUST become context-free and may be passed and shared between group contexts without any firewall restrictions.

A call to the `intern()` method on a `String` object MUST return a canonical representation for the `String` object which is not bound to any context (group context or Java Card RE context). The `String` object may be added to the pool of strings maintained internally by the `String` class (and therefore common to all contexts). If the pool already contained a string equal to that `String` object, then the

string from the pool MUST be returned. Otherwise, the `String` object MUST be made context-free then added to the pool and a reference to that `String` object MUST be returned.

The Java Card Platform MUST NOT intern `String` objects passed as parameters to an API method by an application or computed at runtime from such parameters, if not otherwise stated in the API specification. An extension library SHOULD clearly state if any of its methods intern `String` objects passed as parameters or computed at runtime from such parameters.

The Java Card Platform MUST support the following `javacardx.framework.JCSystem` method for testing the transferability of an object:

- `isTransferable(Object)` - This method MUST return true only if the object passed as an argument is a direct instance of an implicitly or explicitly transferable class.

7.2.2 Transferring Object Ownerships

The Java Card Platform MUST support transferring objects' ownership by calls to the `javacardx.framework.JCSystem.transferOwnership` methods.

Applying transfer of ownership on instances of implicitly shared classes MUST have no effect on the instances' ownership.

Applying transfer of ownership to an array of objects MUST NOT affect the ownership of the objects referenced by the components of the array (transfer of ownership MUST be “shallow” as opposed to “deep”).

Attempting to transfer ownership of instances of classes that are not transferable classes MUST result in a security exception being thrown.

Attempting to transfer ownership of instances which are not direct instances of classes that are transferable classes MUST result in a security exception being thrown.

Attempting to transfer ownership of objects not owned by the caller's group context MUST result in a security exception being thrown.

Attempting to transfer ownership of objects to the Java Card RE context MUST result in a security exception being thrown.

Transfer of ownership MUST be subject to access control. When a `JCSystem.transferOwnership` method is invoked, a security check MUST ensure that the permission `javacardx.framework.ContextPermission` with the target canonicalized application URI and the action `transfer` is granted.

7.2.3 Defensive Copy

To facilitate interactions between server applications and their clients through shareable interfaces, the `javacardx.framework.JCSystem.copyTransferable` method allow for applications to create copies of objects they own and want to share only the state with other applications. Using this method, an application may create *Defensive Copies* of explicitly transferable objects, which are typically mutable objects, in order to only transfer the ownership of the copies. This allows for the application to continue processing with the original objects.

The copy operation performed by the `copyTransferable` method is a "shallow copy" of the object being copied, not a "deep copy" operation:

- On instances of explicitly transferable classes, the copy operation **MUST** behave as follows: it creates a new instance of the class of the object being copied and initializes all its fields with exactly the contents of the corresponding fields of the object, as if by assignment. The contents of the fields are not themselves duplicated.
- On instances of implicitly transferable classes, the copy operation **MUST** return the original object.

7.2.4 Thread Safety

Transfer of ownership **MUST** be synchronized on the object being transferred to avoid concurrency issues.

Note – If two threads from the same group context concurrently attempt to transfer the ownership of the same object, only the first one to acquire that object's lock will succeed transferring the object's ownership. The other thread after acquiring the object's lock will fail with a security exception because the object is no longer owned by an application in the same group context.

7.2.5 Transactional Behavior

Transfer of ownership **MUST** be an atomic operation. Once the ownership transfer to another application has been successfully performed, the object is owned and **MUST** stay owned by that other application even in the occurrence of a platform reset and that until it is explicitly transferred again or disposed of.

Note – An application should not expect the ownership of an object it formerly owned to be returned back, especially not automatically after a platform reset. For example, for performance reasons two inter-communicating applications may agree to reuse the same object, such as an array, to transfer data according to the following sequence: the client application allocates an array; the client application copies data to the array, transfers the ownership of the array to the server application and then passes the array to the server application by invoking one of its SIO methods; the server application copies the data from the array, transfers the ownership of the array back to the client application and returns from the SIO method call. The client application that initially allocated and owned the object should account for situations where the object's ownership is not returned, such as when a platform reset occurs before the server application transfers back the ownership to the client application.

7.3 Shareable Interface Object-based Services

The inter-application communication facility provided by the Java Card Platform, Connected Edition, extends the classic shareable interface mechanism and allows for all application models, including classic applet applications, extended applet applications and web applications, to interact through *shareable interface object-based services* in a unified way. This facility also ensures a seamless integration of classic SIO registration and lookup.

The inter-application communication facility allows for:

- Applications to define and dynamically register (and unregister) SIO-based services
- Applications to lookup SIO-based services registered by other applications

Web applications and extended applet applications can lookup and register SIO-based services to a global service registry. Classic applet applications cannot use this facility directly. To ensure the seamless integration of the classic SIO registration and lookup, the service registry lookup mechanism uses the classic Java Card RE SIO lookup mechanism when an AID-based service URI cannot be found in the service registry.

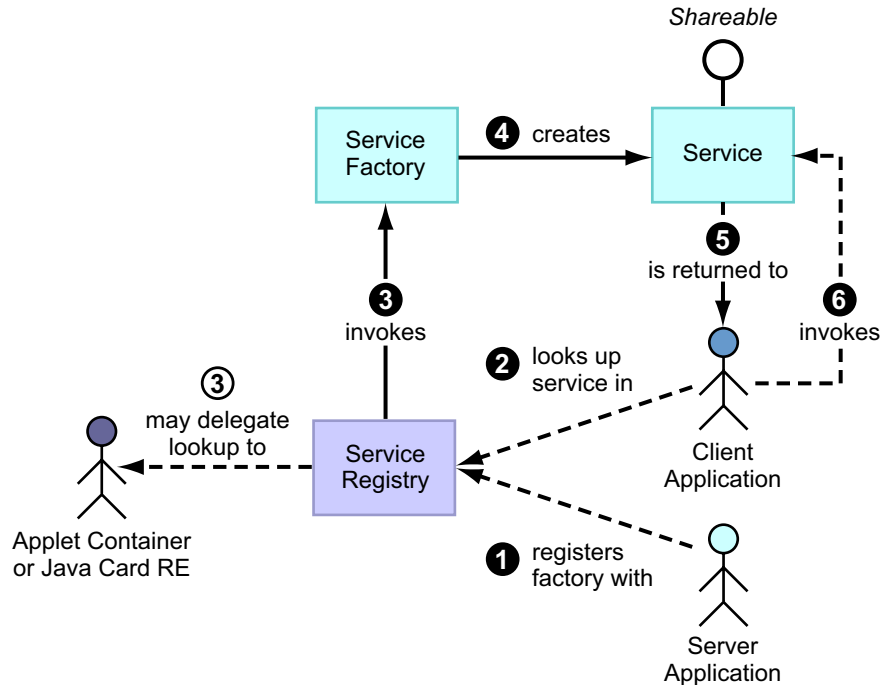
The `javacardx.facilities.ServiceRegistry` class allows for applications to register SIO-based services they want to expose to other applications. Service-providing applications (server applications) may typically, but not necessarily, run in group contexts different from those of the client applications.

The `ServiceRegistry` instance **MUST** be a singleton. It **MUST** be a permanent Java Card RE entry point object. It is retrieved by calling the `ServiceRegistry.getServiceRegistry` method.

An application **MUST NOT** be allowed to use the service registry until it is registered, that is until it has been assigned an application URI. In particular, an extended applet **MUST NOT** be allowed to use the `ServiceRegistry` singleton instance within its `install` method until it is duly registered by a call to its `register` method.

FIGURE 7-2 depicts the interactions between the different components and actors of the SIO-based service registry. The numbering indicates a typical interaction sequence with possible alternatives. Colors indicate common ownership of interacting objects.

FIGURE 7-2 SIO-based Service Registry Interactions (Collaboration Diagram)



7.3.1 SIO-based Service Definition and Identification

Services **MUST** be shareable interface objects (SIO). They **MUST** implement interfaces that extend the `javacard.framework.Shareable` interface.

A service MUST be uniquely identified in the registry with *service URI*. Service URIs are in the *service namespace* and MUST use the URI scheme `sio`. The service namespace MUST be partitioned as follows:

- **The standard service sub-namespace** - for standard services that may provide well-defined services.
- **The platform service sub-namespace** - for well-defined platform service (currently not used for service registration, reserved for future use).
- **Application-defined service sub-namespaces** - for services that an application may define.

Only one service may be associated with a service URI.

The service registry MUST implement the following requirements:

- All service URI and server URI (application URI) matching operations MUST apply on canonicalized forms of the service URIs, that is URIs MUST have first been resolved, then normalized. See [Section , “Resolution of Relative URI References” on page 2-9](#) and [Section , “Default Normalization of URIs” on page 2-10](#).

As a result of service URI normalization, standard service URIs that use the default registry-based URI authority and those that use the `aid` registry-based URI authority are equivalent. For example, the two following standard service URIs designate the same entry in the registry:

```
sio:///standard/auth/holder/global/owner/fingerprint,  
sio://aid/standard/auth/holder/global/owner/fingerprint.
```

7.3.1.1 Platform and Standard Services

The Java Card Platform, Connected Edition, does not currently define any platform services. The `/platform` namespace is currently not used for service registration and is reserved for future use.

An application MUST NOT be allowed to register a service under the reserved platform service namespace rooted at `sio:///platform/`.

The standard service namespace rooted at `sio:///standard/` specifies an extensible set of standard services that any application may implement. The intent for this namespace is to define a set of service URIs for common, well-defined services.

The standard authentication service namespace rooted at `sio:///standard/auth/` defines standard services for card holder and other user authentication. See [Section 6.4.1, “Scheme-specific Authenticators” on page 6-30](#) for more details about standard authentication services. See [TABLE 7-1](#) for the list of standard authentication services a Java Card Platform MUST support. The definition of additional authentication service URIs in the standard authentication service

namespace is reserved for this specification. In particular, an application **MUST** not be able to register a service with a URI in the standard authentication namespace that is not one defined by this specification.

An application, if authorized, **MUST** be allowed to register a service under the standard service namespace rooted at `sio:///standard/`.

The set of standard services **MAY** be extended by standards bodies. It is suggested that the service URI path, relative to the standard service root URI `sio:///standard/`, be prefixed in accordance with the reverse domain name convention for package naming suggested by *The Java Programming Language Specification*. The prefixes `javacard` and `javacardx` are reserved by this specification.

TABLE 7-1 Standard SIO-based Services

Service Description	Service URI	Service Interfaces
Global card holder authenticator	<code>sio:///standard/auth/holder/global/<auth-service-path></code>	<code>javacardx.framework.SharedPINAuth</code> <code>javacardx.framework.SharedPasswordAuth</code> <code>javacardx.framework.SharedBioTemplateAuth</code>
Session-scoped card holder authenticator	<code>sio:///standard/auth/holder/session/<auth-service-path></code>	<code>javacardx.framework.SharedPINAuth</code> <code>javacardx.framework.SharedPasswordAuth</code> <code>javacardx.framework.SharedBioTemplateAuth</code>
Session-scoped user authenticator	<code>sio:///standard/auth/user/session/<auth-service-path></code>	<code>javacardx.framework.SharedPINAuth</code> <code>javacardx.framework.SharedPasswordAuth</code> <code>javacardx.framework.SharedBioTemplateAuth</code>
Standards body-defined service	<code>sio:///standard<service-path-prefix>/<service-path></code>	implements <code>javacard.framework.Shareable</code>

See [TABLE 7-1](#) for the standards body-defined service naming convention a Java Card Platform **MUST** support.

7.3.1.2 Application-defined Services

An application may define, meaning register, services in the application-defined service namespace. Application-defined services **MUST** be named relatively to their *application's root service URI*. For example, if an application's URI is `/transit/pos`, the application's root service URI is `sio:///transit/pos` and a well-formed service URI could be `sio:///transit/pos/ticketbook`.

An application, if authorized, may register a service under its own service namespace rooted at `sio://<app-path>/`, where `<app-path>` represents the application's URI path, either a web application URI represented by its context path or an applet application URI represented by its RID and PIX AID components. See [TABLE 7-2](#) for the application-defined service naming convention a Java Card Platform MUST support.

An application MUST NOT be allowed to register a service under a URI rooted in another application's namespace.

TABLE 7-2 Application-defined Services

Service Description	Service URI	Service Interface
Application-defined service	<code>sio://<app-path>/<service-path></code> :	<i>implements</i>
	• <code>sio://<context path>/<service-path></code>	<code>javacard.framework.Shareable</code>
	• <code>sio://aid/<RID>/<PIX>/<service-path></code>	

7.3.2 SIO-based Service Factory Registration

7.3.2.1 Registration

To provide a service to other applications, an application MUST register a *service factory* under a unique service URI. Service factory objects MUST implement the `javacardx.facilities.ServiceFactory` interface.

The application MUST call the `ServiceRegistry.register` method and provide a *legal* service URI and a service factory object to register the service.

The service URI passed for registration MUST be *legal* for the application or a security exception is thrown. A *legal* service URI for registering a service is of one of the following types (after canonicalization):

- an absolute URI rooted in the application's own service namespace
- an absolute URI rooted in the standard service namespace

Registering a service factory with a service URI already used by the same application MUST result in the new registration overwriting the old one.

The service registry MUST support the following requirements:

- the same service factory object may be registered under multiple service URIs
- only one service may be registered with one service URI

Registering a service factory with a standard service URI already used by another application MUST result in a security exception being thrown.

Registration of service factories MUST be subject to access control. When the `ServiceRegistry.register` method is invoked, a security check MUST ensure that the permission `javacardx.facilities.ServiceRegistryPermission` with the target canonicalized service URI and the action `register` is granted.

Applications may register service factories anytime during their lifetime².

7.3.2.2 Unregistration

An application may unregister a service factory it previously registered at any time. To unregister a service factory, an application MUST provide the service URI with which the service factory was registered.

The application MUST call the `ServiceRegistry.unregister` method and provide a *legal* service URI.

The service URI passed for unregistration MUST be *legal* for the application or a security exception is thrown. A *legal* service URI for unregistering a service is of one of the following types (after canonicalization):

- an absolute URI rooted in the application's own service namespace
- an absolute URI rooted in the standard service namespace

Attempting to unregister a service factory registered by another application, either under its own namespace or under the standard service namespace, MUST result in a security exception being thrown.

Unregistration of service factories MUST be subject to access control. When the `ServiceRegistry.unregister` method is invoked, a security check MUST ensure that the permission `javacardx.facilities.ServiceRegistryPermission` with the target canonicalized service URI and the action `unregister` is granted.

Note – An application developer must account for other applications still holding references to instances of a service after the service's factory has been unregistered. A client application may still use a service even though the service's factory has been unregistered.

Applications may unregister service factories anytime during their lifetime³. Attempting to unregister a service factory that is not, or is no longer, registered has no effect.

2. After they have been duly registered and assigned an application URI.

3. After they have been duly registered and assigned an application URI.

7.3.3 SIO-based Service Lookup

An application may look up a service registered under its own service namespace, by any other application under that application's own namespace or the standard service namespace, provided it has the required permissions. To look up a service, an application **MUST** provide the URI of the service and an optional object parameter.

The service registry **MUST** look up the registered service factory and it **MUST** invoke its `create` method with the service URI and the optional object parameters. The `create` method executes within the group context of the owning application, meaning the application which registered the service factory. The service factory may return a new instance or the same instance at each invocation. The service factory may use the service URI and the optional object parameters to determine the service instance to create and initialize and/or return, such as when the same service factory object has been registered under multiple service URIs.

The application **MUST** call one of the `ServiceRegistry.lookup` methods and provide a service URI. The value of the optional lookup parameter may typically be an SIO or an implicitly transferable object.

Lookup of services **MUST** be subject to access control. When the `ServiceRegistry.lookup` methods are invoked, a security check **MUST** ensure that the permission `javacardx.facilities.ServiceRegistryPermission` with the target canonicalized service URI and the action `lookup` is granted.

7.3.3.1 Classic Applet SIO Integration

The registry **MUST** delegate to the classic Java Card RE SIO lookup mechanism if all the following conditions apply:

- the looked up service URI does not map to any registered service factory
- the calling application (the client) is an applet application
- the looked up service URI translates to an AID⁴
- the optional data parameter is provided⁵, is not `null` and is assignable to a byte primitive type, that is it is a `java.lang.Byte` instance

The delegation to the classic Java Card RE SIO lookup mechanism **MUST** be equivalent to invoking the `JCSYSTEM.getAppletShareableInterfaceObject` method with the AID instance corresponding to the looked up service URI and the optional data parameter converted to a byte primitive type. See *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition*.

4. An AID URI with a / path component such as `sio://aid/A000000062/03010C0D01/`.

5. If the optional data parameter is not provided it is assumed to be `null` by default and therefore the delegation to the classic Java Card RE SIO lookup mechanism will not be performed

If any of the above conditions does not apply, the lookup operation **MUST** return as if no service factory was registered for the looked up service URI.

This fallback mechanism does not apply to other registry operations.

This mechanism primarily allows for the integration of classic applet applications. Extended applet applications can look up classic applet server applications. Extended applet applications can also use this mechanism to look up other extended applet applications that expose shareable interfaces via the `Applet.getShareableInterfaceObject` method.

Note – Backward compatibility support on the Java Card Platform, Connected Edition, includes mechanisms that guarantee the thread-safety of classic applet applications when invoked through SIO methods calls from concurrently executing extended applet applications and web applications⁶.

Classic applet applications may also look up extended applet server applications, provided they expose Java Card Platform, v2.2.2-compatible shareable interfaces via the `Applet.getShareableInterfaceObject` method.

Note – Classic applet application developers and classic applet application providers must account for the issues, such as race conditions and lock-induced delays, related to transitioning from the single threaded environment of classic applet applications to the extended applet and web application multi-threaded environment.

7.3.4 Role-based Security for SIO-based Services

The role-based security API for SIO-based services consists of the following methods of the `javacardx.framework.JCSystem` class:

- **The `isUserInRole` method:** this method determines if the authenticated identity associated with the current thread is in a specified user security role. See [Section 6.3.1.2, “Programmatic User Role-based Security” on page 6-24](#) for more details.
- **The `isClientInRole` method:** this method determines if the client application (as per the semantic of the `javacardx.framework.JCSystem.getClientURI` method, see [Section 7.3.7, “Per-Thread Active Context” on page 7-15](#)) is in a specified client security role. See [Section 6.3.2.1, “Programmatic Client Role-based Security” on page 6-27](#) for more details.

⁶ Web applications cannot look up Classic applet SIOs but they may be passed such SIOs by Extended applet applications.

These methods may be invoked from within the method of any SIO (SIO-based services and other unregistered SIOs) or code that it calls, such as library code. These methods may also be invoked from within the `create` method of an SIO-based service factory to check the authorization to access a service prior to returning it to the client.

7.3.5 Lifetime and Persistence of SIO-based Services

A service factory **MUST** remain registered until it is removed from the registry by the application that registered it or when that application is forcefully deleted. See [Section 8.1, “The Card Manager Application” on page 8-1](#) for more details on application instance deletion.

The registry **MUST** ensure that service factory objects are persistent across card sessions. Therefore, applications do not have to hold on to references to these objects to ensure their persistence.

7.3.6 Thread Safety

The `ServiceRegistry` implementation **MUST** be thread safe to account for concurrent registrations, unregistrations and lookups. Though, it **SHOULD** not serialize the calls to the `lookup` methods. A service factory object may, therefore, be concurrently looked up. The application developer must properly account for thread safety in the `create` method of the service factory.

7.3.7 Per-Thread Active Context

Each thread is associated to a current active context and namespace. When entering a different group context through an SIO method call, the thread's current active context and namespace **MUST** be changed to the owner context of the SIO. Upon return to its previous group context, the thread's current active context and namespace **MUST** be restored. See [Section 2.7.2.2, “Per-Thread Active Context and Active Namespace” on page 2-38](#).

Note – The current active namespace, when executing a method of an SIO-based service, may not always correspond to the namespace of the application that registered the service because an application may register under its own namespace, or the standard namespace, a service object owned by another application from the same group context. Note though, that the current active namespace always corresponds to the namespace of one of the applications from the same group context as that of the application which registered the service.

7.3.8 Transactional Behavior

The `ServiceRegistry` singleton instance **MUST** be a permanent Java Card RE entry point object and therefore **MUST NOT** participate in any application transaction. The `ServiceRegistry` class **MUST** be annotated with the `NOT_SUPPORTED TransactionType` annotation, see [Section 2.9.2, “Transaction Demarcation” on page 2-47](#). All registration operations **MUST** nevertheless be atomic and **MUST** ensure that the service registry is at all time consistent.

