Application Programming Notes

Java Card™ 3 Platform, Version 3.0.2

Connected Edition

April 2010

Please
Recycle

Contents

Figures

Preface

This book contains tips and guidelines for developers using the Java Card™ 3 Platform, Connected Edition, to create applications and for developers of vendor-specific frameworks. The Java Card 3 Specification currently includes releases 3.0 and 3.0.1. The Java Card 3 Platform Development Kit currently includes releases 3.0.1 and 3.0.2.

This book covers several programming features that are of interest because they differ from those found in earlier versions of the Java Card platform. This book is not meant to comprehensively introduce or cover general programming topics.

These programming notes are not a tutorial on the Java Card 3 Platform. They complement the tutorials and programming notes for the different Java technologies that are integrated in the Java Card 3 Platform. The reader should refer to those tutorials and programming notes for other programming guidelines and best practices that may apply.

Who Should Use This Book

This book is for developers using the *Application Programming Interface, Java Card Platform, Version 3.0.1, Connected Edition* to implement web applications and applets with extended and enhanced capabilities for the Java Card platform.

This book is also for developers who are considering creating a vendor-specific framework based on version 3.0.1 of the Java Card 3 platform specifications, Connected Edition.

If you are unfamiliar with the Java Card 3 Platform, you might want to see Chapter 1, otherwise you can go directly to Chapter 2.

Before You Read This Book

Before reading this guide, become familiar with the Java programming language, object-oriented design, the Java Card technology specifications, and smart card technology. A good resource for becoming familiar with Java and Java Card technology is the web site, located at

`http://java.sun.com`

How This Book Is Organized

Chapter 1, Java Card 3 Platform Introduction, provides an introduction to the Java Card 3 Platform.

Chapter 2, Programming Notes Introduction, provides an introduction to this book and to the sample used throughout, the Transit sample reference application.

Chapter 3, Authenticating and Authorizing Application Users, describes issues related to programming and deploying handle user authentication on the Java Card 3 Platform, Connected Edition.

Chapter 4, Authenticating and Authorizing On-Card Applications, describes issues related to authenticating and authorizing applications that are on the smart card.

Chapter 5, Securing Connections End-to-end, describes how to handle the security constraints on the Java Card 3 Platform, Connected Edition.

Chapter 6, Internationalizing and Localizing Applications, describes how to program and deploy applications that are ready for internationalization and localization.

Chapter 7, Migrating Classic Applet Applications, describes how to migrate a classic applet application to become an extended applet application for the extended Connected environment.

Chapter 8, Packaging, Distributing and Deploying Applications, describes how to handle packaging and descriptors on the Java Card 3 Platform, Connected Edition.

Glossary lists terms used throughout the Connected Edition.

Related Books

References to various documents or products are made in this book. Version 3.0.1of the Java Card specifications for the Connected Edition can be downloaded at

`http://java.sun.com/javacard`

- *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Connected Edition*
- *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition*
- *Application Programming Interface, Java Card Platform, Version 3.0.1, Connected Edition*
- *Development Kit User's Guide, Java Card Platform, Version 3.0.2, Connected Edition* can be downloaded with the Development Kit for the Connected Edition.
- *Java Card Technology for Smart Cards* by Zhiqun Chen (Addison-Wesley, 2000).
- *The Java Programming Language (Java Series)*, Second Edition by Ken Arnold and James Gosling (Addison-Wesley, 1998).
- *The Java Virtual Machine Specification (Java Series)*, Second Edition by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999).
- *The Java Class Libraries: An Annotated Reference*, Second Edition (Java Series) by Patrick Chan, Rosanna Lee and Doug Kramer (Addison-Wesley, 1999).
- *ISO 7816 Specification* Parts 1-6.

Typographic Conventions

The following table lists the typographic conventions used in this book.

TABLE P-1 Typographic Conventions For This Book

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su`
		`Password:`
		1. **Run `cref` in a new window.**

AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Please Send Your Comments

Oracle is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments to `jc3-ri-feedback@sun.com`.

Please include the title of your document with your feedback:

Application Programming Notes, Java Card 3 Platform, Version 3.0.2, Connected Edition

CHAPTER 1

Java Card 3 Platform Introduction

This introduction includes details on the Java Card 3 Platform, especially the Connected Edition, and is intended for readers not yet familiar with the Java Card 3 Platform. For precise information on each of these features, readers should, nevertheless, refer to the relevant specifications. If you are already familiar with the Java Card 3 Platform, or for more information on the organization and uses of this book, please see the Chapter 2, Programming Notes Introduction.

1.1 Java Card 3 Platform Editions

This section gives an overview of the Java Card 3 technology with most information on the Connected Edition, comparing it to the Classic Edition only when relevant.