7.4 Events

The event notification facility provided by the Java Card Platform, Connected Edition, allows for applications to be notified by the Java Card RE, or by other applications, about particular conditions. When these conditions occur, they are encapsulated in objects called *events*. The event notification facility builds on the inter-application communication facility and allows for web applications and applet applications to communicate asynchronously⁷ with each other through events.

The event notification facility allows for:

- Applications to dynamically register and unregister for notification of other applications’ events and platform events
- Applications to define and fire events

Web applications and extended applet applications can use a global event registry to fire events and/or register for event notification. Classic applet applications cannot use this facility directly.

7. Events are delivered to an event-consuming application asynchronously, that is in a thread different from its main request or command processing threads. Additionally an event-producing application may choose not to wait for an event it fires to be delivered to all the consuming applications, therefore notifying the consuming applications asynchronously from its own thread.

The `javacardx.facilities.EventRegistry` class allows for applications to fire events and/or register for notification of events fired by the Java Card RE or other applications. Applications firing events (event-producing applications) may typically, but not necessarily, run in group contexts different from those registering for notification of these events (event-consuming applications).

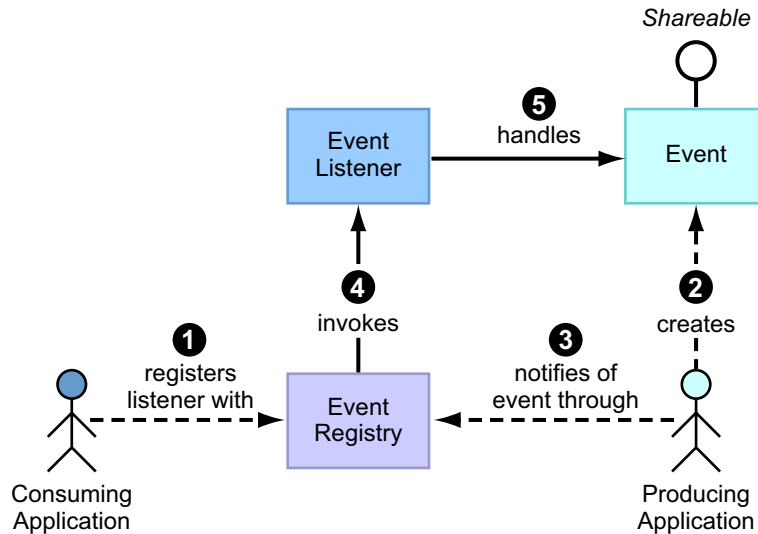
The `EventRegistry` instance **MUST** be a singleton. It **MUST** be a permanent Java Card RE entry point object. It is retrieved by calling the `EventRegistry.getEventRegistry` method.

The event registry **MUST** operate in the Java Card RE context, therefore, it **MUST** be defined as a Java Card RE entry point object.

An application **MUST NOT** be allowed to use the event registry until it is registered, that is until it has been assigned an application URI. In particular, an extended applet **MUST NOT** be allowed to use the `EventRegistry` singleton instance within its `install` method until it is duly registered by a call to its `register` method.

FIGURE 7-3 depicts the interactions between the different components and users of the event registry. The numbering indicates a typical interaction sequence. Colors indicate common ownership of interacting objects.

FIGURE 7-3 Event Registry Interactions (Collaboration Diagram)



7.4.1 Event Definition and Identification

Events are shareable interface objects (SIO). They **MUST** extend the `javacardx.facilities.Event` class, which implements the `javacardx.facilities.SharedEvent` interface (which itself extends the `javacard.framework.Shareable` interface).

An event type **MUST** be uniquely identified in the registry with an *event URI*. Event URIs are in the event namespace and **MUST** use the URI scheme `event`. The event namespace **MUST** be partitioned as follows:

- **The standard application and resource event sub-namespaces** - for standard events with a well-defined semantic.
- **The platform event sub-namespace** - for well-defined platform events.
- **Application-defined event sub-namespaces** - for events that an application may define.

Each event object **MUST**, at the minimum, encapsulate the URI identifying the event and its source, the application that created it.

The URI identifying the source of an application-defined, a standard application, or a resource event **MUST** be in the namespace of the application that created it or on behalf of which the event was created and fired by the platform. The event's source **MUST NOT** be overwritable.

The URI identifying the source of a platform event MUST designate the platform and MUST be `///platform`.

The source of an event may be retrieved by calling the `Event.getSourceURI` method.

Note – The source of an event may be different from the owner of the event when the event source identifies an application’s resource or when the event is a standard application event that was fired by the platform on behalf of an application.

Note – The event notification facility builds on the Java SE event framework classes `java.util.EventObject` and `java.util.EventListener`. Note though that the source of an event as represented by the source URI attribute and retrieved by calling the `Event.getSourceURI` method is different from the source object attribute of the `EventObject` base class as retrieved by calling the `EventObject.getSource` method. The source URI identifies the application or application resource the event occurred upon, while the source object MUST always be set to the `EventRegistry` singleton instance.

The URI identifying an event object MUST be assigned to the object at instantiation-time and MUST NOT be overwritable. The URI of an event may be retrieved by calling the `Event.getURI` method.

The event registry MUST implement the following requirements:

- All event URI and source URI matching operations MUST apply on canonicalized forms of the event URIs. Therefore, URIs MUST first be resolved, then normalized. See [Section , “Resolution of Relative URI References” on page 2-9](#) and [Section , “Default Normalization of URIs” on page 2-10](#).

As a result of event URI normalization, platform event URIs, as well as standard application and resource event URIs that use the default registry-based URI authority, and those that use the `aid` registry-based URI authority are equivalent. For example, the two following platform event URIs designate the same entry in the registry: `event:///platform/clock/resynced` and `event://aid/platform/clock/resynced`.

7.4.1.1 Platform and Standard Events

The platform event namespace rooted at `event:///platform/` defines a set of platform events that encapsulate specific Java Card RE conditions. This includes a card lifecycle event such as a real-time clock resynchronization. See [TABLE 7-3](#) for the list of platform events a Java Card Platform MUST support.

Instances of `Event` that encapsulate platform events **MUST** be owned by the Java Card RE.

The standard event namespace rooted at `event:///standard/` defines an extensible set of standard events that any application may fire or that may be fired by the platform on behalf of an application. This namespace is intended to define a set of event URIs for common, well-defined conditions.

The standard application and resource event namespaces rooted at `event:///standard/app/` and `event:///standard/rsrc/`, respectively, define a set of event URIs for common, well-defined application and resource-related conditions, such as application and resource lifecycle events. See [Section 8.1, “The Card Manager Application” on page 8-1](#) for more details about application lifecycle events. See [TABLE 7-3](#) for the list of standard application and resource events a Java Card Platform **MUST** support. The definition of additional event URIs in the standard application and resource event namespaces is reserved for this specification. In particular, an application **MUST** not be able to create or fire an event with a URI in the standard application or resource event namespaces that is not one defined by this specification.

Instances of `Event` created by the Java Card RE on behalf of an application and that encapsulate standard application and resource events **MUST** be owned by the Java Card RE.

An application, if authorized, may fire an event under the standard event namespace rooted at `event:///standard/`.

An application **MUST NOT** be allowed to create or fire an event under the platform event namespace rooted at `event:///platform/`.

The set of standard events **MAY** be extended by standards bodies. It is suggested that the event URI path, relative to the standard event root URI `event:///standard/`, be prefixed in accordance with the reverse domain name convention for package naming suggested by *The Java Programming Language Specification*. The prefixes `javacard` and `javacardx` are reserved by this specification.

See [TABLE 7-3](#) for standards body-defined event naming conventions the Java Card Platform MUST support.

TABLE 7-3 Platform and Standard Events

Event Description	Event URI	Event Sources
Clock resynchronization	event:///platform/clock/resynced	///platform
Application instance creation	event:///standard/app/created	/// <i><context path></i> * //aid/<RID>/<PIX>
Application instance deletion	event:///standard/app/deleted	/// <i><context path></i> //aid/<RID>/<PIX>
Application instance deletion request	event:///standard/app/deleting	/// <i><context path></i> //aid/<RID>/<PIX>
Resource creation	event:///standard/rsrc/created	/// <i><context path></i> / <i><rsrc-path></i> //aid/<RID>/<PIX>/<rsrc-path> ///platform/<rsrc-path>
Resource update	event:///standard/rsrc/updated	/// <i><context path></i> / <i><rsrc-path></i> //aid/<RID>/<PIX>/<rsrc-path> ///platform/<rsrc-path>
Resource deletion	event:///standard/rsrc/deleted	/// <i><context path></i> / <i><rsrc-path></i> //aid/<RID>/<PIX>/<rsrc-path> ///platform/<rsrc-path>
Standards body-defined event	event:///standard/ <i><event-path-prefix></i> / <i><event-path></i>	/// <i><context path></i> / <i><rsrc-path></i> //aid/<RID>/<PIX>/<rsrc-path> ///platform/<rsrc-path>

* *<context-path>* corresponds to a web application URI which exactly corresponds to its context path.

7.4.1.2 Application-defined Events

An application may define events in the application-defined event namespace. Therefore, application-defined events MUST be named relatively to their *application's root event URI*. For example, if an application URI is */transit/pos*, the application's event root URI is *event:///transit/pos* and a well-formed event URI could be *event:///transit/pos/ticketbook/overdraft*. See [TABLE 7-4](#) for the application-defined event naming convention a Java Card Platform MUST support.

The base Event class can be used by applications to notify about events that do not have to encapsulate domain-specific information and behavior. When an event object must encapsulate domain-specific information and behavior, a new event object must be defined by extending the Event class.

An application, if authorized, may fire an event under its own event namespace rooted at event : // <app-path> /, where <app-path> represents the application's URI path, either a web application URI represented by its context path or an applet application URI represented by its RID and PIX AID components.

An application **MUST NOT** be allowed to fire an event under a URI rooted in another application's namespace.

TABLE 7-4 Application-defined Events

Event Description	Event URI	Event Classes (or base classes)
Application-defined event	event : // <app-path> / <event-path>: <ul style="list-style-type: none"> • event : // <context path> / <event-path> • // aid / <RID> / <PIX> / <event-path> 	javacardx.facilities.Event

7.4.2 Event Listener Registration

7.4.2.1 Registration

An application may register for notification of any event, fired either by the platform or any other applications, provided it has the required permissions. To register its interest in an event using the event notification facility, an application **MUST** register an *event notification listener* for that event's URI. Event listener objects **MUST** implement the `javacardx.facilities.EventNotificationListener` interface.

The registering application **MUST** call one of the `EventRegistry.register` methods and provide:

- An optional event source URI identifying an application, a resource or the platform
- An exact or a path-prefix URI pattern identifying a set of events
- A listener object, which will handle the event notification

Registering several event listener objects for the same source and event URIs **MUST** result in each of the listeners being called upon event notification.

Registering the same event listener object several times for the same source and event URIs **MUST** result in the listener being called only once upon event notification.

The same event listener object may be registered for multiple source and event URIs.

The event notification facility **MUST** allow for applications to register for events produced by an event source (an application or an application's resource) that is not yet installed. When an event source is uninstalled, the listeners of events from this source registered by other applications **MUST** stay registered.

Registration of event listeners **MUST** be subject to access control. When the `EventRegistry.register` method is invoked, a security check **MUST** ensure that the permission `javacardx.facilities.EventRegistryPermission` with the target canonicalized event URI and the action `register` is granted.

Applications may register event listeners anytime during their lifetime⁸.

7.4.2.2 Unregistration

An application may unregister an event listener it previously registered at any time.

The application **MUST** call one of the `EventRegistry.unregister` methods and provide:

- An optional event source URI identifying an application, a resource or the platform
- An exact or path-prefix event URI pattern identifying a set of events
- A listener object, which was registered to handle the event notification

Only event listeners previously registered by the calling (current) application for the specified source and event URIs **MUST** be unregistered. The event listener passed may be null, in which case all listeners registered by the calling (current) application for that source and event URIs are removed.

Unregistration of event listeners **MUST** be subject to access control. When the `EventRegistry.unregister` method is invoked, a security check must ensure that the permission `javacardx.facilities.EventRegistryPermission` with the target canonicalized event URI and the action `unregister` is granted.

Applications may unregister event listeners anytime during their lifetime⁹. Attempting to unregister an event listener that is not, or is no longer, registered has no effect.

7.4.3 Event Notification

An application may fire an event with a URI from its own event namespace or the standard event namespace, provided it has the required permissions.

8. After they have been duly registered and assigned an application URI.

9. After they have been duly registered and assigned an application URI.

The application **MUST** call one of the `EventRegistry.notifyListeners` methods and provide an instance of the `Event` class with an event source URI that identifies the firing application or a resource in that application's namespace, and an event URI, which identifies the event type, a standard event or an application-defined event in the firing application's event namespace.

The URI of the event passed for notification **MUST** be *legal* for the application or a security exception is thrown. A *legal* event URI is of one of the following types (after canonicalization):

- An absolute event URI rooted in the application's own event namespace
- An absolute URI rooted in the standard event namespace

The event registry **MUST** look up all event listeners registered for that event's source URI and event URI and it **MUST** invoke each listener's `notify` method in sequence passing the event as a parameter. The `notify` methods execute within the group context of their owning applications, meaning the applications which registered the event listeners. The order in which the event listeners may be notified is nondeterministic.

Event notification **MUST** be subject to access control. When an `EventRegistry.notifyListeners` method is invoked, a security check **MUST** ensure that the permission `javacardx.facilities.EventRegistryPermission` with the target canonicalized event URI and the action `notify` is granted.

7.4.4 Role-based Security for Events

Since events (application-defined events and standard application and resource events) are shareable interface objects, role-based security for events uses the same API as for role-based security for SIO-based services, see [Section 7.3.4, "Role-based Security for SIO-based Services"](#) on page 7-14.

In addition, the `EventRegistry.notifyListenersInRole` methods allow for specifying a client security role. The notification process is identical to the one described in [Section 7.4.2, "Event Listener Registration"](#) on page 7-22 with the difference that only the listeners belonging to the client security role and to the notifying application itself **MUST** be notified.

7.4.5 Lifetime and Persistence of Event Listeners

An event listener **MUST** remain registered until it is removed from the registry by the application that registered it or when that application is forcefully deleted. See [Section 8.1, “The Card Manager Application” on page 8-1](#) for more details on application instance deletion.

The registry **MUST** ensure that event listener objects are persistent across card sessions. Therefore, applications do not have to hold on to references on these objects to ensure their persistence.

7.4.6 Thread Safety

The `EventRegistry` implementation **MUST** be thread safe to account for concurrent registrations, unregistrations and notifications. However, it **SHOULD** not serialize the calls to the `notifyListeners` methods. An event listener object may, therefore, be concurrently invoked. The application developer must properly account for thread safety in the `notify` method of the event listener.

Event listener notifications **MUST** be handled by the event registry in Java Card RE-owned threads. The thread notifying a particular application-owned listener **MUST** belong to the Java Card RE context and **MUST**, therefore, switch to the application's group context when calling the event listener's `notify` method. A single Java Card RE-owned thread **MUST** handle the notification of all the listeners for a single event (a single call to an `EventRegistry.notifyListeners` method). All concurrent calls to the `EventRegistry.notifyListeners` methods **MUST** be handled concurrently in separate threads. An exception thrown by an event listener during a notification **MUST** be caught by the Java Card RE-owned notification thread and **MUST NOT** result in the notification of the remaining listeners being aborted.

By default, the event-producing application's thread **MUST** wait until the event notification is complete, meaning all listeners have been notified. An application may choose not to wait for all listeners to be notified by setting the `noWait` argument of the `EventRegistry.notifyListeners` method to `true`. When this occurs, the `EventRegistry.notifyListeners` method will immediately return and the notification will be handled asynchronously by a Java Card RE-owned event notification thread.

7.4.6.1 Special Case of Application Instance Deletion Event and Other Platform and Card Management Events

Because the notification of all the listeners for a single event (a single call to an `EventRegistry.notifyListeners` method) is performed in sequence (serially), a listener not returning from its `notify` method may block the event notification and, therefore, some listeners might not get notified until the blocking listener returns.

A Java Card Platform implementation MAY limit the effects of an unresponsive listener application when notifying of some standard application (card management-related) events or platform events by performing the notification of all or some of the listeners asynchronously. The sequence of notification can, therefore, not be guaranteed on card management-related standard application or platform events. However, a Java Card Platform implementation MUST guarantee that all applications which registered for such an event have had at least one of their registered event listeners invoked. A Java Card Platform implementation MUST do its best to notify all listeners of all applications.

Asynchronous notification MAY be used by a Java Card Platform implementation to prevent an unresponsive listener application from interfering with the application instance deletion sequence detailed in [Section 8.9, “Deletion of Application Instance” on page 8-33](#).

[TABLE 7-5](#) lists all card management-related standard application or platform events for which the notification is not guaranteed to be sequential and complete.

TABLE 7-5 Platform and Card Management-related Standard Application Events

Event Description	Event URI
Clock resynchronization	<code>event:///platform/clock/resynced</code>
Application instance creation	<code>event:///standard/app/created</code>
Application instance deletion	<code>event:///standard/app/deleted</code>
Application instance deletion request	<code>event:///standard/app/deleting</code>

7.4.7 Per-Thread Active Context

Events are shareable interface objects (SIO). Therefore, calling the event methods results in context-switching to the group context of the application that fired the event or to the Java Card RE context, if the event was fired by the platform. See [Section 7.3.7, “Per-Thread Active Context” on page 7-15](#) for more details.

Note – The current active namespace when executing a method of an SIO-based event may not always correspond to the namespace of the application that fired the event as an application may fire under its own namespace or the standard namespace an event object owned by another application from the same group context. Note though, that the current active namespace will always correspond to the namespace of one of the applications from the same group context as that of the application which fired the event.

Note – A call from within an event listener's `notify` method to the `javacardx.framework.JCSystem.getClientURI` or `javacardx.framework.JCSystem.getPreviousURI` method will not return the application URI of the application that fired the event. It will return null. To retrieve the source of the event (which either identifies the platform, an application or an application's resource), the `Event.getSourceURI` method must be called. To retrieve the URI of the application that created and fired the event, the `javacardx.framework.JCSystem.getServerURI(Shareable)` method must be called with the event as the parameter. If the event is a platform event or the event is a standard application event that was fired by the platform on behalf of the application, that call will return null.

7.4.8 Transactional Behavior

The `EventRegistry` singleton instance is a permanent Java Card RE entry point object and therefore **MUST NOT** participate in any application transaction. The `EventRegistry` class **MUST** be annotated with the `NOT_SUPPORTED TransactionType` annotation, see [Section 2.9.2, “Transaction Demarcation” on page 2-47](#). All registration operations **MUST** nevertheless be atomic and **MUST** ensure that the event registry is at all time consistent.

Card Management

The requirements of a card manager on the Connected Edition are described in this chapter.

This chapter describes the following aspects of card management:

- [The Card Manager Application](#)
- [The Card Management Facility](#)
- [Unit of Distribution and Deployment](#)
- [Distribution Formats](#)
- [Descriptor Formats](#)
- [Loading Application Modules](#)
- [Loading Libraries](#)
- [Creation of Application Instances](#)
- [Deletion of Application Instance](#)
- [Unloading of Deployment Units](#)

8.1 The Card Manager Application

The card manager application is an on-card application that manages the secure loading, configuring and deleting of applications and libraries on the Java Card platform. The card manager application MAY be a web application or an extended applet application. A Java Card platform may support multiple card manager applications.

On a Java Card platform that supports post-issuance download, an off-card client MUST be able to establish a communication session with the card manager application. The actual communication dialog used by the off-card client to communicate with the card manager application is implementation specific. On a

Java Card platform that supports post-issuance download, the card manager application MUST be able to perform the following operations upon request from the off-card client:

- Enforce security for card management operations:
 - Check and authorize operations based on the off-card client credentials
- Report status on the card management operations
- List the loaded applications and libraries

In addition, the card manager application relies on the card management facility of the Java Card Platform to perform the card management operations, the loading and unloading of applications and the instantiating and deleting of application instances. The card manager application interacts with the card management facility in an implementation specific manner. The *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition* includes an informative section that contains the recommended Card Management System Programming Interface between the card manager application and the card management facility.

8.2 The Card Management Facility

The card management facility is the Java Card platform layer responsible for securely adding and removing application code and instances onto the platform. The card management facility MUST check if the caller has card management application permissions (see `CardManagementPermission` in [Section 6.2, “Permission-based Security” on page 6-4](#)), based on the caller’s protection domain, prior to performing any card management functions. On a Java Card platform that supports post-issuance download, the card manager facility MUST be able to perform the following operations upon request from the card manager application:

- Load deployment units
- Create application instances
- Delete application instances
- Delete deployment units
- Manage information about deployment units and application instances on the card

8.3 Unit of Distribution and Deployment

An application is distributed and deployed as an application module JAR file.

Libraries are distributed and deployed as standard library JAR files containing the library classes.

Applications and libraries **MUST** be distributed in the public domain in the formats described in this specification. An application module or library **MAY** be transformed during deployment on the card only if the deployment format is visible within a private network domain and nowhere else. See “Public Representation of Java Applications and Resources” in the *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Connected Edition*.

Thus, the card manager **MUST** support three types of distribution and deployment units. They are:

- The application module distribution format JAR file
 - This format contains exactly one application. All instance(s) of the contained application **MUST** be created in its own unique owner context protected by the firewall.
- The extension library JAR file
 - This is a standard library JAR format containing Java class files. Extension library classes are accessible to all applications on the card. Instances of classes instantiated from the extension library are placed in the owner context of the application which creates the instance.
- The classic library JAR file
 - This is a standard JAR library format containing Java class files. Classic library classes are only accessible to the classic applications on the card. Instances of classes instantiated from the classic library are placed in the owner context of the classic application which creates the instance.

The card manager, when operating in a public network domain, **MUST** reject distribution and deployment unit formats other than the three defined above.

Note – The *Runtime Environment Specification, Java Card Platform, v3.0.1, Connected Edition* does not currently define a distribution unit for encapsulating an application group, meaning more than one application module, or for encapsulating a group library, meaning classes shared by the application modules.

8.4 Distribution Formats

The section describes distribution and deployment unit formats, as well as the descriptors, including the web application deployment descriptor, the applet application deployment descriptor, the Java Card Platform-specific application descriptor, and the runtime descriptor.

The assertions which use the MUST directive MUST be enforced by any validating tool, as well the card manager, when processing the application and library distribution formats described in this chapter. Violations MUST result in an error being flagged by the tool or the rejection of the application or library by the card manager.

8.4.1 Application Module Distribution Format

The application module distribution format is a Java Archive (JAR) format described in the Java Platform, Standard Edition JAR file specification and is used to encapsulate a single application that can be deployed on a Java Card platform. The format used by each of the application types follows a common directory structure template. The common directory structure includes:

- Runtime descriptor information in the standard JAR meta-data file
META-INF/MANIFEST.MF.
- The optional Java Card Platform-specific application descriptor file
META-INF/javacard.xml
- An application model specific directory WEB-INF for a web application or
APPLET-INF for an applet application.

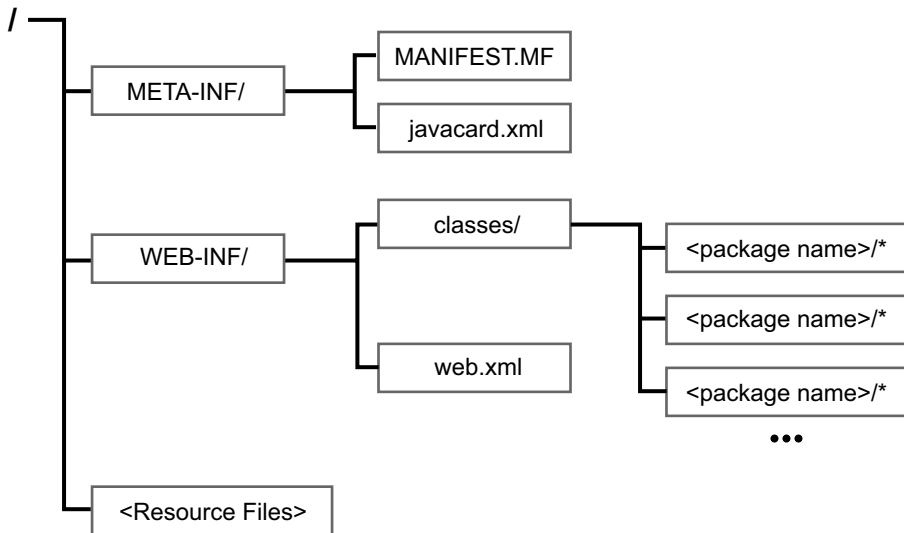
The following sections describe the individual variants for the web application model type, the extended application model type and the classic application model type respectively.

8.4.1.1 Web Application Module Distribution Format

FIGURE 8-1 shows the directory structure of the web application module distribution format. The structure MUST be that of the web archive (.war) file with the following differences:

- No support for application private library directory WEB-INF/lib
- An additional Java Card Platform-specific application descriptor file
javacard.xml is supported. The format of this descriptor is specified in
[Section 8.5.3, “Java Card Platform-specific Application Descriptor”](#) on page 8-11.

FIGURE 8-1 Web Application Module Distribution Format



The `WEB-INF` directory at the root of the JAR directory structure contains the `classes` directory for the web application module and the web application deployment descriptor file (`web.xml`). Each Java class file (`.class`) MUST be in a directory with a relative directory path that matches the package name for that class within the `classes` directory, to be successfully located by the class loader. Additional resource files may also be present along with the class files. The format of this web application deployment descriptor is specified in [Section 8.5.4, “Web Application Deployment Descriptor”](#) on page 8-15.

The `META-INF/MANIFEST.MF` file contains standard JAR file meta data information. The Java Card Platform defines additional runtime descriptor information in the `META-INF/MANIFEST.MF` file. The format of this descriptor is specified in [Section 8.5.6, “The Runtime Descriptor”](#) on page 8-17. The `META-INF` directory also contains the Java Card Platform-specific application descriptor file `javacard.xml`.

Additional, implementation specific meta data information files, MAY be present only in the `META-INF` and `WEB-INF` directories within the JAR file. The card management facility MUST silently ignore the files that it does not recognize.

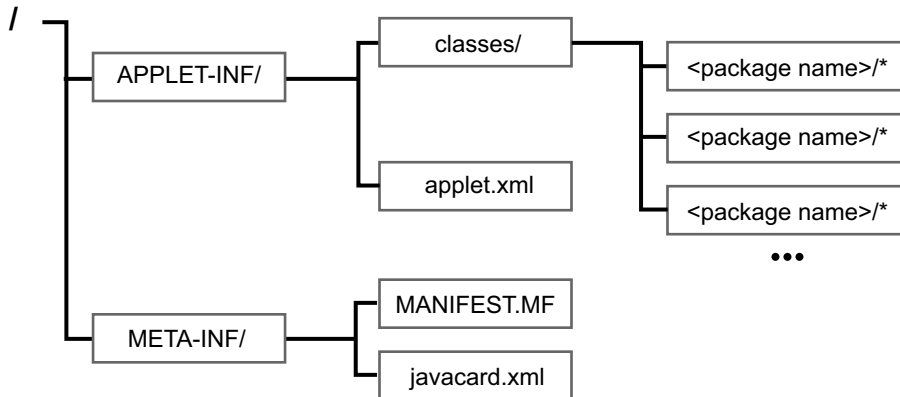
All other files, outside the `META-INF` and `WEB-INF` directory hierarchies, included in the JAR file are not directly processed by the Java Card platform and simply stored as application resource files. These resource files are static web resources serviced by the web container to off-card clients. The various elements of a web application are described in Chapter 9 of *Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition*.

The recommended file name extension for the web application module distribution format file is `war`.

8.4.1.2 Extended Applet Application Module Distribution Format

FIGURE 8-2 shows the directory structure of the extended applet application module distribution format.

FIGURE 8-2 Extended Applet Application Module Distribution Format



The `APPLET-INF` directory at the root of the JAR directory structure contains the `classes` directory for the extended applet application module and the applet application deployment descriptor file (`applet.xml`). Each Java class file (`.class`) MUST be in a directory with a relative directory path that matches the package name for that class within the `classes` directory, to be successfully located by the class loader. Additional resource files may also be present along with the class files. The format of this applet application deployment descriptor is specified in [Section 8.5.5, “Applet Application Deployment Descriptor” on page 8-15](#).

The `META-INF/MANIFEST.MF` file contains standard JAR file meta data information. The Java Card Platform defines additional runtime descriptor information in the `META-INF/MANIFEST.MF` file. The format of this descriptor is specified in [Section 8.5.6, “The Runtime Descriptor” on page 8-17](#). The `META-INF` directory also contains the Java Card Platform-specific application descriptor file `javacard.xml`.

Additional, implementation specific meta data information files, MAY be present only in the `META-INF` and `APPLET-INF` directories within the JAR file. The card management facility MUST silently ignore the files that it does not recognize.

The card management facility MUST silently ignore all other files, outside the `META-INF` and `APPLET-INF` directory hierarchies, that are included in the JAR file.

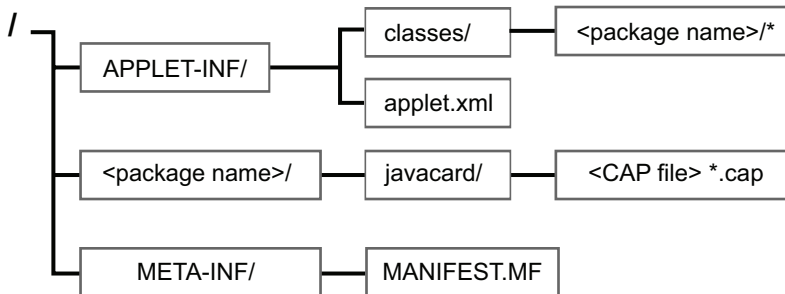
The recommended file name extension for the extended applet application module distribution format file is `eap`.

8.4.1.3 Classic Applet Application Module Distribution Format

FIGURE 8-3 shows the directory structure of the classic applet application module distribution format. The structure is similar to that of the extended applet application module (Section 8.4.1.2, “Extended Applet Application Module Distribution Format” on page 8-6) with the following differences:

- The `classes` directory contains only one package and optionally a subpackage named `proxy` containing SIO proxy classes, see Section 4.4.2, “SIO Synchronization Proxy Classes” on page 4-17.
- The Classic Edition’s CAP file components, `*.cap`, are included in the JAR file.

FIGURE 8-3 Classic Applet Application Module Distribution Format



The `APPLET-INF` directory at the root of the JAR directory structure contains the `classes` directory for the classic application module and the applet application deployment descriptor file (`applet.xml`). Each Java class file (`.class`) MUST be in a directory with a relative directory path that matches the package name for that class within the `classes` directory, to be successfully located by the class loader. The class files MUST belong to a single named package.

The `META-INF/MANIFEST.MF` file contains standard JAR file meta data information. The Java Card Platform defines additional runtime descriptor information in the `META-INF/MANIFEST.MF` file. The format of this descriptor is specified in Section 8.5.6, “The Runtime Descriptor” on page 8-17. The `META-INF/` directory also contains the Java Card Platform-specific application descriptor file `javacard.xml`.

The CAP file components (`*.cap`) MUST be present in a directory named `javacard` that is in a subdirectory representing the application package directory as described in *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*. The

format of the CAP file components are described in *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*. The presence of the CAP file components and their structural and semantic correctness **MUST** be verified by off-card tools. Validation by the card management facility is not required.

Additional, implementation specific meta data information files, **MAY** be present only in the `META-INF` and `APPLET-INF` directories within the JAR file. The card management facility **MUST** silently ignore the files that it does not recognize.

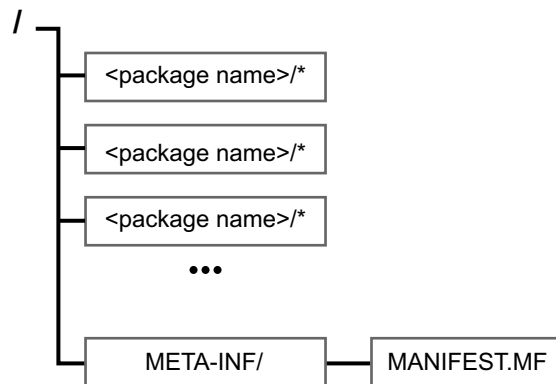
The card management facility **MUST** silently ignore all other files, outside the `META-INF` and `APPLET-INF` directory hierarchies, that are included in the JAR file.

The recommended file name extension for the classic applet application module distribution format file is `cap`.

8.4.2 Extension Library Distribution Format

The extension library distribution format uses the Java Platform Standard Edition library JAR file structure. [FIGURE 8-4](#) shows the format of a Java Platform Standard Edition library JAR file format.

FIGURE 8-4 Java Platform Standard Edition Library JAR Format



Each Java class file (`.class`) **MUST** be in a directory with a relative directory path starting at the root of the library structure that matches the package name for that class, to be successfully located by the class loader.

The `META-INF/MANIFEST.MF` file at the root level of the extension library distribution format contains standard JAR file meta data information defined in the Java Platform Standard Edition JAR specification. Java Card Platform does not define any additional information.

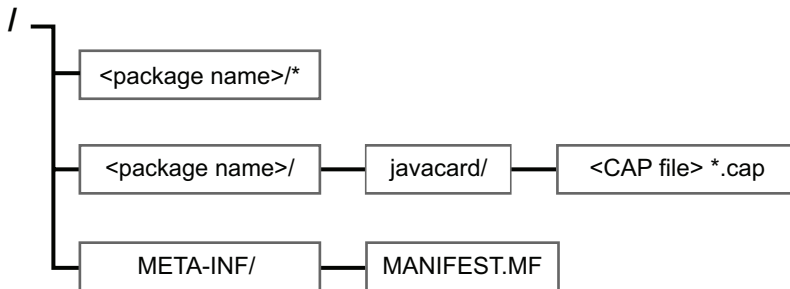
The recommended file name extension for the extension library distribution format file is `jar`.

8.4.3 Classic Library Distribution Format

FIGURE 8-5 shows the format of a classic library distribution format. The classic library distribution format uses the Java Platform Standard Edition library JAR file format (see FIGURE 8-4) with the following restrictions and additions:

- It contains only one package and, optionally, a subpackage `proxy` containing SIO proxy classes, see [Section 4.4.2, “SIO Synchronization Proxy Classes”](#) on page 4-17.
- It includes the classic CAP file components, `*.cap`, in a directory named `javacard` that is in a subdirectory representing the library package directory as described in *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*. The format of the CAP file components are described in *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*.

FIGURE 8-5 Classic Library Distribution Format



The `META-INF/MANIFEST.MF` file at the root level of the classic library distribution format contains standard JAR file meta data information defined in the Java Platform Standard Edition JAR specification. Java Card Platform does not define any additional information. The `Sealed` attribute SHOULD be set as `true` (by off-card tools) to ensure backward compatibility with the classic platform.

Files other than the `<package name>` directory and the `META-INF/MANIFEST.MF` file MAY be silently discarded by the card management facility.

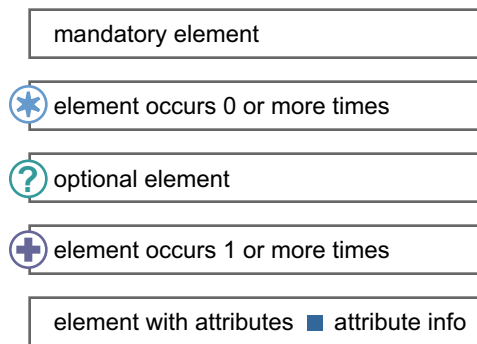
The recommended file name extension for the classic library distribution format file is `cap`.

8.5 Descriptor Formats

8.5.1 Conventions Used in XML Descriptor Element Diagrams

All diagrams in this section that illustrate XML format structures follow the convention displayed in [FIGURE 8-6](#). Note that elements with attributes may also use the occurrence notation using a circle. The sequential ordering requirement of sub-elements within an element are shown using a red arrow. See the individual descriptor schema for more detailed information on the exact syntax.

FIGURE 8-6 Conventions Used in Diagrams of Elements



8.5.2 Common Rules for Processing the XML Descriptors

This section lists some general rules that the card management facility **MUST** implement and that the developer must account for concerning the processing of all the XML descriptors within the Java Card application distribution unit.

- The card management facility **MUST** support XML descriptors in canonical XML format without comments as specified in the W3C Recommendation document *Canonical XML* at <http://www.w3.org/TR/xml-c14n>.
- The card management facility **MUST** support XML descriptors that are valid against the associated schema.

- Syntactic validation by the card management facility is not required. However, required elements and attributes **MUST** be present. Optional elements and attributes **MAY** be omitted.
- Semantic validation (such as proper references to class names, authenticator URI, consistency amongst descriptors etc.) **MUST** be performed by the card management facility. Semantic errors **MUST** result in the rejection of the application.
- Assertions which use the **MUST** directive in the format descriptions **MUST** be enforced by the card management facility. Violations **MUST** result in the rejection of the application.

8.5.3 Java Card Platform-specific Application Descriptor

The Java Card Platform-specific application descriptors convey Java Card Platform-specific elements and configuration information of an application between application developers, application assemblers, and deployers. The Java Card Platform-specific application descriptor is an optional file named `META-INF/javacard.xml` which **MAY** be present in the application module JAR file. If the file is absent, all elements within the `javacard-app` root element of the descriptor are assumed to be absent.

The Java Card Platform-specific application descriptor structure, content and semantics are defined in an XML schema document.

8.5.3.1 Java Card Platform-specific Application Descriptor Elements

The following types of configuration and deployment information **MUST** be supported in the Java Card Platform-specific application descriptor:

- Authentication information for remote web application access
- List of dynamically loaded classes
- List of shareable interface classes
- List of user and on-card client security roles

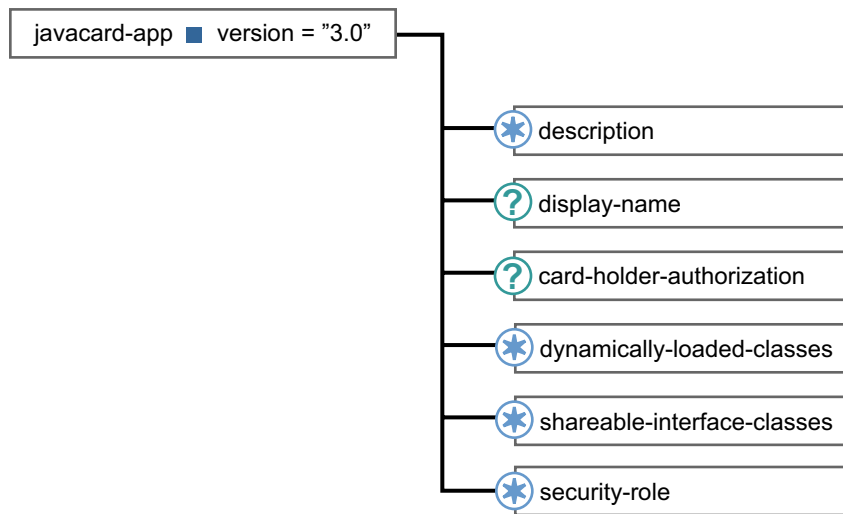
8.5.3.2 Java Card Platform-specific Application Descriptor Element Structure

This section illustrates the elements in a Java Card Platform-specific application descriptor. The notation convention used here is described previously in [Section 8.5.1, “Conventions Used in XML Descriptor Element Diagrams”](#) on page 8-10.

`javacard-app` Element

The `javacard-app` element is the root element of the Java Card Platform-specific application descriptor. This element has a required attribute `version` to specify to which version of the schema the Java Card Platform-specific application descriptor conforms. The attribute value **MUST** be “3.0”. [FIGURE 8-7](#) shows the structure of the `javacard-app` element. Each sub-element is described in the next sections.

FIGURE 8-7 `javacard-app` Element Structure



`description` Element

The `description` element is used to provide text describing the parent element. This element occurs not only under the `javacard-app` element, but also under the elements `dynamically-loaded-classes`, `shareable-interface-classes` and `security-role`. It has an optional attribute `xml:lang` to indicate which language is used in the description. The default value of this attribute is English (“en”).

display-name *Element*

The `display-name` element contains a short name that is intended to be displayed by tools. The display name need not be unique. This element has an optional attribute `xml:lang` to specify the language.

card-holder-authorization *Element*

The `card-holder-authorization` element is used by a remotely accessible application to configure the card holder authentication requirements to authorize access for a remote client. A `role-name` listed here **MUST** be a `role-name` listed in the `security-role` element with a `USER` category attribute. [FIGURE 8-8](#) shows the structure of the `card-holder-authorization` element. For more detailed information on authorization of remotely accessible applications see [Section 6.4.6, “Card Holder Authorization For Remotely Accessible Applications”](#) on page 6-40.

FIGURE 8-8 `card-holder-authorization` Element Structure

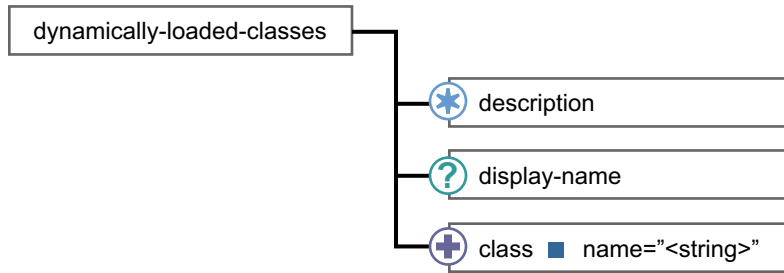


dynamically-loaded-classes *Element*

The `dynamically-loaded-classes` element is used to declare the classes that the application loads dynamically using the `Class.forName` API. All classes within the application distribution unit that are dynamically loaded **SHOULD** be declared in the `name` attribute of `class` sub-elements. An attempt to dynamically load a class from within the application module that is not declared, and which has not already been loaded, **MUST** result in a `SecurityException` being thrown. The attribute value **MUST** be the fully qualified classname of the dynamically loaded class. Classes in extension library packages that are dynamically loaded **SHOULD** also be declared here. [FIGURE 8-9](#) shows the structure of the `dynamically-loaded-classes` element.

Note that all Classic SIO synchronization proxies (see [Section 4.4.2, “SIO Synchronization Proxy Classes”](#) on page 4-17) and application defined subclasses of the `java.util.ResourceBundle` class (see which are dynamically loaded by the Java Card RE on behalf of the application **SHOULD** also be declared in this element.

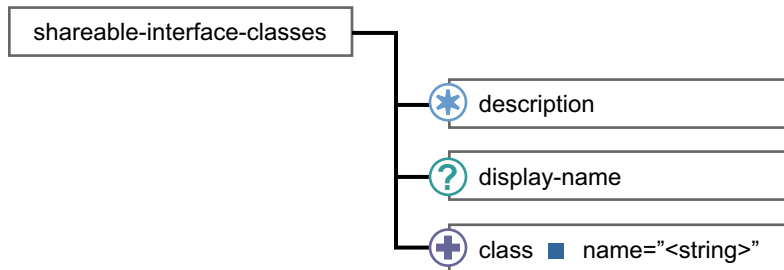
FIGURE 8-9 dynamically-loaded-classes Element Structure



shareable-interface-classes *Element*

The `shareable-interface-classes` element is used to declare the shareable interface classes of the application. All public shareable interface classes within the application distribution unit that are accessible to applications in other group contexts on the card SHOULD be declared in the `name` attribute of `class` sub-elements. The attribute value MUST be the fully qualified class name of the shareable interface class. A shareable interface class not listed here, or not declared with the `public` class modifier, MUST NOT be loaded by the shareable interface class loader. Note that the same interface may already have been loaded by the shareable interface class loader while loading another application - a name conflict MUST NOT be flagged. [FIGURE 8-10](#) shows the structure of the `shareable-interface-classes` element.

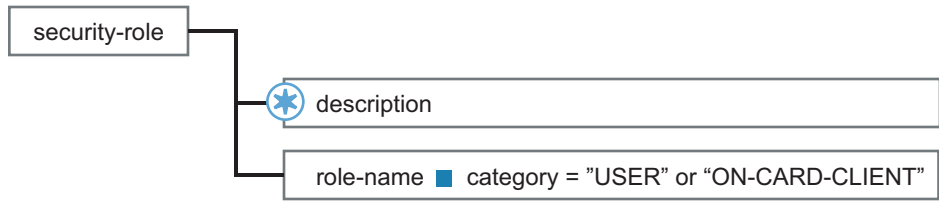
FIGURE 8-10 shareable-interface-classes Element Structure



security-role *Element*

The `security-role` element defines a security role. The sub-element `role-name` designates the name of the security role. The `role-name` element has a required attribute category to specify the type of role, either "USER" OR "ON-CARD-CLIENT". [FIGURE 8-11](#) shows the structure of the `security-role` element.

FIGURE 8-11 security-role Element Structure



8.5.4 Web Application Deployment Descriptor

The web application deployment descriptor conveys web application model elements and configuration information of an application between application developers, application assemblers, and deployers. The web application deployment descriptor is a mandatory file named `WEB-INF/web.xml` which **MUST** be present for each web application module encapsulated within the distribution format JAR file.

The format and description of the web application deployment descriptor is detailed in the deployment descriptor chapter of *Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition*.

[Section 8.5.2, “Common Rules for Processing the XML Descriptors” on page 8-10](#) lists some additional rules, beyond those listed in the *Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition*, that the card management facility **MUST** implement and that the developer must account for concerning the processing of the web application deployment descriptor of the Java Card application.

8.5.5 Applet Application Deployment Descriptor

The applet application deployment descriptor conveys APDU-based application model elements and configuration information of an application between application developers, application assemblers, and deployers. The applet application deployment descriptor is a mandatory file named `APPLET-INF/applet.xml`, which **MUST** be present in the applet application module JAR file.

The applet application deployment descriptor structure, content and semantics are defined in an XML schema document.

8.5.5.1 Applet Application Deployment Descriptor Element

The following type of configuration and deployment information **MUST** be supported in the applet application deployment descriptor:

- List of applet classes and their Application Identifiers (AID)

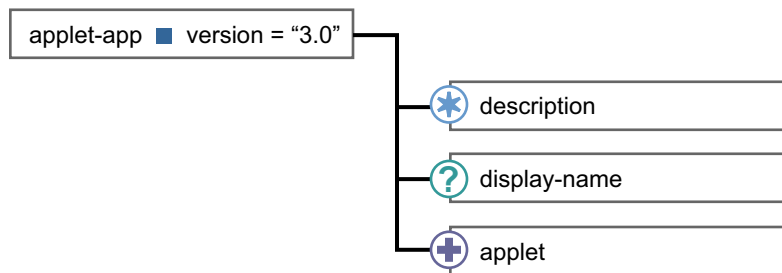
8.5.5.2 Applet Application Deployment Descriptor Element Structure

This section illustrates the elements in an applet application deployment descriptor. The notation convention used here is described in [Section 8.5.1, “Conventions Used in XML Descriptor Element Diagrams”](#) on page 8-10.

applet-app Element

The `applet-app` element is the root element of the applet application deployment descriptor. This element has a required attribute `version` to specify to which version of the schema the Java Card Platform-specific application descriptor conforms. The attribute value **MUST** be “3.0”. [FIGURE 8-12](#) shows the structure of the `applet-app` element. Each sub-element is described in the next sections.

FIGURE 8-12 `applet-app` Element Structure



description Element

The `description` element is used to provide text describing the parent element. This element occurs not only under the `applet-app` element, but also under other multiple elements. It has an optional attribute `xml:lang` to indicate which language is used in the description. The default value of this attribute is English (“en”).

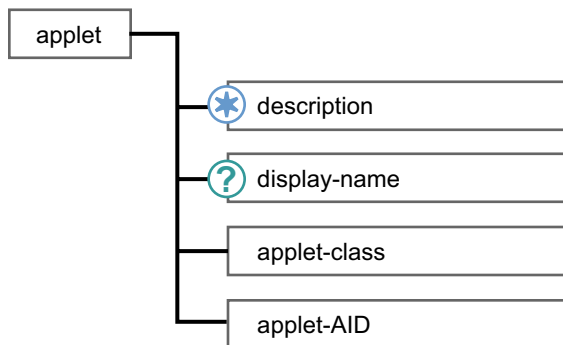
display-name *Element*

The `display-name` element contains a short name that is intended to be displayed by tools. The display name need not be unique. This element has an optional attribute `xml:lang` to specify the language.

applet *Element*

The `applet` element is used to declare the applet classes of the application. All concrete subclasses of `javacard.framework.Applet` within the application distribution unit that can be instantiated **SHOULD** be declared in the `applet-class` element with its corresponding Application Identifier in the `applet-AID` element. The `applet-class` element **MUST** be the fully qualified classname of the applet class. The `applet-AID` element uses the applet application URI format notation described in [Section 2.3.1, “Unified Naming Scheme” on page 2-5](#). An applet not listed here **MUST NOT** be instantiated by the card management facility. [FIGURE 8-13](#) shows the structure of the `applet` element.

FIGURE 8-13 applet Element Structure



8.5.6 The Runtime Descriptor

The runtime descriptors convey runtime elements and configuration information of an application between application developers, application assemblers, and deployers.

The runtime descriptor information is not stored as a separate file within the distribution format JAR file structure. Instead, the runtime descriptor information is included in the metadata file, the `META-INF/MANIFEST.MF` file, at the top level of the directory structure of the application module distribution format.

8.5.6.1 Runtime Descriptor Elements

The runtime descriptor includes information about the distribution format components, and runtime configuration and environment information. The information in the runtime descriptor is provided by the application provider and includes policy attributes relating to security and external access rules for the application.

The runtime descriptor information includes:

- Application Type
- Mapping of security roles to user credentials and on-card client credentials
- Visibility control on shareable interface classes
- Web application runtime specifics
 - Context Path
 - Secure port number
 - Secure session requirement
 - Client authentication requirement during secure sessions
- Classic applet application runtime specifics
 - Package AID (of single module)

8.5.6.2 Rules for Processing the Runtime Descriptor

This section lists some general rules that the card management facility **MUST** implement and that the developer must account for concerning the processing of the runtime descriptor within the Java Card application distribution unit.

- The card management facility **MUST** support runtime descriptors that conform to the requirements specified in [Section 8.5.6.3, “Runtime Descriptor Element Structure” on page 8-19](#). Validation by the card management facility is not required, but required attributes **MUST** be present. Optional attributes **MAY** be omitted.
- In elements whose value is an enumerated type, the value is case sensitive.
- Semantic validation (such as proper references to class names, authenticator URI, consistency amongst descriptors etc.) **MUST** be performed by the card management facility. Semantic errors **MUST** result in the rejection of the application.
- Assertions which use the **MUST** directive in the format descriptions **MUST** be enforced by the card management facility. Violations **MUST** result in the rejection of the application.

8.5.6.3 Runtime Descriptor Element Structure

The runtime descriptor information is transcribed using the name/value pair format for each runtime attribute. The runtime descriptor information for an application module only defines attributes in the main manifest file section.

The syntax used for the runtime descriptor information in the application module is shown below in [Section FIGURE 8-14, “Runtime Descriptor Structure” on page 8-20](#) using a notation similar to that used for regular expressions conforming to the manifest file syntax conventions. The manifest file syntax specifications are applicable to the runtime descriptor information also, such as the use of the `SPACE` character at the start of a new line to indicate the continuation of the (previous) attribute value and the use of empty lines between manifest file sections. Manifest file attribute names are case-insensitive.

The metadata file(s), `META-INF/MANIFEST.MF`, may contain additional attributes beyond those described here as runtime descriptor information. These additional attributes declared in the `META-INF/MANIFEST.MF` file are case-insensitive application properties and may be retrieved by the application at runtime via the API `javacardx.framework.JCSystem.getAppProperty()` by providing the attribute name as the parameter. JAR file specification rules apply when attributes are duplicated. The application provider may define manifest file attributes to assign values to application properties defined by the application programmer for configuring the application code.

In addition, these standard manifest file attributes **MUST** be supported by the card manager:

- The `Sealed` attribute(s) defined for package sealing information in the `META-INF/MANIFEST.MF` file of the application module. This attribute is defined in the JAR file specification and is applicable within the application module's class loader delegation hierarchy. Any package declared as sealed in the application module **MUST NOT** be present in the class loader hierarchy. Conversely, any package declared as sealed in the class loader hierarchy **MUST NOT** be present in the application module.

The following attributes in the main section are defined as runtime descriptor information for the application module. Attributes not specified in this list **MUST** be treated as application defined items. Attributes may appear in any order. Attribute names are case insensitive.

FIGURE 8-14 Runtime Descriptor Structure

```

Runtime-Descriptor-Version: 3.0
Application-Type: <web | extended-applet | classic-applet>
[ Classic-Package-AID: <package AID> ] ?
[ Web-Context-Path: <web-app-default-context-path> ] ?
[ Web-Secure-Port-Number: <port #> ] ?
[ Web-Client-Auth-Required: <true | false> ] ?
[ Web-Secure-Access-Only: <true | false> ] ?
{ On-Card-Client-Role-List: <Client-role> [ , <Client-role> ] * } ?
{
    { <Client-role>-Mapped-To-Client-URI: <client-uri-pattern>
      [ , <client-uri-pattern> ] * } |
    { <Client-role>-Mapped-To-Domain-Name: <protection-domain>
      [ , <protection-domain> ] * } |
    { <Client-role>-Mapped-To-Auth-Credential: <credential-alias>
      [ , <credential-alias> ] * }
} *
{ On-Card-Clients-Credential-Auth-Type: <client-only> } ?
{ On-Card-Clients-Credential-Auth-Duration: <access |
    card-session | client-lifetime> } ?
{ <Client-role>-Credential-Auth-Type: <client-only> } *
{ <Client-role>-Credential-Auth-Duration: <access |
    card-session | client-lifetime> } *
{ User-Role-List: <User-role> [ , <User-role> ] * } ?
{ <User-role>-Mapped-To-Auth-URI: <Authenticator-URI>
    [ , <Authenticator-URI> ] * } *

```

The items shown in boldface above are reserved keyword names representing runtime properties. The : character separates the attribute from its assigned value. The <..> notation is used to describe the contents of the attribute value. The (..),{..} and [..] notation with the quantification suffix uses the standard regular expression notation with ? for 0 or 1, + for 1 or more, and * for 0 or more.

Each runtime descriptor attribute is described in detail in the following sections.

Runtime-Descriptor-Version *Attribute*

The Runtime-Descriptor-Version attribute **MUST** be present and marks the syntax described here. The value **MUST** be 3.0.

Application-Type *Attribute*

The Application-Type attribute MUST be present and its value indicates the type of the application model implemented by the module. The possible values are web, extended-applet or classic-applet.

Classic-Package-AID *Attribute*

The Classic-Package-AID attribute SHOULD be present for a classic applet application module. It MUST NOT be present for an extended applet application module or a web application module. The value of this attribute MUST be the classic application package AID URI. The AID URI is prefixed with //aid to denote the AID registry-based name space authority. The actual AID is represented as a string using a series of hexadecimal digits in uppercase, 2 per byte, concatenated together. The RID and PIX fields are separated by a slash. For example, the Java Card Platform RI sample JavaPurse package is represented by the package AID URI //aid/A000000062/03010C02. The Classic-Package-AID attribute uses the applet application URI format notation described in [Section 2.3.1, “Unified Naming Scheme” on page 2-5](#).

Web-Context-Path *Attribute*

The Web-Context-Path attribute MUST be present for a web application module. It MUST NOT be present for an extended applet application module or a classic applet application module. It contains the default Context path to be used by the web container for this web application. The web application URI is assigned this Context path by default if an alternate is not specified during instantiation.

The Web-Context-Path attribute value uses the application URI naming format as described in [Section 2.3.1, “Unified Naming Scheme” on page 2-5](#).

Web-Secure-Port-Number *Attribute*

This optional Web-Secure-Port-Number attribute MAY be present for a web application module. It MUST NOT be present for an extended applet application module or a classic applet application module. It contains the static port number that the web application module wishes to use during secure HTTP communication. If the static port number is already in use, the instantiation of the web application MUST fail. If this attribute is not specified, a port number is dynamically assigned to the web application module, for use during secure web container managed HTTP communication.

Web-Client-Auth-Required *Attribute*

This optional Web-Client-Auth-Required attribute MAY be present for a web application module. It MUST NOT be present for an extended applet application module or a classic applet application module. It contains a boolean value keyword - true or false. If specified as true, the web container MUST perform Client authentication during the SSL/TLS handshake for an HTTPS session to access secure application resources. If this attribute is omitted, or specified as false, the default card setting for HTTPS sessions is used.

Web-Secure-Access-Only *Attribute*

This optional Web-Secure-Access-Only attribute MAY be present for a web application module. It MUST NOT be present for an extended applet application module or a classic applet application module. It contains a boolean value keyword - true or false. If specified as true, the web container MUST allow access to the application's resources only during an HTTPS session. If this attribute is omitted, or specified as false, the web container MUST allow access over plain HTTP to the application's resources that are not protected by security constraints with transport guarantee requirements as declared in the web application deployment descriptor of the application.

On-Card-Client-Role-List *Attribute*

The On-Card-Client-Role-List attribute is used to list all the on-card client roles. An on-card client MAY be mapped to client URIs by using the <Client-Role>-Mapped-To-Client-URI attribute or to a protection domain by using the <Client-Role>-Mapped-To-Domain-Name attribute or to an authentication credential by using the <Client-Role>-Mapped-To-Auth-Credential attribute that follows. An on-card client role MUST not be mapped more than once. A <Client-role> token listed here MUST be the On-Card-Client role name defined in the Java Card Platform-specific application descriptor element security-role/role-name with an ON-CARD-CLIENT category attribute.

Since manifest file attribute names are case-insensitive, on-card client role names which differ in case only cannot be uniquely mapped and MUST therefore result in the application being rejected.

<Client-Role>-Mapped-To-Client-URI *Attribute*

The optional <Client-Role>-Mapped-To-Client-URI attribute allows an application provider to specify the mapping of on-card client security roles to actual client application URIs. The <Client-role> token used here MUST be one of the On-Card-Client roles listed for the On-Card-Client-Role-List attribute

above. The attribute name is a concatenation of the *<Client-role>* token and the reserved keyword *Mapped-To-Client-URI*. Multiple on-card clients may be designated by using the path-prefix URI pattern notation.

<Client-Role>-Mapped-To-Domain-Name Attribute

The optional *<Client-Role>-Mapped-To-Domain-Name* attribute allows an application provider to specify the mapping of on-card client security roles to client applications' protection domain names. The *<Client-role>* token used here MUST be one of the *On-Card-Client* roles listed for the *On-Card-Client-Role-List* attribute above. The attribute name is a concatenation of the *<Client-role>* token and the reserved keyword *-Mapped-To-Domain-Name*. The protection domain name value used here MUST be prefixed with the protection domain URI scheme identifier *pd* and corresponds to the name of a protection domain, for example *pd:Classic*.

<Client-Role>-Mapped-To-Auth-Credential Attribute

The optional *<Client-Role>-Mapped-To-Auth-Credential* attribute allows an application provider to specify the mapping of on-card client security roles to the credential alias names to be used for on-card client authentication. The *<Client-role>* token used here MUST be one of the *On-Card-Client* roles listed for the *On-Card-Client-Role-List* attribute above. The attribute name is a concatenation of the *<Client-role>* token and the reserved keyword *-Mapped-To-Auth-Credential*. The credential alias names used here reference the peer credential alias names defined in its *CredentialManager* object(s). If no corresponding peer credential alias name is defined in the *CredentialManager* object(s), authentication MUST fail.

On-Card-Clients-Credential-Auth-Type Attribute

The optional *On-Card-Clients-Credential-Auth-Type* attribute indicates the type of the on-card client authentication required by the application for its on-card client applications. The only supported value is *client-only*. The default value is *client-only*.

On-Card-Clients-Credential-Auth-Duration Attribute

The optional *On-Card-Clients-Credential-Auth-Duration* attribute indicates the applicable duration of a successful on-card client authentication for its on-card client applications. The possible values are *access*, *card-session* or *client-lifetime*. The default value is *card-session*.

<Client-Role>-Credential-Auth-Type Attribute

The optional *<Client-Role>-Credential-Auth-Type* attributes indicate the type of the on-card client authentication required by the application for this specific *<Client-Role>*. This attribute overrides any application wide authentication type setting in *On-Card-Clients-Credential-Auth-Type* attribute. The *<Client-Role>* name used here MUST correspond to a *<Client-Role>* mapped to credential alias names for on-card client authentication in the *<Client-Role>-Mapped-To-Auth-Credential* attribute. The only supported value is *client-only*.

<Client-Role>-Credential-Auth-Duration Attribute

The optional *<Client-Role>-Credential-Auth-Duration* attribute indicate the applicable duration of a successful on-card client authentication for this specific *<Client-Role>*. This attribute overrides any application wide authentication duration setting in the *On-Card-Clients-Credential-Auth-Duration* attribute. The *<Client-Role>* name used here MUST correspond to a *<Client-Role>* mapped to credential alias names for on-card client authentication in the *<Client-Role>-Mapped-To-Auth-Credential* attribute. The possible values are *access*, *card-session* or *client-lifetime*.

User-Role-List Attribute

The *User-Role-List* attribute is used to list all the user roles that are mapped to authenticator URIs in the *<User-Role>-Mapped-To-Auth-URI* attributes that follow. The *<User-role>* token listed here MUST be the User role name defined in the web application deployment descriptor element *security-role/role-name* or the Java Card Platform-specific application descriptor element *security-role/role-name* with an *USER* category attribute.

Since manifest file attribute names are case-insensitive, user role names which differ in case only cannot be uniquely mapped and MUST therefore result in the application being rejected.

<User-Role>-Mapped-To-Auth-URI Attribute

The optional *<User-Role>-Mapped-To-Auth-URI* attribute allows an application provider to specify the mapping of user security roles to authenticator URIs. The *<User-role>* token used here MUST be one of the User roles listed for the *User-Client-Role-List* attribute above. The attribute name is a concatenation of the *<User-role>* token and the reserved keyword *-Mapped-To-Auth-URI*. Multiple authenticators may be designated by using the path-prefix URI pattern notation.

Authenticators and their URI naming conventions are described in [Section 6.4.1, “Scheme-specific Authenticators” on page 6-30](#). User role to authenticator mapping is described in detail in [Section 6.3.1.3, “Role to User Mapping” on page 6-25](#).

8.6 Loading Application Modules

The card manager application and the card management facility cooperate in the loading of application modules on the card. Application modules ([Section 8.4.1, “Application Module Distribution Format” on page 8-4](#)) are deployment units which encapsulate applications.

The card manager application typically performs the following functions during the loading of an application module:

- Communicates with the off-card client requesting the load operation
- Checks the credentials of the off-card client and authorizes the request
- Determines the type of application being loaded, either web application, extended applet, or classic applet
- Determines the free-form name for the application module being loaded. This name may be used to reference the application on the card
- Receives the contents of the application module and other implementation-specific loading information
- Ensures the authenticity and privacy of the data during the loading
- Instructs the card management facility to load the application module passing it the application module content and application module type information
- Reports progress information to the off-card client
- Upon successful completion, assigns a protection domain ([Section 6.2.2, “Protection Domains” on page 6-13](#)) and a Credential Manager to the group context of the newly loaded application module
- If the loading attempt is unsuccessful:
 - Rolls back all partially completed loading steps such that the loading can be retried by the off-card client after the error condition is rectified

The card management facility is the platform level entity responsible for operating on the application module contents and communicating with the appropriate platform subsystems to perform the deployment of the application module and initialize its security.

Prior to loading the application module, the card management facility **MUST** ensure that the permission `javacardx.spi.cardmgmt.CardManagementPermission` with the target name `deploymentUnit.load` is granted to the card manager application.

The card management facility performs the following functions during the loading of an application module:

- Extracts the classes, descriptors and other resource files of the application module
- Creates a new application module class loader for the application module and a group library class loader, if applicable, within the class loading delegation hierarchy as defined in [Section 6.7, “Code Isolation”](#) on page 6-57
- Loads the classes of the application module and implements the class pre-loading optimization rules defined in [Section 8.6.3, “Class Pre-loading Optimizations”](#) on page 8-28:
 - Loads the classes for the application module
 - Enforces code isolation and class file lookup order requirements described in [Section 8.6.1, “Code Isolation and Class File Lookup Order Requirements”](#) on page 8-27
 - Enforces package access control rules defined in [Section 6.8, “Package Access Control”](#) on page 6-63. Note that the `Sealed` attribute **MUST** be enforced by the shareable interface class loader, and the application module’s class loader for package sealing within its own delegation hierarchy.
- Assigns a new unique group context to the application module that **MUST** be associated with any application (or applet) instance created from within this application module.
- Loads the application module into the appropriate container based on its application model. Implements the application model-specific loading requirements described in [Section 3.2.1, “Application Module Loading”](#) on page 3-3 and [Section 4.2.1, “Application Module Loading”](#) on page 4-3
- Configures the runtime and security requirements of the application module based on the descriptors
- The class-path resources in the application module **MUST** be stored and made available to the application module code using the `Class.getResourceAsStream` method. Note that files with file name extension `.class` **MUST NOT** be accessible via this method.
- Rolls back any partially completed loading steps in case of loading failures
- Provides completion status to the card manager

8.6.1 Code Isolation and Class File Lookup Order Requirements

The classes in each application module are loaded in a namespace which is isolated from classes in other application modules. Applications may communicate with other applications via classes in the shareable interface class loader, extension library class loader, classic library class loader or the bootstrap class loader namespaces. The card management facility **MUST** create a new application module class loader within the class loader delegation hierarchy for loading the application module and follow the code isolation rules and class file lookup order requirements described in [Section 6.7, “Code Isolation” on page 6-57](#).

The application programmer enumerates the exposed shareable interface classes used for inter-communication with other applications in other group contexts via the `shareable-interface-classes` elements of the Java Card Platform-specific application descriptor. These shareable interface classes **MUST** be loaded by the shareable interface class loader as described in [Section 6.7, “Code Isolation” on page 6-57](#). Note that an interface class with an identical class name may already have been loaded by the shareable interface class loader. While loading another application, a name conflict **MUST NOT** be flagged.

When a shareable interface class is “promoted” from the application module class loader to the shareable interface class loader, all super interfaces of the shareable interface class and pseudo array classes of these classes, loaded (or being loaded) by the application module class loader, **MUST** also be automatically “promoted” to the shareable interface class loader. Super interfaces of the shareable interface class being loaded, which have been previously loaded by a different class loader **MUST NOT** be “promoted” to the shareable interface class loader.

8.6.2 Class Dependency Resolution Requirements

Loading and Linking a class into the Runtime Environment of the card requires dependencies on other classes to be resolved. Class dependencies occur when a class directly links with another. A dependent class may be one of the following:

- a system class
- a previously loaded shareable interface class
- a class in an extension or classic library
- a class within the application module itself.

An application module class **MUST** only depend on class in the application module class loader or a class higher up in the class loader delegation hierarchy.

A shareable interface class loaded by the shareable interface class loader **MUST** only depend on a class in the shareable interface class loader or a class higher up in the class loader delegation hierarchy.

A class linking dependency on a class loaded by a class loader lower in the class loader delegation hierarchy **MUST** result in the rejection of the deployment unit.

8.6.3 Class Pre-loading Optimizations

The Java Virtual Machine specification describes the process of loading, linking (including verifying, preparing and resolving) and initializing classes. The Java programming language allows an implementation flexibility as to when the loading and linking activities may take place. The Java Card platform requires the loading and linking activities to principally be performed when the card management facility loads the application module for the following reasons:

- Minimizes linking errors - class verification errors, preparation errors and resolution errors - during the functional operation of the application
- Allows the Java Card platform to optimize the storage space required for the application module

The process of performing the class loading and linking while loading the application module is called pre-loading. Note that pre-loading **MUST NOT** include initializing the classes as described in the Java Virtual Machine specification.

The following class pre-loading rules **MUST** be enforced by the card management facility during the loading of an application module:

- All the root classes of the application module (and the dependent classes) based on the application model it implements **MUST** be pre-loaded:
 - The root classes of a web application are the servlet, filter and listener classes enumerated in the web application deployment descriptor elements `servlet/servlet-class`, `filter/filter-class`, and the `listener/listener-class` respectively
 - The root classes of an applet application are the applet classes enumerated in the applet application deployment descriptor element `applet/applet-class`
- All the dynamically loaded classes of the application module enumerated in the `dynamically-loaded-classes` element of the Java Card Platform-specific application descriptor (and the tree of classes referenced by these classes) **MUST** be pre-loaded
- All the exposed shareable interface classes of the application module (and the super interface classes) enumerated in the `shareable-interface-classes` element of the Java Card Platform-specific application descriptor **MUST** be pre-loaded by the shareable interface class loader. In addition, pseudo classes, if

any, representing arrays of those classes being pre-loaded by the shareable interface class loader MUST also be pre-loaded by the shareable interface class loader.

- Errors during loading and linking to resolve dependencies, including those described in [Section 8.6.2, “Class Dependency Resolution Requirements” on page 8-27](#), during pre-loading which result in errors associated with the Error classes - `ClassFormatError`, `UnsupportedClassVersionError`, `ClassCircularityError`, `NoClassDefFoundError`, `VerifyError`, `IncompatibleClassChangeError`, `IllegalAccessError`, `InstantiationError`, `NoSuchFieldError`, `NoSuchMethodError`, `AbstractMethodError` - MUST result in an application module loading failure.
- Classes from the application module which are not pre-loaded according to the class pre-loading rules described in the previous steps MAY be silently discarded. Errors during pre-loading flagged against these unreachable classes MAY also result in an application module loading failure. Linking or structural errors in an unreachable constant pool entry of a class MAY also result in an application module loading failure.

8.7 Loading Libraries

The card manager application and the card management facility cooperate in the loading of libraries on the card. Extension libraries ([Section 8.4.2, “Extension Library Distribution Format” on page 8-8](#)) and classic libraries ([Section 8.4.3, “Classic Library Distribution Format” on page 8-9](#)) are deployment units which encapsulate libraries.

The card manager application typically performs the following functions during the loading of a library:

- Communicates with the off-card client requesting the loading operation
- Checks the credentials of the off-card client and authorizes the request
- Determines the type of library being loaded, extended library or classic library
- Determines a free-form name for the library being loaded. This name may be used to reference the library on the card
- Receives the contents of the library and other implementation specific loading information
- Ensures the authenticity and privacy of the data during the loading
- Instructs the card management facility to load the library, passing it the library content and library type information
- Reports progress information to the off-card client
- If the loading attempt is unsuccessful

- Rolls back all partially completed loading steps such that the loading can be retried by the off-card client after the error condition is rectified

The card management facility is the platform level entity responsible for operating on the library contents and communicating with the appropriate platform subsystems to perform the deployment of the library.

Prior to loading the library, the card management facility **MUST** ensure that the permission `javacardx.spi.cardmgmt.CardManagementPermission` with the target name `deploymentUnit.load` is granted to the card manager application.

The card management facility performs the following functions during the loading of a library:

- Extracts the classes of the library
- Loads the classes for the library
 - Enforces code isolation rules for library classes described in [Section 6.7.1, “Class Loader Delegation Hierarchy”](#) on page 6-58
 - Extension library classes, except shareable interface classes, are loaded by the extension library class loader. shareable interface classes (and their super interfaces and their respective pseudo array classes) are loaded by the shareable interface class loader.
 - Classic library classes, except shareable interface classes, are loaded by the classic library class loader. Shareable interface classes (and their super interfaces and their respective pseudo array classes) are loaded by the shareable interface class loader
 - Enforces package access control rules defined in [Section 6.8, “Package Access Control”](#) on page 6-63. Note that the `Sealed` attribute **MUST** be enforced by the shareable interface class loader, extension library class loader, and the classic library class loader, respectively, for package sealing within its own delegation hierarchy
 - Pre-loads all classes contained in the library, in a manner similar to that described for the application module in [Section 8.6.3, “Class Pre-loading Optimizations”](#) on page 8-28. Note that if a class from the library has been previously loaded, the library being loaded **MUST** be deemed as having a “library unloading dependency” on the library containing the previously loaded class. Similarly, if a shareable interface class from the library has been previously loaded, the library being loaded **MUST** be deemed as having a “library unloading dependency” on the library or the application module which loaded the shareable interface class
 - All shareable interface classes (and their super interfaces and their respective pseudo array classes) from within the library **MUST** be pre-loaded by the shareable interface class loader, in contrast to the application module where only the explicitly declared ones (and their super interfaces and their respective pseudo array classes) are pre-loaded by the shareable interface class loader

- Pre-loading is the process of loading, linking (including verifying, preparing and resolving) classes

A class loaded by the extension library class loader MUST only depend on a class in the extension library class loader or a class higher up in the class loader delegation hierarchy.

A shareable interface class loaded by the shareable interface class loader MUST only depend on a class in the shareable interface class loader or a class higher up in the class loader delegation hierarchy.

- Pre-loading MUST NOT include initializing the classes as described in the Java Virtual Machine specification
- Errors during loading and linking to resolve dependencies during pre-loading which result in errors associated with the Error classes -- `ClassFormatError`, `UnsupportedClassVersionError`, `ClassCircularityError`, `NoClassDefFoundError`, `VerifyError`, `IncompatibleClassChangeError`, `IllegalAccessError`, `InstantiationError`, `NoSuchFieldError`, `NoSuchMethodError`, `AbstractMethodError` -- MUST result in a library loading failure
- The class-path resources in the extension library MUST be stored and made available to the extension library code using the `Class.getResourceAsStream` method. The implicit class-path used to search for the resource in the extension libraries MUST be based on the order of loading - starting with the first library loaded on the card. Note that files with file name extension `.class` MUST NOT be accessible via this method.
- Rolls back any partially completed loading steps in case of loading failures
- Provides completion status to the card manager

8.8 Creation of Application Instances

The card manager application and the card management facility cooperate to create instances of previously loaded application modules.

Note – In this section, the term *application instance* is used to refer to both a web application instance as well as an applet instance.

The card manager application typically performs the following functions while creating an application instance of a previously loaded application module:

- Communicates with the off-card client requesting the application instance creation operation
- Checks the credentials of the off-card client and authorizes the request

- Determines the free-form name of the application module
- Determines the declared web application context path URI or applet class AID URI of the application to be created
- Determines the requested application instance URI to be used for the new application instance respectively
- Determines application instance initialization data and other implementation specific application instance creation information
- Ensures the authenticity and privacy of the data during the application instance creation operation
- Instructs the card management facility to create an instance of the previously loaded application module passing the declared web application context path URI or applet class AID URI, the requested application instance URI, and initialization data
- Reports progress information to the off-card client
- If the loading attempt is unsuccessful
 - Rolls back all partially completed application instance creation steps such that the application instance creation attempt can be retried by the off-card client after the error condition is rectified

The card management facility is the platform level entity responsible for creating application instances and communicating with the appropriate platform subsystems to manage the instance lifecycle, and their initialization and security requirements.

Prior to creating an application instance, the card management facility **MUST** ensure that the permission `javacardx.spi.cardmgmt.CardManagementPermission` with the target name `application.create` is granted to the card manager application.

The card management facility **MUST** perform the following validation steps and other functions during the creation of an application instance:

- The application module **MUST** have been previously loaded
- Assigns a unique application context for the new application instance, within the application module's group context. All objects created by the new application instance **MUST** be owned by the application context
- For a web application:
 - The declared web application URI **MUST** be the declared `Web-Context-Path` attribute in the runtime descriptor
 - An instance of the web application module **MUST NOT** be currently registered on the card
 - The requested web application instance URI **MUST NOT** be currently registered on the card

- Implements application model-specific instance creation requirements described in [Section 3.2.3, “Application Instance Creation”](#) on page 3-5
- For an applet application:
 - The declared applet class URI MUST be the applet AID of an applet declared in the `applet/applet-AID` element of the applet application deployment descriptor
 - Implements application model specific instance creation requirements described in [Section 4.2.3, “Application Instance Creation”](#) on page 4-5
- Upon successful application instance creation as defined for the application model, fires an application instance creation event -
`event:///standard/app/created` - with the actual new application instance URI as the event source to all registered listeners
- In case of failure, rolls back any partially completed application instance creation steps
- Provides completion status to the card manager

8.9 Deletion of Application Instance

The card manager application and the card management facility cooperate to delete instances of previously instantiated applications/applets.

Note – In this section, the term *application instance* is used to refer to both a web application instance, as well as an applet instance.

The card manager application typically performs the following functions while deleting an instance of a previously instantiated application/applet:

- Communicates with the off-card client requesting the instance deletion operation
- Checks the credentials of the off-card client and authorizes the request
- Determines the application instance URI of the application instance that is to be deleted
- Instructs the card management facility to delete the instance of the previously created application instance
- Reports progress information to the off-card client
- If the loading attempt is unsuccessful
 - Rolls back all partially completed application instance deletion steps such that the application instance deletion attempt can be retried by the off-card client after the error condition is rectified.

The card management facility is the platform level entity responsible for deleting application instances and communicating with the appropriate platform subsystems to manage the application instance deletion and security requirements. An application instance is successfully deleted when all objects owned by the application instance are inaccessible on the card.

Note – Interned String objects are not bound to any context (group contexts or Java Card RE context) and, therefore, references to these objects from other applications or libraries will not prevent an application instance from being deleted. Similarly, instances of explicitly transferable object classes, namely arrays and String objects, are owned by the group context of the application that created them, not by the application itself and, therefore, references to these objects from other applications or libraries will not prevent an application instance from being deleted.

Prior to deleting an application instance, the card management facility **MUST** ensure that the permission `javacardx.spi.cardmgmt.CardManagementPermission` with the target name `application.delete` is granted to the card manager application.

The card management facility **MUST** perform the following validation steps and other functions during the deletion of an application instance:

- The application instance **MUST** have been previously created and currently registered on the card
- Before attempting to delete the application instance, fires an application instance deletion request event - `event:///standard/app/deleting` - with the application instance URI as the event source to all registered listeners¹
- After sending an application instance deletion request event to all registered listeners, checks to ensure that there are no references to any objects owned by the application instance directly or indirectly (via Java Card RE owned registries) from any another application instance, from any thread which is not dedicated² to the application being deleted, any application module, from any extended library, or from any classic library on the card. If any dependencies exist, the deletion operation **MUST** fail.

The Java Card RE may freeze the other executing threads, including the container managed dispatcher threads on the card during the deletion process to prevent any new dependencies being introduced.

- For a web application:
 - Implements application model specific instance deletion requirements described in [Section 3.2.4, “Application Instance Deletion”](#) on page 3-7

1. The Java Card RE **MAY** limit the effects of an unresponsive listener application from interfering with the instance deletion sequence, in the manner described in [Section 7.4.6.1, “Special Case of Application Instance Deletion Event and Other Platform and Card Management Events”](#) on page 7-26

2. A thread or resource is dedicated to an application if it was created by the application instance itself or by the Java Card RE on behalf of the application

- For an applet application:
 - Implements application model-specific instance deletion requirements described in [Section 4.2.4, “Application Instance Deletion”](#) on page 4-6
- All GCF connections opened by the application instance, including network connections as well as file connections, **MUST** be closed
- All entries associated with the application instance managed by the Java Card Platform facilities **MUST** be deleted
 - All SIO-based service factories registered by the application instance **MUST** be unregistered
 - All event listeners registered by the application instance **MUST** be unregistered
 - All restartable tasks registered by the application instance **MUST** be unregistered
 - All threads and other resources dedicated to the application instance **MUST** be safely stopped and deleted
 - The application/applet instance URI **MUST** be unregistered from the card.
- Upon successful deletion, fires an application instance deletion event - `event:///standard/app/deleted` - with the application instance URI as the event source to all registered listeners
- In case of failure, rolls back any partially completed instance deletion steps
- Provides completion status to the card manager

8.9.1 Multiple Application/Applet Instance Deletion

The card manager application and the card management facility **MUST** also cooperate to support the atomic deletion of multiple instances of previously instantiated applications/applets. The maximum number of application instances that may be deleted atomically is implementation dependent, but the deletion of two application instance **MUST** be minimally supported.

The process of deleting multiple instances is similar to that described for deleting a single application instance ([Section 8.9, “Deletion of Application Instance”](#) on page 8-33), with each application instance being deleted sequentially, except for the step to perform dependency checks, which are combined for all the application instances. The dependency checks **MUST** be performed on objects owned by all the application instances being deleted as one unit. The deletion process **MUST** fail if any object owned by any application instance being deleted is referenced from any other application instance or thread (not being deleted), any application module, any extended library, or any classic library.

8.10 Unloading of Deployment Units

The card manager application and the card management facility cooperate to delete previously loaded deployment units on the card. Application modules are deployment units which encapsulate applications. Extended libraries and classic libraries are deployment units which encapsulate libraries.

The card manager application typically performs the following functions during the unloading of a deployment unit:

- Communicates with the off-card client requesting the deployment unit deletion operation
- Checks the credentials of the off-card client and authorizes the request
- Determines the free-form name of the deployment unit
- Instructs the card management facility to unload the deployment unit, passing it information about the deployment unit
- Reports progress information to the off-card client
- If the unloading attempt is unsuccessful
 - Rolls back all partially completed unloading steps such that the unloading operation can be retried by the off-card client after the error condition is rectified

The card management facility is the platform level entity responsible for unloading previously loaded deployment units and communicating with the appropriate platform subsystems to perform cleanup actions. A deployment unit is successfully deleted when all the classes, resources and descriptors comprising the deployment unit are not accessible on the card.

Prior to unloading a deployment unit, the card management facility **MUST** ensure that the permission `javacardx.spi.cardmgmt.CardManagementPermission` with the target name `deploymentUnit.unload` is granted to the card manager application.

The card management facility **MUST** perform the following validation steps and other functions during the unloading of a deployment unit:

- The application module or library **MUST** have been previously loaded on the card.
- Checks to ensure that there are no instantiated classes of the application module or library on the card. If any dependencies exist, the deletion operation **MUST** fail.

- Checks to ensure that there are no references to instances of explicitly transferable object classes, namely arrays and `String` objects, owned by the group context of the application module from any application instance, any thread, any other application module, or any library on the card. If any dependencies exist, the unloading operation **MUST** fail.
- Checks to ensure that there are no references to the application module or library from any application instance, any thread, any other application module, or any library on the card. If unloading a library, checks to ensure that there is no “library unloading dependency” on the library being unloaded (see [Section 8.7, “Loading Libraries” on page 8-29](#)). If any dependencies exist, the unloading operation **MUST** fail.

The Java Card RE may freeze the other executing threads, including the container managed dispatcher threads on the card during the deletion process to prevent any new dependencies being introduced

- For a web application:
 - Implements application model specific unloading requirements for each component application module contained in the application module as described in [Section 3.2.5, “Application Module Unloading” on page 3-8](#)
- For an applet application:
 - Implements application model-specific unloading requirements for each component application module contained in the application module as described in [Section 4.2.5, “Applet Application Module Unloading” on page 4-6](#)
- In case of failure, rolls back any partially completed deployment unit unloading steps
- Provides completion status to the card manager

8.10.1 Application Module Unloading with Instance Deletion

The card manager application and the card management facility **MUST** also cooperate to support the atomic unloading of an application module combined with the deletion of all application instances from the application module.

The process to unload the application module and delete all contained instances is a combination of the process described for unloading an application module ([Section 8.10, “Unloading of Deployment Units” on page 8-36](#)) and that for deleting multiple application instances ([Section 8.9.1, “Multiple Application/Applet Instance Deletion” on page 8-35](#)), except for the step to perform dependency checks, which are also combined. The dependency checks **MUST** be performed on the application module and all objects owned by all the application instances being deleted as one

unit. The deletion process **MUST** fail if the application module or any object owned by any application instance being deleted is referenced from any other application instance or thread (not being deleted), any other application module, or any library.

File System

This chapter describes the optional file system supported on the Java Card Platform, Connected Edition, specifically:

- [File System Requirements](#)
- [File System Object Identification](#)
- [File Access Permissions](#)
- [Atomicity and Transactional Behavior](#)
- [Generic Connection Framework-based File Access](#)
- [File Resource Event Notifications](#)
- [Thread Safety](#)
- [Platform Reset Behavior](#)

9.1 File System Requirements

A Java Card Platform implementation MAY support one or several file systems. Each of these file systems MUST have the following characteristics:

- It MUST be a hierarchical file system of directories and files.
- For each file system object, generic read and write access permission attributes MUST be maintained, see [Section 9.3, “File Access Permissions”](#) on page 9-3.
- It MUST be exclusively managed and accessible from the Java Card Platform. That is, the underlying data storage and the file system itself cannot be concurrently or alternatively accessed other than through the Java Card Platform.
- It MUST provide an interface that can be exposed through the Generic Connection Framework as a `javacardx.io.FileConnection`, see [Section 9.5, “Generic Connection Framework-based File Access”](#) on page 9-5.

The underlying data storage MAY be partitioned and each partition MAY support a distinct physical file system, but the Java Card Platform MUST support all file system objects across all file systems to be uniformly accessed using file URIs, see [Section 9.2, “File System Object Identification” on page 9-2](#).

9.2 File System Object Identification

To each valid file system path to a file system object (directory or file) MUST correspond a unique canonical file URI. The path name component of a file URI MUST follow the syntax of a URI path component, which uses the slash character ‘/’ as the path separator.

A *canonical* file URI MUST satisfy the following requirements:

- it MUST have a `file` URI scheme,
- it MUST have a normalized path with the following characteristics:
 - it MUST NOT contain any “.” or “..” path name components
 - it MUST use the slash character ‘/’ as the path name component separator.

The path component of a file URI MAY correspond to a *virtual* path that the platform MUST map to a *real* path on the file system(s) it supports. In such a case, the platform MUST ensure that the constraints that apply to the file URIs are also enforced on the corresponding real paths. For example, dedicated application namespaces MUST NOT overlap, see [Section 9.2.1, “Application-private File System Objects” on page 9-3](#), therefore the corresponding dedicated directories on the file system(s) MUST NOT overlap in any manner. Similarly, the Java Card Platform MUST ensure that the real paths to file system objects identified with URIs that use the `aid` registry-based authority do not overlap with any real path of file system objects identified with URIs that use the default registry-based authority.

9.2.1 Application-private File System Objects

Every application instance is named with a relative URI - its *application URI*. This application URI defines the root of a dedicated namespace within which all its resources, including files, MUST be named. The root of an application's file namespace is of the form given in [TABLE 9-1](#).

TABLE 9-1 Application-private File System Object Identification

Service Description	File URI
Application-private file namespace root	<code>file: //<app-path>:</code> <ul style="list-style-type: none">• <code>file: //<context path></code>• <code>file: //aid/<RID>/<PIX></code>
Application-private files and directories	<code>file: //<app-path>/<app-relative-path>:</code> <ul style="list-style-type: none">• <code>file: //<context path>/<app-relative-path></code>• <code>file: //aid/<RID>/<PIX>/<app-relative-path></code>

All files in an application's namespace MUST only be accessible to that application. These files MUST be owned by that application. See [Section 9.3, “File Access Permissions”](#) on page 9-3.

Any attempt to access a file under another application's namespace MUST result in a security exception.

Any attempt to access a file under the reserved `file:///platform`, `file:///standard`, `file://aid/platform` and `file://aid/standard` namespaces MUST result in a security exception.

9.3 File Access Permissions

On the Java Card Platform, a file system MUST implement restrictions to read and write operations on the actual file system objects. These restrictions are known as *access permissions*.

A file system MUST manage a generic set of read and write access permission attributes on every file system object. This set of attributes constitutes the object's *file permissions mode*. The file permissions mode of a file system object MUST apply to all applications without distinctions.

Operations on a file system object that require read or write access to that file system object MUST fail if that file system object's permissions mode does not grant the required permission.

TABLE 9-2 defines how a file system object’s permissions modes MUST be interpreted to grant read and write permissions for file system operations.

TABLE 9-2 Operations Granted By Read and Write Permissions on Files and Directories

Permission/File System Object	On a File	On a Directory
Read	Reading this file’s content and attributes.	Reading the names (but not other attributes) of files and subdirectories in this directory.
Write	Modifying (appending or overwriting) this file’s content. Deleting this file, only if write permission is also set on the containing directory.	Creating new files and subdirectories in this directory. Renaming existing files and subdirectories contained in this directory. Deleting subdirectories contained in this directory. Deleting existing files contained in this directory, only if write permission is also set on these files and subdirectories.

In addition to the file system access permissions, access to a file system object MUST be controlled by the permissions granted by the protection domain of the application attempting to access the object. These protection domain permissions are `javacardx.io.ConnectorPermission` objects with file URI target names. Permission checks MUST apply on the canonical forms of the file URI target names before translation to real file system paths that may apply. See [Section 9.2, “File System Object Identification”](#) on page 9-2.

9.4 Atomicity and Transactional Behavior

The Java Card Platform MUST ensure the atomicity of operations that affect the hierarchical structure of file systems, that is operations on directory entries:

- creation of a new file or directory,
- deletion of a file or directory,
- renaming of a file or directory within the same physical file system,
- change of file or directory attributes such as access permissions and last modification time.

All other operations on file content, such as read and write operations, are not transactional.

9.5 Generic Connection Framework-based File Access

The Java Card Platform MUST exclusively support access to file system objects through the Generic Connection Framework as follows:

- using a `javacardx.io.FileConnection` object returned by a call to a `javax.microedition.io.Connector.open` method with a file URL parameter,
- using an input or output stream object returned by a call to a `javax.microedition.io.Connector.openXXXStream` method with a file URL parameter.

If a Java Card Platform implementation does not support a file system, any attempt to open a file URL using the Generic Connection Framework MUST result in a `javax.microedition.io.ConnectionNotFoundException` being thrown.

9.6 File Resource Event Notifications

A developer may use the standard resource event URIs defined in the namespace rooted at `event:///standard/rsrc` to notify other applications of lifecycle events on file resources his/her application manages. See [Section 7.4.1.1, “Platform and Standard Events”](#) on page 7-19 for more details about resource lifecycle events.

Note – The mechanism an application may use to share the content of a file with other applications that may have registered for such events are beyond the scope of this specification, but can be implemented by developers using the shareable interface and object ownership transfer mechanisms, see [Section 7.3, “Shareable Interface Object-based Services”](#) on page 7-7 and [Section 7.2, “Object Ownership Transfer Mechanism”](#) on page 7-3.

9.7 Thread Safety

The atomic operations that affect the hierarchical structure of file systems, that is operation on directory entries, such as the creation, the deletion and the renaming of file or directories, **MUST** be thread safe.

All other operations on file system objects are not guaranteed to be thread safe. The application developer must properly account for thread safety and should handle synchronization explicitly.

9.8 Platform Reset Behavior

After a platform reset, the Java Card Platform **MUST** perform the following steps before any request or command can be dispatched to any application instance:

1. Any persistent GCF file connection objects, instances of classes implementing the `javacardx.io.FileConnection` interface, and associated input or output streams that are reachable from the persistence root **MUST** be placed in a "closed" state.
2. All files and directories for which a delete upon platform reset has been successfully requested through a call to the `FileConnection.deleteOnReset` **MUST** be deleted.

Refer to [Chapter 5](#) for the other generic requirements that **MUST** be implemented by the Java Card Platform after a platform reset.

Default Platform Security Policy

This chapter describes the default *platform security policy*. The platform security policy defines for each of the three application models supported on the Java Card Platform a *platform protection domain* that guarantees the consistency and integrity of the applications implementing each of these application models. Additionally, the platform security policy defines a platform protection domain for the card management applications. Each of these platform protection domains defines the minimum¹ set of permissions granted to an application of the corresponding type and is defined as a set of included permissions as well as a set of excluded permissions so that no additional permissions can be granted that may violate the platform security policy. See [Chapter 6, Section 6.2.2, “Protection Domains” on page 6-13](#) for more information on the use of protection domains.

The default platform protection domains defined in this chapter MAY be tuned for specific environments, provided the consistency and integrity of each application model, and of the platform itself, is guaranteed.

Refer to [Chapter 6, TABLE 6-1](#) and [TABLE 6-2](#) for a description of the different permission classes used to define the default platform protection domains.

1. Additional permissions may be granted by the card management security policy.

A.1 Permissions in Default Protection Domain for Web Applications

The default protection domain for web applications **MUST** include the set of included permissions listed in [TABLE A-1](#).

TABLE A-1 Default Included Permission Set of the Default Web Protection Domain

Permission Description	Permission Class Permission Name Permission Actions List
Calling methods of Java Card RE-owned instances of <i>Extended</i> set of Permanent Java Card RE EPO	<code>javacardx.spi.framework.JCREPermission</code> <ul style="list-style-type: none">• <code>callPermJCREEPO.EXTENDED</code>
Calling methods of Java Card RE-owned instances of <i>classic</i> set of Permanent Java Card RE EPO	<code>javacardx.spi.framework.JCREPermission</code> <ul style="list-style-type: none">• <code>callPermJCREEPO.CLASSIC</code>
Firing events and registering, unregistering and listing event listeners in one's own namespace	<code>javacardx.facilities.EventRegistryPermission</code> <ul style="list-style-type: none">• <code>event:///~/*</code>• <code>notify,register,unregister</code>
Registering and unregistering platform event listeners	<code>javacardx.facilities.EventRegistryPermission</code> <ul style="list-style-type: none">• <code>event:///platform/*</code>• <code>register,unregister</code>
Registering and unregistering standard application event listeners	<code>javacardx.facilities.EventRegistryPermission</code> <ul style="list-style-type: none">• <code>event:///standard/*</code>• <code>register,unregister</code>
Looking up, registering and unregistering services in one's own namespace	<code>javacardx.facilities.ServiceRegistryPermission</code> <ul style="list-style-type: none">• <code>sio:///~/*</code>• <code>lookup,register,unregister</code>
Looking up and listing authenticator services	<code>javacardx.facilities.ServiceRegistryPermission</code> <ul style="list-style-type: none">• <code>sio:///standard/auth/*</code>• <code>lookup</code>
Registering and unregistering of restartable tasks	<code>javacardx.facilities.TaskRegistryPermission</code> <ul style="list-style-type: none">• <code>task.*</code>
Context switching and transferring ownership of objects to authenticator applications	<code>javacardx.framework.ContextPermission</code> <ul style="list-style-type: none">• <code>sio:///standard/auth/*</code>• <code>switch,transfer</code>

TABLE A-1 Default Included Permission Set of the Default Web Protection Domain

Permission Description	Permission Class
	Permission Name Permission Actions List
Setting and getting one’s own credential manager(s)	javacardx.framework.JCRuntimePermission • credentialManager.*
Creating new threads and modifying threads’ states	javacardx.framework.JCRuntimePermission • thread.*
Reading files from one’s own namespace	javacardx.io.ConnectorPermission • file:///~/* • read

The default protection domain for web applications MUST include the set of excluded permissions listed in [TABLE A-2](#).

TABLE A-2 Excluded Permission Set of the Default Web Protection Domain

Permission Description	Permission Class
	Permission Name Permission Actions List
Calling methods of Java Card RE-owned instances of any temporary Java Card RE EPO or Global Arrays*	javacardx.spi.framework.JCREPermission • callTempJCREEPO.*

* This excluded permission is implicitly enforced by ensuring that only application-owned instances, such as of web container-managed objects, are created by the web container and the Java Card RE on behalf of web applications, see [Section 3.2.9, “Container-managed Object Lifetime and Persistence”](#) on page 3-11.

A.2 Permissions in Default Protection Domain for Extended Applets

The default protection domain for extended applets MUST include the set of included permissions listed in [TABLE A-3](#).

TABLE A-3 Default Included Permission Set of the Default Extended Protection Domain

Permission Description	Permission Class Permission Name Permission Actions List
Calling methods of Java Card RE-owned instances of <i>Extended</i> set of Permanent Java Card RE EPO	<code>javacardx.spi.framework.JCREPermission</code> <ul style="list-style-type: none">• <code>callPermJCREEPO.EXTENDED</code>
Calling methods of Java Card RE-owned instances of <i>classic</i> set of Permanent Java Card RE EPO	<code>javacardx.spi.framework.JCREPermission</code> <ul style="list-style-type: none">• <code>callPermJCREEPO.CLASSIC</code>
Calling methods of Java Card RE-owned instances of <i>classic</i> set of Temporary Java Card RE EPO and Global Arrays	<code>javacardx.spi.framework.JCREPermission</code> <ul style="list-style-type: none">• <code>callTempJCREEPO.CLASSIC</code>
Firing events and registering, unregistering and listing event listeners in one's own namespace	<code>javacardx.facilities.EventRegistryPermission</code> <ul style="list-style-type: none">• <code>event://aid/~/*</code>• <code>notify,register,unregister</code>
Registering and unregistering platform event listeners	<code>javacardx.facilities.EventRegistryPermission</code> <ul style="list-style-type: none">• <code>event:///platform/*</code>• <code>register,unregister</code>
Registering and unregistering standard application event listeners	<code>javacardx.facilities.EventRegistryPermission</code> <ul style="list-style-type: none">• <code>event:///standard/*</code>• <code>register,unregister</code>
Looking up, registering and unregistering services in one's own namespace	<code>javacardx.facilities.ServiceRegistryPermission</code> <ul style="list-style-type: none">• <code>sio://aid/~/*</code>• <code>lookup,register,unregister</code>
Looking up and listing authenticator services	<code>javacardx.facilities.ServiceRegistryPermission</code> <ul style="list-style-type: none">• <code>sio:///standard/auth/*</code>• <code>lookup</code>
Registering and unregistering of restartable tasks	<code>javacardx.facilities.TaskRegistryPermission</code> <ul style="list-style-type: none">• <code>task.*</code>

TABLE A-3 Default Included Permission Set of the Default Extended Protection Domain

Permission Description	Permission Class Permission Name Permission Actions List
Context switching and transferring ownership of objects to authenticator applications	javacardx.framework.ContextPermission • sio:///standard/auth/* • switch,transfer
Setting and getting one’s own credential manager(s)	javacardx.framework.JCRuntimePermission • credentialManager.*
Creating new threads and modifying threads’ states	javacardx.framework.JCRuntimePermission • thread.*
Reading files from one’s own namespace	javacardx.io.ConnectorPermission • file://aid/~/* • read

The default protection domain for extended applets MUST include the set of excluded permissions listed in [TABLE A-4](#).

TABLE A-4 Excluded Permission Set of the Default Extended Protection Domain

Permission Description	Permission Class Permission Name Permission Actions List
NONE	NONE

A.3 Permissions in Default Protection Domain for Classic Applets

The default protection domain for classic applets MUST include the set of included permissions listed in [TABLE A-5](#).

TABLE A-5 Default Included Permission Set of the Default Classic Protection Domain

Permission Description	Permission Class Permission Name Permission Actions List
Calling methods of Java Card RE-owned instances of <i>classic</i> set of Permanent Java Card RE EPO	javacardx.spi.framework.JCREPermission • callPermJCREEPO.CLASSIC
Calling methods of Java Card RE-owned instances of <i>classic</i> set of Temporary Java Card RE EPO and Global Arrays	javacardx.spi.framework.JCREPermission • callTempJCREEPO.CLASSIC

The default protection domain for classic applets MUST include the set of excluded permissions listed in [TABLE A-6](#).

TABLE A-6 Excluded Permission Set of the Default Classic Protection Domain

Permission Description	Permission Class Permission Name Permission Actions List
Calling methods of Java Card RE-owned instances of <i>Extended</i> set of Permanent Java Card RE EPO	javacardx.spi.framework.JCREPermission • callPermJCREEPO.EXTENDED
Transferring ownership of objects to any application	javacardx.framework.ContextPermission • /**/* • transfer
Setting and getting one's own credential manager	javacardx.framework.JCRuntimePermission • credentialManager.*
Creating new threads and modifying threads' states	javacardx.framework.JCRuntimePermission • thread.*
Using the Generic Connection Framework	javacardx.io.ConnectorPermission • * • accept,listen,connect,read,write

A.4 Permissions in Default Protection Domain for Card Management Applications

The default protection domain for card management applications MUST include the set of included permissions listed in [TABLE A-7](#).

TABLE A-7 Default Included Permission Set of the Default CardManagement Protection Domain

Permission Description	Permission Class Permission Name Permission Actions List
Setting and getting of an application's credential manager(s)	<code>javacardx.spi.cardmgmt.CardManagementPermission</code> <ul style="list-style-type: none"><code>credentialManager.*</code>
Calling methods of Java Card RE-owned instances of <i>Extended</i> set of Permanent Java Card RE EPO	<code>javacardx.spi.framework.JCREPermission</code> <ul style="list-style-type: none"><code>callPermJCREEPO.EXTENDED</code>
Calling methods of Java Card RE-owned instances of <i>classic</i> set of Permanent Java Card RE EPO	<code>javacardx.spi.framework.JCREPermission</code> <ul style="list-style-type: none"><code>callPermJCREEPO.CLASSIC</code>
Calling methods of Java Card RE-owned instances of <i>classic</i> set of Temporary Java Card RE EPO and Global Arrays	<code>javacardx.spi.framework.JCREPermission</code> <ul style="list-style-type: none"><code>callTempJCREEPO.CLASSIC</code>
Firing events and registering, unregistering and listing event listeners in one's own namespace	<code>javacardx.facilities.EventRegistryPermission</code> <ul style="list-style-type: none"><code>event:/*/~/*</code><code>notify,register,unregister</code>
Registering and unregistering platform event listeners	<code>javacardx.facilities.EventRegistryPermission</code> <ul style="list-style-type: none"><code>event:/*/platform/*</code><code>register,unregister</code>
Firing, registering for and unregistering for standard application event listeners	<code>javacardx.facilities.EventRegistryPermission</code> <ul style="list-style-type: none"><code>event:/*/standard/*</code><code>notify,register,unregister</code>
Looking up, registering and unregistering services in one's own namespace	<code>javacardx.facilities.ServiceRegistryPermission</code> <ul style="list-style-type: none"><code>sio:/*/~/*</code><code>lookup,register,unregister</code>
Looking up, registering, unregistering and listing authenticator services	<code>javacardx.facilities.ServiceRegistryPermission</code> <ul style="list-style-type: none"><code>sio:/*/standard/auth/*</code><code>register,unregister,lookup</code>

TABLE A-7 Default Included Permission Set of the Default CardManagement Protection Domain

Permission Description	Permission Class Permission Name Permission Actions List
Registering and unregistering of restartable tasks	javacardx.facilities.TaskRegistryPermission • task.*
Context switching and transferring ownership of objects to authenticator applications	javacardx.framework.ContextPermission • sio://*/standard/auth/* • switch,transfer
Setting and getting one’s own credential manager(s)	javacardx.framework.JCRuntimePermission • credentialManager.*
Creating new threads and modifying threads’ states	javacardx.framework.JCRuntimePermission • thread.*
Reading files from one’s own namespace	javacardx.io.ConnectorPermission • file://*/~/* • read

The default protection domain for card management applications MUST include the set of excluded permissions listed in [TABLE A-8](#).

TABLE A-8 Excluded Permission Set of the Default CardManagement Protection Domain

Permission Description	Permission Class Permission Name Permission Actions List
NONE	NONE

Security Annotations

This chapter describes optional *security annotations*, which can be used by developers to secure their applications. Annotations are very useful when implementing security for many features, including user authentication, secure communication channels, and local encryption.

However, on smart card platforms, application-level security is not always sufficient. Smart cards are widely deployed and they may fall in the wrong hands. This means that, beyond software-based attacks, it is also possible to perform other, physical attacks on cards. Such physical attacks are considered in most smart card risk analyses and countermeasures exist for them. The two most significant categories of such physical attacks are:

- **Side-channel observation attacks.** The attacker observes a side effect of the program's execution in order to determine some properties of the application and/or its data. The most well-known examples are the various versions of power analysis, which can be used to retrieve the values of cryptographic keys.
- **Perturbation or fault induction attacks.** The attacker exploits a property of the silicon media on which all chips are based to disturb the execution of a program and, for instance, skip a part of the execution.

These physical attacks usually target either the confidentiality of sensitive data or the integrity of sensitive code and data. While it is possible to defend against such attacks at the application level, because the attacks target low-level features, such as cryptographic routines, it is easier to implement countermeasures at the platform level.

Some of these countermeasures can be activated continuously, as, for instance, when they can be systematically activated during cryptographic computations. In contrast, some countermeasures cannot be active at all times, especially for an application that manages mostly non-sensitive data. These countermeasures often prove too costly to implement when activating them at all times would result in performance issues.

The set of annotations defined in this chapter allow a developer to declare the security requirements of an application regarding the platform. The objective of such applications is to associate the source code with a set of security requirements that indicate the parts of the code whose confidentiality and/or integrity are particularly sensitive.

More precisely, this set of annotations is visible in the class file, but without mandating any particular behavior at the platform level. The choice of the implementation is left to the platform developer and to the application developer. The annotations are used to define security requirements in an unambiguous way, but they do not mandate any particular kind of countermeasure. Among the possible implementations are to:

- Launch some specific mechanisms inside the virtual machine, but with no specific behavior being mandatory. This is what currently happens when each platform provider has, along with the application code, the security policy it wants to enforce. In this scenario, it is up to the platform provider to decide which protections will be run in order to guarantee the policy.
- Use an external tool to process the source code and its annotations and perform some transformations that enhance the application code's resistance to attacks.
- Use an external tool to process the byte code (because the annotations are visible inside the class file) and increase the code's resistance to attacks in a way similar to the preceding scenario.

These techniques can be used separately, or they can be combined. Standard APIs can be used, or you can rely on proprietary APIs. As mentioned above, there are no constraints on the implementation.

This lack of implementation constraint is possible because these security measures should not modify the nominal behavior of the application. As such, they cannot be tested, because they have no visible effect on the execution of the application. This is why no compliance testing can be defined for this feature, and this is also why interoperability is not the main objective here.

Although security cannot be tested using a functional testing tool, there are standardized test procedures, vulnerability assessments, in which platforms are subjected to attacks by specialized laboratories and must demonstrate their resistance to a given attack level. These standardized procedures are defined in ISO-15408, which defines the Common Criteria for Information Technology Security Evaluation (known as Common Criteria or “CC”). The detailed definition of security levels for CC evaluations is outside of the scope of this document, as it should be included in the definition of a Protection Profile for Java Card-based platforms.

These security annotations allow an application developer to document the basic security requirements of an application by associating confidentiality and integrity requirements to classes and methods. Such annotations are expected to be used in security evaluations of the application to rapidly identify which security-sensitive

features were identified. This is particularly important in contexts where composition of evaluations is used, as applications should use the features offered by the underlying platform in order to protect their most sensitive assets.

B.1 Annotations Defined

The required level of security can be defined for a class, for an interface or for a method, but not for a field. If a field is protected, the methods that use the fields will need to perform special operations, and the execution of methods that use these fields will, therefore, need to be adapted accordingly.

B.1.1 Type Annotations

Annotations can be defined for a class or an interface.

B.1.1.1 Class Annotations

By tagging an entire class, you can indicate that a given set of fields are sensitive, together with the methods that handle them. This annotation is inherited. Once a class has been annotated as sensitive, all its subclasses are also considered sensitive.

The following code shows security annotation usage for sensitivity tagging of a class.

CODE EXAMPLE B-1 Class Security Annotation Example

```
@SensitiveType(  
    sensitivity=SensitiveValue.CONFIDENTIALITY,  
    proprietaryValue="Transit"  
)  
public class TicketBookSIO implements TicketBookSI,  
TicketBookControlSI {  
    public int getBalance() {...}  
    public int credit(int count) {...}  
    public int debit(int count) {...}  
    public void unblock() {...}  
}
```

B.1.1.2 `SensitiveType(sensitivity=INTEGRITY)`

Tagging a class as sensitive in `INTEGRITY` indicates that:

- Consistency of the data (the fields) of the class is important. If this data is modified outside the scope of the normal proposed services, it could be detrimental to the application, the application user, and/or the application provider.
- Consistency of the services (the methods) of the class is important. If these services' behavior is modified outside the scope of the normal proposed services, it could be detrimental to the application, the application user and/or the application provider.

B.1.1.3 `SensitiveType(sensitivity=CONFIDENTIALITY)`

Tagging a class as sensitive in `CONFIDENTIALITY` indicates that:

- Hiding the data (the fields) of the class from the outside is important. If this data is disclosed outside the application, it could be detrimental to the application, the application user, and/or the application provider.
- Hiding the data handling by the services (the methods) of the class from the outside is important. If the data handling by these services is disclosed outside the application, it could be detrimental to the application, the application user and/or the application provider.

B.1.1.4 `SensitiveType(sensitivity=FULL)`

Tagging a class as `FULL` in sensitivity is equivalent to annotating the class both as sensitive in `INTEGRITY` and in `CONFIDENTIALITY`.

B.1.1.5 `SensitiveType(proprietaryValue="...")`

Tagging a class with a `proprietaryValue` value provides additional information that can be used by proprietary tools or the platform to perform more precise tasks. This element can be used by each platform provider to give its own proprietary information and, thereby, improve its tools' processing or VM execution. This information may be ignored if processing information from another platform provider.

B.1.2 Interface Annotation

Annotations of implemented interfaces have no effect on a class. Nevertheless, the `SensitiveType` annotation can be used to annotate an interface, in which case, it has just an informative purpose and simply describes the security level that is expected of classes which implements the interface. The effective security annotation is only that defined on the classes which implement the interface.

The following code shows security annotation usage for sensitivity tagging of an interface. The effective security annotation is only that of the `TicketBookSIO` class given in [CODE EXAMPLE B-1](#).

CODE EXAMPLE B-2 Interface Security Annotation Example

```
@SensitiveType(  
    sensitivity=CONFIDENTIALITY,  
    proprietaryValue="Transit"  
)  
public interface TicketBookSI {  
    int getBalance();  
    int credit(int count);  
    int debit(int count);  
}
```

B.1.3 Method Annotations

Method annotations are used to tag specific methods as sensitive. Method-scoped annotations cannot be inherited, so every individual sensitive method needs to be annotated as required.

The following code shows security annotation usage for sensitivity tagging of a method. The annotation of the unblock method with sensitivity=INTEGRITY overrides the type annotation with sensitivity=CONFIDENTIALITY of the TicketBookSIO class.

CODE EXAMPLE B-3 Method Security Annotation Example

```
public class TicketBookSIO implements TicketBookSI,
    TicketBookControlSI {
    public int getBalance() {...}
    public int credit(int count) {...}
    public int debit(int count) {...}

    @SensitiveType(
        sensitivity=SensitiveValue.INTEGRITY,
        proprietaryValue="Transit"
    )
    public void unblock() {...}
}
```

B.1.3.1 SensitiveMethod(sensitivity=INTEGRITY)

Tagging a method as sensitive in INTEGRITY indicates that consistency of this method is important. If the method's behavior is modified outside the scope of the normal proposed services, it could be detrimental to the application, the application user and/or the application provider.

B.1.3.2 SensitiveMethod(sensitivity=CONFIDENTIALITY)

Tagging a method as sensitive in CONFIDENTIALITY indicates that hiding the data handling by this method from the outside is important. If the data handling by this method is disclosed outside the application, it could be detrimental to the application, the application user and/or the application provider.

B.1.3.3 SensitiveMethod(sensitivity=FULL)

Tagging a method as FULL in sensitivity is equivalent to annotating the method both as sensitive in INTEGRITY and in CONFIDENTIALITY.

B.1.3.4 SensitiveMethod(proprietaryValue="...")

Tagging a method with a `proprietaryValue` value provides additional information that can be used by proprietary tools or the platform to perform more precise tasks. This element can be used by each platform provider to give its own proprietary information and, thereby, improve its tools' processing or VM execution. This information may be ignored if processing information from another platform provider.

B.2 Semantics of Annotations

B.2.1 Scope of Annotations

The scope of security annotations is determined at runtime, as follows:

- If a method that is declared sensitive in `INTEGRITY` (respectively `CONFIDENTIALITY`) runs, then the integrity-preserving (respectively confidentiality-preserving) countermeasures are activated during its execution.
- The activated countermeasures remain active until the end of the method's execution, including the execution of any method that is invoked from the method initially declared sensitive. This includes in particular methods from parent classes.
- The activated countermeasures remain active until the end of the method's execution, regardless of the way in which the execution ends (method return or exception), unless the countermeasures were already active when the execution started.
- The countermeasures are also assumed to remain active during execution of API methods.
- The countermeasures are activated separately on distinct threads.

In some cases, the behavior is not defined:

- If a field of a class that is declared sensitive in `INTEGRITY` or in `CONFIDENTIALITY` is not declared private and is accessed from a method outside of its declaring class, the resulting security behavior is undefined.

Although the scope of the annotations is normally known only at runtime, it is also possible to determine it at deployment time, once the classes of an application have been bundled, by using simple static analysis techniques.

B.2.2 Annotated APIs

Some of the classes and interfaces specified in *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition* are assumed to be annotated. Because of inheritance, all their subclasses are also considered sensitive. For instance, all `Key` objects are integrity sensitive, and all `SecretKey` and `PrivateKey` objects are also confidentiality sensitive.

Glossary

access control mechanism	a mechanism that permits or denies the access to a particular resource by a particular entity. An access control mechanism enforces a security policy.
active applet instance	an applet instance that is selected on at least one of the logical channels.
AID (application identifier)	<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/<RID>/<PIX></pre> <p>where <RID> (resource identifier) and <PIX> (proprietary identifier extension) are components of the AID.</p>
APDU	an acronym for Application Protocol Data Unit as defined in ISO 7816-4.
APDU-based application environment	consists of all the functionalities and system services available to applet applications, such as the services provided by the applet container.
API	an acronym for Application Programming Interface. The API defines calling conventions by which an application program accesses the operating system and other services.

applet	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.
applet application	an application that consists of one or more applets.
applet container	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.
applet framework	an API that enables applet applications to be built.
applicable credential manager	the credential manager instance, application-assigned or card manager-assigned, that applies for a particular mode of communication. See credential manager .
applicable security requirements	the security requirements instance, application-assigned or card manager-assigned, that applies for a particular mode of communication. See security requirements .
application assembler	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.
application-defined event	an event that an application may define in its own namespace and may be the only one allowed to fire.
application-defined service	a service that an application may define in its own namespace and may be the only one allowed to register.
application descriptor	see descriptor .
application developer	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.
application firewall	see firewall .
application framework class loader	a direct child of the extension library class loader, in charge of loading application framework libraries shared among a restricted set of application groups.
application group	a set of one or more applications executing in a common group context.

application-managed authentication	authentication that is programmatically triggered by an application's code based on some business logic.
application-managed connection endpoint	a client or server connection endpoint managed directly by an application.
application module class loader	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.
application protection domain	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.
application security policy	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.
application URI	a URI uniquely identifying an application instance on the platform.
atomic operation	an operation that either completes in its entirety or no part of the operation completes at all.
atomicity	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.
authentication	the process of establishing or confirming an application or a user as authentic using some sort of credentials
authenticator	an authentication service that can be invoked both by applications for application-managed authentication and by the web container for container-managed authentication.
authorization	the process of allowing access to those resources by entities (applications or users) that have been granted authority to use them.
basic logical channel	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.
bootstrap class loader	the root of the class loader delegation hierarchy in charge of loading the Java Card RE system classes.
bytecode	machine-independent code generated by the compiler and executed by the Java virtual machine.
canonicalization (URI)	the combined process of resolving a URI against a base URI, then normalizing it.

card holder	the primary user of a smart card.
card holder-facing client	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.
card holder user	a user whose identity may be assumed by the card holder.
card manager	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.
card management facility	the Java Card platform layer responsible for securely adding and removing application code and instances onto the platform.
card management security policy	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.
card session	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.
classic applet	applets with the same capabilities as those in previous versions of the Java Card platform and in the Classic Edition.
classic applet container mutex object	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.
Classic Edition	one of the two editions in the Java Card Platform, Version 3. 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.
classic library	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.
classic library class loader	a direct child of the shareable interface class loader in charge of loading classic library classes.
classic SIO proxy	see <i>classic SIO synchronization proxy</i> .

classic SIO synchronization proxy	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.
class loader	a Java Card RE component that defines and enforces a different class namespace for the classes it loads.
class loader delegation hierarchy	the hierarchy of class loaders that enforces code isolation among applications while allowing for sharing of system and library code.
client application	an on-card application that uses services provided by other applications (server applications).
client-role-based security	see <i>role-based security</i> .
Connected Edition	one of the two editions in the Java Card Platform, Version 3. 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.
connection endpoint (client, server)	see <i>application-managed connection endpoint</i> , <i>container-managed connection endpoint</i> .
container-managed authentication	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.
container-managed connection endpoint	a server connection endpoint managed by a container, such as an HTTP or HTTPS server connection endpoint managed by the servlet container.
container-managed object	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> .
context path	the path within the web server a servlet context is rooted at. The context path of a web application corresponds to its application URI.

context switch	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.
converter	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.
credential	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.
credential manager	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.
currently active context	when an object instance method is invoked, an owning context of the object becomes the currently active context for that particular thread of execution.
currently active namespace	corresponds to the application owner identifier of the active context set upon entry into the group context for a particular thread of execution.
currently selected applet	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.
declarative security	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.
default applet	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.
default servlet	the application-defined servlet that is used to serve a request when no other servlet applies.
default default servlet	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.

deployer	<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>
deployment unit	entity that can be distributed, deployed and installed on the Java Card platform.
deployment descriptor	see descriptor .
descriptor	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.
distribution format	structure and encoding of a distribution or deployment unit intended for public distribution.
distribution unit	see deployment unit .
EEPROM	an acronym for Electrically Erasable, Programmable Read Only Memory.
entry point method	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.
event	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.
event consuming application	an application that registers for notification of events fired by an event producing application.
event listener	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.

event notification facility	a Java Card RE facility (or subsystem) that is used for event-driven inter-application communications.
event notification listener	see <i>event listener</i> .
event producing application	an application that fires events.
event registry	the core component of the event notification facility. The event registry is used for registering for notification of events and for notifying of events.
event URI	a URI that uniquely identifies an event produced by an event-producing application.
export file	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.
extended applet	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.
extension library	library that extends the functionality of the platform.
extension library class loader	a direct child of the shareable interface class loader in the class loader delegation hierarchy in charge of loading extension libraries.
externally visible	<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>Runtime Environment Specification, Java Card Platform, Version 3.0.1, 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>
file permissions mode	an attribute of a file system object that indicates whether read or write operation on the object are permitted or denied.
filter	a web application component that is used to transform the content or header information of HTTP requests or responses.

finalization	<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>
firewall	the mechanism that prevents unauthorized accesses to objects in one application group context from another application group context.
flash memory	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.
garbage collection	the process by which dynamically allocated storage is automatically reclaimed during the execution of a program.
global array	an applet environment array objects accessible from any context.
global authentication	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.
group context	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.
group-library class loader	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.
heap	<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>
instance variables	also known as non-static fields.

instantiation	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.
instruction	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.
internally visible	items that are not externally visible to other applications on the card. See also <i>externally visible</i> .
inter-application communication facility	see <i>service facility</i> , <i>event notification facility</i> .
JAR file	an acronym for Java Archive file, which is a file format used for aggregating and compressing many files into one.
Java Card Platform Remote Method Invocation	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.
Java Card Runtime Environment (Java Card RE)	consists of the Java Card virtual machine and the associated native methods.
Java Card Virtual Machine (Java Card VM)	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.
Java Card RE context	the context of the Java Card RE has special system privileges so that it can perform operations that are denied to contexts of applications.

Java Card RE entry point object	<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>temporary - 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>permanent - 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>
JDK™ software	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.
local variable	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.
locally accessible web application	an application that may interact with the card holder.
logical channel	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> . (http://www.iso.org).
MAC	an acronym for Message Authentication Code. MAC is an encryption of data for security purposes.
mask production (masking)	refers to embedding the Java Card virtual machine, runtime environment, and applications in the read-only memory of a smart card during manufacture.
method	a procedure or routine associated with one or more classes in object-oriented languages.
mode (communication)	designates the type or protocol of communication (HTTPS, SSL/TLS, SIO...) and the mode of operation (client or server) that characterizes a communication endpoint.

module (application)	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.
multiselectable applets	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.
multiselectd applet	an applet instance that is selected and, therefore, active on more than one logical channel in the APDU application environment simultaneously.
named permission	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.
namespace	a set of names in which all names are unique.
native method	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.
nibble	four bits.
non-card holder-facing client	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.
non-volatile memory	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.
normalization (classic applet)	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 Platform, Version 3, Connected Edition and the Java Card Platform, Version 3, Classic Edition.
normalization (URI)	the process of removing unnecessary "." and ".." segments from the path component of a hierarchical URI.
Normalizer	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.

object-oriented	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.
object owner	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.
object	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.
off-card client	see off-card client application .
off-card client application	an application that is not resident on the card, but runs at the request of a user's actions.
off-card installer	the off-card application that transmits the application and library executables to the card manager application running on the card.
off-card proxy generator	a program or tool used to generate classic SIO synchronization proxies prior to packaging and deploying a classic applet application.
on-card client	see client application .
origin logical channel	the logical channel in the APDU application environment on which an APDU command is issued.
"other user"	a user other than a card holder user, such as a remote card administrator.
owning context	the application or Java Card RE context in which an object is instantiated or created.
owner context	see owning context .
package	a namespace within the Java programming language that can have classes and interfaces.
permission	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.
permission actions list	an attribute of a permission used to designate those actions for which the resources designated by the target name are protected.
permission-based security	measures defined by a permission-based security policy that restrict access to protected system and library resources.

permission-based security policy	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.
permission name	an attribute of a permission used to designate a protected function or resource, or a set thereof.
permission target name	the name attribute of a permission object (permission name) that designates the resource or set of resources that are protected with that permission.
permission type	a type defined by a permission class.
persistent object	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.
PIX	see AID (application identifier) .
platform event	a well-defined event fired by the platform. Examples are clock resynchronization events.
platform protection domain	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.
platform security policy	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.
principal	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.
programmatic security	a means for a security aware application to express the security model of the application when declarative security alone is not sufficient.
protected content	see protected resource .
protected resource	an application or system resource that is protected by an access control mechanism.

protection domain	a set of permissions granted to an application or group of applications.
RAM (random access memory)	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.
reachability disrupting object	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.
reference implementation	a fully functional and compatible implementation of a given technology. It enables developers to build prototypes of applications based on the technology.
remote interface	<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, v3.0.1, Classic Edition</i>.</p>
remote methods	the methods of a remote interface of an applet application.
remote object	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.
remote user	an user whose identity may be assumed by a remote entity, such as a remote card administrator.
remotely accessible web application	an application that is not expected to interact with the card holder but with other-users, potentially remote.
resolution (URI)	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.

resource URI	a URI that uniquely identifies a resource on the platform. Examples are service URI, event URI, application URI and file URI.
RFU	acronym for Reserved for Future Use.
RID	see AID (application identifier) .
RMI	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).
role (development)	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.
role (security)	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.
role-based security	measures defined by a role-based security policy that restrict access by clients or by users to protected application resources.
role-based security policy	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.
ROM (read-only memory)	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.
root URI	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.
runtime descriptor	see descriptor .
runtime environment	see Java Card Runtime Environment (Java Card RE) .
secure port redirector	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.
security constraint	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.

security policy domain	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> .
security policy	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.
security requirements	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.
server application	an on-card application that provides a service to its clients.
service	a shareable interface object that a server application uses to provide a set of well-defined functionalities to its clients.
service facility	a Java Card RE facility (or subsystem) that is used for inter-application communications.
service factory	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.
service registry	the core component of the service facility. The service facility is used for registering and looking up services.
service URI	a URI that uniquely identifies a service provided by a server application.
servlet	a web application component, managed by a container, that generates dynamic web content and that runs in the web application environment.
servlet container	see web application container .
servlet context	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.
servlet definition	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> .
servlet mapping	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> .

session-scoped authentication	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.
shareable interface	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.
shareable interface class loader	the direct child of the bootstrap class loader in the class loader delegation hierarchy in charge of loading publicly exposed shareable interfaces.
shareable interface object (SIO)	an object that implements the shareable interface.
shareable interface object-based service	see <i>service</i> .
smart card	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.
SPI	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.
standard event	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.
standard service	a standard service with a well-defined interface that an application may provide and register. Examples are authenticators - authentication services.
restartable task	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.
restartable task registry	a Java Card RE facility that is used for registering tasks for recurrent execution over card sessions.
terminal	is typically a computer in its own right with an interface which connects with a smart card to exchange and process data.
thread	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.

thread's active context	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> .
transaction	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.
transaction facility	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.
transient object	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.
transferable classes	<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>Implicitly transferable classes - 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 String objects.</p> <p>Explicitly transferable classes - 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 String objects.</p>
transfer of ownership	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.
trusted client	an on-card or off-card application client that an on-card application trusts on the basis of credentials presented by the client.
trusted client credentials	credentials that an on-card application uses to ascertain the identity of clients it trusts.
uniform resource identifier (URI)	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.
uniform resource locator (URL)	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).

user role-based security	see <i>role-based security</i> .
verification	a process performed on an application or library executable that ensures that the binary representation of the application or library is structurally correct.
volatile memory	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.
volatile object	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).
web application	<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>
web application archive	<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>
web application container	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.
web application environment	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.
web client	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, 1-6, 6-19
- access control mechanism, 6-1, GLOSSARY-1
- access permissions, 9-3
- accessing
 - array object methods, 2-32
 - array objects, 2-29
 - class instance object fields, 2-29
 - class instance object methods, 2-29
 - class instance objects, 2-31
 - objects
 - across contexts, 2-20
 - rules, 2-18
 - shareable interface, 2-32
 - shareable interface methods, 2-30
 - standard interface methods, 2-30
 - standard interfaces, 2-31
 - static class fields, 2-28
- active applet instance, GLOSSARY-1
- active context, 2-38, 2-58, 7-15, 7-26
- active namespace, 2-38
- AID
 - applet identification, 2-4, 4-4
 - format, 4-4
- AID (application identifier), GLOSSARY-1
- annotations, APPENDIXB-1
- APDU, 4-1, GLOSSARY-1
 - buffer array, 4-8
 - object, 4-8
 - off-card interactions, 2-3, 4-1
 - scheme of communication, PREFACE-xx
- APDU application, roots of persistence, 2-45
- APDU-based application
 - environment, GLOSSARY-1
- API, GLOSSARY-1
- applet, GLOSSARY-2
 - Classic, PREFACE-xix, PREFACE-xx
 - extended, PREFACE-xix, PREFACE-xx
 - firewall, 2-17
- applet application, GLOSSARY-2
 - communication, 7-1
 - composition, 2-4
 - creation, 4-5, 4-9
 - deletion, 4-6
 - deployment descriptor, 8-15
 - module, 4-2
 - module unloading, 4-6
 - selection, 4-10
 - Shareable Interface Object, 7-7
 - standalone, 4-2
 - two types, 4-2
- applet container, 1-4, 4-1, GLOSSARY-2
- applet framework, GLOSSARY-2
- applicable credential manager, GLOSSARY-2
- applicable security requirements, GLOSSARY-2
- application
 - APDU-based, 4-1
 - applet, 2-4
 - applet-based, PREFACE-xx
 - effective protection domain, 6-13
 - extended applet, 2-53
 - locally accessible, 6-28
 - remotely accessible, 6-28
 - URI, 9-3

- web, PREFACE-xx
- application assembler, GLOSSARY-2
- application authentication, client, 6-46
- application descriptor, GLOSSARY-2
 - Java Card Platform-specific, 8-11
- application developer, GLOSSARY-2
- application firewall, GLOSSARY-2
- application framework class loader, GLOSSARY-2
- application group, 7-1, GLOSSARY-2
- Application Identifier, 4-4
- application identifiers, 2-4
- application instance
 - creation, 8-31
 - deletion, 8-33
 - use of term, 8-31
- application model, 2-1
- application module
 - application distribution, 8-2
 - code isolation, 8-27
 - deployment units, 8-25
 - loading, 3-3
- application module class loader, GLOSSARY-3
- application owner identifier, 2-38
- application programming, 2-1
- application protection domain, GLOSSARY-3
- Application Protocol Data Unit, 4-1
- application security policy, 6-3, GLOSSARY-3
- application URI, 2-5, 3-4, GLOSSARY-3
- application-defined event, GLOSSARY-2
- application-defined service, GLOSSARY-2
- application-managed authentication, GLOSSARY-3
- arrays
 - accessing object methods, 2-32
 - global, 2-24
 - objects, accessing, 2-29
- atomic operation, GLOSSARY-3
- atomicity, 2-46, GLOSSARY-3
- atomicity of operations, 9-4
- authentication, GLOSSARY-3
 - duration, 6-48
 - global, 6-29, 6-32
 - session-scoped, 6-29
 - web container-managed, 6-35
- authenticator, 6-30, GLOSSARY-3
- authorization, GLOSSARY-3

- card holder, 6-45

B

- backward compatibility, 4-16, 7-14
- basic logical channel, GLOSSARY-3
- bootstrap class loader, GLOSSARY-3
- bytecode, GLOSSARY-3

C

- canonicalization (URI), GLOSSARY-3
- CAP, 8-7
- card
 - management, 8-1
 - reset, 5-2
- card holder, 6-28, GLOSSARY-4
- card holder user, GLOSSARY-4
- card holder-facing client, GLOSSARY-4
- card initialization, 5-1
- card management application, default protection domain, APPENDIXA-7
- card management facility, 8-2, 8-25, 8-34, GLOSSARY-4
- card management security policy, GLOSSARY-4
- card manager, GLOSSARY-4
 - deployment, 8-3
 - deployment units, 8-25
 - description, 8-1
 - distribution, 8-3
- card session, GLOSSARY-4
- class
 - access, 6-63
 - access behavior, 2-28
 - explicitly transferable, 7-3
 - implicitly transferable, 7-3
 - transferable, 7-3
- class dependency, 8-27
- class file lookup, 8-27
- class holder delegation hierarchy, 6-58
- class loader, 7-1, GLOSSARY-5
- class loader delegation hierarchy, 7-1, GLOSSARY-5
- class loading, 6-60
- classic applet, GLOSSARY-4
 - default protection domain, APPENDIXA-6
 - definition, PREFACE-xx
 - SIO integration, 7-13

- classic applet container mutex object, GLOSSARY-4
- classic application
 - application support, 4-16
 - transaction model, 4-23
- Classic Edition, PREFACE-xix, GLOSSARY-4
- classic libraries, 8-29
- classic library, GLOSSARY-4
- classic library class loader, GLOSSARY-4
- classic SIO proxy, GLOSSARY-4
- client-role-based security, GLOSSARY-5
- clock, 2-37
- code isolation, 6-57, 7-1, 7-2
- communication, asynchronous, 7-16
- concurrency, 2-53, 5-5
- concurrent processing, 4-13
- Connected Edition, PREFACE-xix, GLOSSARY-5
- connection endpoint, GLOSSARY-5
- connection endpoint (client, server), GLOSSARY-5
- connectivity, 1-1
- contacted interface, 1-2
- contactless interface, 1-2
- container, default, 3-19
- container-managed authentication, GLOSSARY-5
- container-managed endpoints, 2-2
- container-managed object, GLOSSARY-5
- context
 - active, 2-38
 - isolation, 6-65
 - switching, 6-65
- context isolation, 7-1, 7-2
- context path, 2-5, 3-4, GLOSSARY-5
- context switch, GLOSSARY-6
- contexts, 2-26
 - currently active, 2-16
 - Java Card RE, 2-16
 - object accessing across, 2-20
 - rules in firewall, 2-17
 - switching in the VM, 2-17
- Converter, 2-50
- converter, GLOSSARY-6
- credential, GLOSSARY-6
- credential management, 6-49
- credential manager, 6-50, GLOSSARY-6
- currently active context, GLOSSARY-6

- currently active namespace, GLOSSARY-6
- currently selected applet, GLOSSARY-6

D

- declarative security, GLOSSARY-6
- dedicated namespace, 6-65
- default applet, GLOSSARY-6
- default default servlet, 3-20, GLOSSARY-6
- default servlet, GLOSSARY-6
- defensive copy, 7-6
- deployer, GLOSSARY-7
- deployment descriptor, 2-2, GLOSSARY-7
- deployment process, 1-6
- deployment unit, unloading, 8-36
- descriptor format, 8-10
- descriptor, web application deployment, 8-15
- development process, 1-6
- distribution format, 8-4, GLOSSARY-7
- distribution unit, GLOSSARY-7
- dynamic content, 2-2

E

- EEPROM, GLOSSARY-7
- endpoint
 - application-managed server, 2-34
 - container-managed, 2-2, 2-34
 - server connection, 2-2
- entry point, 3-14
- entry point method, 2-35, 2-36, 4-9, 4-12, GLOSSARY-7
- entry point objects, 1-6, 7-2
- errors, 3-4, 3-6, 3-14
- event
 - definition, 7-18
 - identification, 7-18
 - listener, 7-25
 - notification, 7-23
 - security, 7-24
 - URI, 7-24
- event consuming application, GLOSSARY-7
- event listener
 - description, 2-34
 - lifetime, 7-25
 - persistence, 7-25
 - registration, 7-22

- thread safety, 7-25
- unregistration, 7-23
- event notification facility, 7-16
- event notification listener, 7-22, GLOSSARY-8
- event notification, file resource, 9-5
- event producing application, GLOSSARY-8
- event registry, 7-24, GLOSSARY-8
- event URI, 7-18, GLOSSARY-8
- exception objects, 2-31
- execution lifetime, 5-1
- export file, GLOSSARY-8
- extended applet, GLOSSARY-8
 - default protection domain, APPENDIXA-4
 - definition, PREFACE-xx
- extended applet application
 - card manager, 8-1
 - multithreading, 2-4
 - Shareable Interface Object, 7-7
 - thread behavior, 2-53
- extension libraries, 8-29
- extension library, GLOSSARY-8
- extension library distribution format, 8-8

F

- fields
 - accessing class instance object, 2-29
 - accessing static class, 2-28
 - static, 2-19
- file access, 3-2
- file access permissions, 9-3
- file resource event notification, 9-5
- file system
 - object, 9-2, 9-3
 - requirements, 9-1
- file URI, 9-2
- filter, 3-1
- filter mapping, 3-2
- firewall, 1-6, 2-14

G

- garbage collection, 2-43
- GCF, 3-2, 3-8, 9-5
- Generic Connection Framework, 3-2, 9-5
- global array objects, 1-6
- global arrays, 2-24

- group context, 2-38, 3-27, 6-17, 7-1, 7-15

H

- hosting, 3-21

I

- identity, 6-28
- initialization sequence, 5-1
- inter-application communication, 2-4, 7-1
- inter-application communication facility, 2-33, 7-7
- interfaces
 - accessing shareable, 2-32
 - accessing shareable methods, 2-30
 - accessing standard, 2-31
 - accessing standard methods, 2-30
 - shareable, 2-25, 2-26
- ISO 7816-4, 4-1

J

- Java Card RE
 - entry point objects, 2-20
 - privileges, 2-25
- Java EE, 3-1
- Java Platform, Enterprise Edition, 3-1
- JavaServer Pages, 3-1

L

- library, loading, 8-29
- lifecycle
 - applet application, 4-1, 4-2
 - applet methods, 4-9
 - servlet, filter, and listener methods, 3-14
- lifecycle event dispatch, 3-11
- lifetime, execution, 5-1
- listener
 - blocking, 7-26
 - event, 2-2
 - unresponsive, 7-26
- logical channel, 4-7

M

- MAY, definition, PREFACE-xx
- memory
 - non-volatile, 2-42
 - volatile, 2-42
- memory store, 2-42

- methods
 - accessing
 - array object, 2-32
 - class instance object, 2-29
 - shareable interface, 2-30
 - standard interface, 2-30
 - static, 2-19
- multithreaded, 2-4, 2-35, 3-17, 4-13
- MUST NOT, definition, PREFACE-xix
- MUST, definition, PREFACE-xix

N

- namespace
 - active, 2-38
 - code isolation, 7-1
 - enforcement, 6-65
 - event, 7-18
 - service, 7-9
 - unified naming, 2-5
- network communications, 6-49
- non-volatile memory, 2-42
- normalization, 1-6
- Normalizer, 2-50, GLOSSARY-12

O

- object
 - copies, 7-6
 - event, 7-16
 - ownership transfer, 7-3
 - persistent, 2-43, 3-16
 - reachability disrupting, 2-43
 - special, 2-44
 - transferable, 6-66
 - transient array, 2-44
 - transient reference, 2-44
 - volatile, 3-16
- object ownership transfer, 6-66
- object ownership transfer mechanism, 7-2
- object store, 2-42
- objects
 - access behavior, 2-28
 - accessing
 - across contexts, 2-20
 - array, 2-29
 - array methods, 2-32
 - class instance, 2-31
 - class instance fields, 2-29

- class instance methods, 2-29
- rules, 2-18
- Java Card RE entry point, 2-20
- ownership, 2-17
- sharing, 2-20
- throwing exception, 2-31
- transient
 - CLEAR_ON_DESELECT, 2-19
 - CLEAR_ON_RESET, 2-19
 - contexts, 2-19
- off-card compatibility, 1-6
- on-card
 - application, 8-1
 - authentication, 6-46
 - initialization, 1-6
- other user, 6-28
- owner context, 2-38
- ownership tagging, 1-6

P

- package access control, 6-63
- package sealing, 6-64
- permission types, 6-12
- permissions
 - assigning, 6-17
 - checking, 6-17
 - named, 6-5
 - new, 6-11
 - URI-named, 6-7
- persistence, 1-6, 2-4, 2-42, 3-16
- persistence roots, 2-44
- persistent object, 2-43
- PIX, 2-5, 4-4
- platform protection domain, APPENDIXA-1
- platform reset, 3-9, 9-6
- platform security policy, APPENDIXA-1
- portability, 3-5, 4-5
- ports, 3-21
- power down, 5-2
- power-up, 5-1
- proprietary identifier extension, 2-5, 4-4
- protected content, 3-22, GLOSSARY-14
- protection domain
 - application, 6-13
 - definition, 6-3, 6-13
 - in access control decision, 6-19

platform, 6-16, APPENDIXA-1

R

reachability disrupting object, 2-43

realm, GLOSSARY-17

request

dispatching, 3-23

redirection, 3-23

request dispatching, 3-9

reset

card, 5-2

I/O interface, 5-3

TCP, 5-4

resource identifier, 2-5, 4-4

restart

applet application, 4-7

on platform reset, 3-9

restartable task, 2-4

restartable task facility, 2-55

RID, 2-5, 4-4

roots of persistence, 2-44

runtime descriptor

application module loading, 3-3

attributes, 8-20

elements, 8-17

runtime emulation environment, 1-5

runtime environment, single threaded, 4-22

S

secure communications, 6-49

secure connection, 3-28

security

annotations, APPENDIXB-1

checks, 6-20

constraint, 6-3

containment mechanism, 7-1

mechanism, 6-1

permissions, 6-4

platform, APPENDIXA-1

restrictions, 4-23

role-based, 6-23, 7-14

secure port, 3-6, 3-21, 3-22

using URL patterns, 3-2

security policy

application, 6-3

card management, 6-3

definition, 6-1

enforcement, 6-19

permission-based, 6-2

platform, 6-3

role-based, 6-3

service

lookup, 7-13

platform, 7-9

SIO-based, 7-8, 7-14

standard, 7-9

URI, 7-9, 7-11, 7-12

service factory, 7-15

lifetime, 7-15

registration, 7-11

service lookup, 7-13

unregistration, 7-12

service namespace

root, 7-9

service URI, 7-9

service registry, 7-8, 7-11, 7-13

servlet, 2-1

API, 1-5

container, 1-5

context, 3-2

default, 3-19

support for, PREFACE-xix

session lifetime, 3-13

sharable interface, 2-33

shareable interface methods, 4-12

shareable interface object, 7-7

shareable interface object-based services, 2-33

Shareable Interface Objects

See SIOs

shareable interfaces

See interfaces, shareable

SHOULD NOT, definition, PREFACE-xx

SHOULD, definition, PREFACE-xx

SIO, 2-33, 3-16

SIO proxy, 4-17

SIO synchronization proxy, 4-18

SIO-based service

lifetime, 7-15

persistence, 7-15

registering, 7-7

SIOs

definition, 2-25

- obtaining, 2-27
- sharing objects, 2-20
- smart card, 1-1
- SSL, 3-27, 6-49
- SSL certificate, 3-1
- standalone application module distribution
 - format, 8-4
- standalone extended applet application module
 - distribution format, 8-6
- standards bodies, 1-4
- static
 - accessing class fields, 2-28
 - fields, 2-19
 - methods, 2-19
- static content, 2-2
- system clock, 2-37

T

- task
 - execution, 2-56
 - lifetime, 2-57
 - persistence, 2-57
 - recurrent, 2-56
 - registration, 2-56
 - registry, 2-56
 - restartable, 2-55
 - unregister, 2-57
 - unregistration, 2-57
- tear, 5-2
- thread, 1-6, 2-53, 2-58, 3-17, 4-13, 7-15, 9-6
 - execution, 4-17
 - ownership, 2-57, 3-19, 4-15
 - safety, 2-57
 - web application, 2-53
- thread creation, 2-36
- thread safety, 3-18, 4-14, 7-15, 7-25, 9-6
- thread's active context, GLOSSARY-19
- TLS, 6-49
- transaction, 1-6, 2-54
 - aborting, 2-54
- transaction demarcation, 2-47
- transaction facility, 2-45
- transaction tag, 2-48
- transaction type, 2-47
- transactional behavior, 9-4
- transactions, 2-4

- transferable classes, 7-3
- transient array object, 2-44, 2-54

U

- unified naming, 2-5
- unload
 - applet application module, 4-6
- URI, 2-5, 3-4
 - absolute, 2-8
 - application, 9-3
 - file, 9-2
 - relative, 2-8
 - resource event, 9-5
- user
 - authentication, 3-1, 6-28
 - authorization, 6-28
 - identity, 6-28

V

- volatile memory, 2-42
- volatile object, 3-16

W

- web application, PREFACE-xx, 2-53, 3-1
 - access, 6-41
 - card manager, 8-1
 - client, 6-40
 - communication, 7-1
 - context path, 3-4
 - creation, 3-5
 - credentials, 3-26
 - default protection domain, APPENDIX A-2
 - deletion, 3-7
 - deployment descriptor, 8-15
 - entry points, 3-16
 - environment, 3-1
 - group context, 3-27
 - hosting, 3-21
 - lifecycle, 3-2
 - module, 3-2
 - module unloading, 3-8
 - naming, 3-4
 - roots of persistence, 2-45
 - security, 3-20
 - Shareable Interface Object, 7-7
- web application deployment descriptor, 8-15
- web client, 3-1

web container, 3-1, 3-9, 3-14
port allocation, 3-23

X

XML format, 8-10

XML schema, 8-15