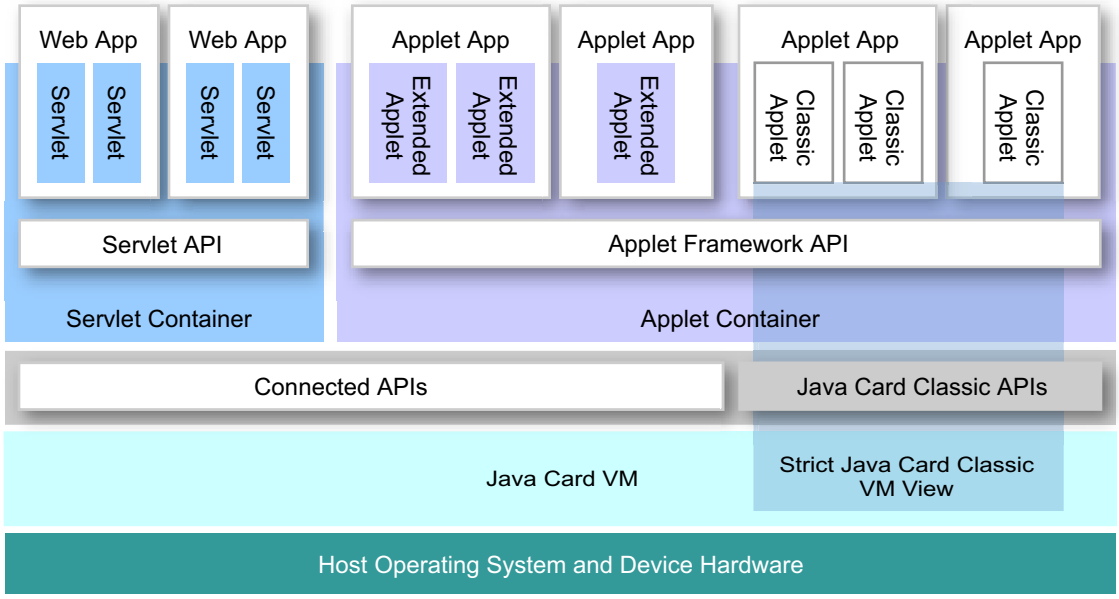
The Java Card technology is available in two separate, yet coherent editions:

- Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and targets more resource-constrained devices that support traditional applet-based applications. It introduces several incremental changes to the previous version to ensure alignment with smart card and security standards.
- Connected Edition features a significantly enhanced execution environment and a new virtual machine. It includes new network-oriented features, such as support for web applications, and support for applets with extended and advanced capabilities. It targets the high-end smart card hardware coming from the latest advances in silicon technology.

Both Editions are compatible with applications written for previous versions. They also share the key security features and build on the trust and expertise derived from ten years of deploying secure Java Card technology products.

FIGURE 1-1 shows the high-level architecture of the Connected Edition.

FIGURE 1-1 High-level Architecture of the Connected Edition



The Connected Edition of the Java Card virtual machine is based on the Connected Limited Device Configuration (CLDC) virtual machine widely used in mobile phones. As such, it is able to support some of the richer features of the Java programming language. The CLDC-based virtual machine has also been enhanced to meet the requirements of a security device environment: it has been reduced in size, and support for smart card protocols and security has been added. It is the cornerstone of a new architecture that can provide a new level of functionality and convenience to developers.

The Connected Edition architecture enables the simultaneous deployment of web applications and traditional smart card applications. On-card web applications provide services to entities on an IP network. They are accessible using standard Internet protocols, such as HTTP/HTTPs, and, therefore, can be integrated easily into existing Internet services infrastructures. Traditional smart card applications leverage standard card protocols such as ISO 7816 or contactless protocols. They are required for integration into existing smart card infrastructures.

1.1.1 Hardware Platform and Virtual Machine

The Java Card Virtual Machine is based on the *Connected Limited Device Configuration Specification, Version 1.1* standard defined by the Java™ Platform, Micro Edition (Java™ ME) Platform. The technology defined in the Java Card specification is suitable for an advanced smart card device. Smart card devices are typically much more resource-constrained than Java ME platform devices.

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

1.1.1.1 Typical Hardware Configuration and Connectivity

The Java Card 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 for the Connected Edition corresponds to high-end hardware when compared to devices targeted by the Classic Edition. It typically has a faster processor and more volatile (RAM) and persistent (EEPROM and ROM) memory. A key characteristic of this high-end hardware is the support of a full duplex, high-speed interface with its hosting device (such as a phone) and connectivity to some kind of network, typically through its hosting device.

The target devices for the Connected Edition typically have a high-speed contacted physical interface such as a USB interface. In addition, they may have additional I/O interfaces, including contactless physical interfaces, typically ISO 14443 compliant. The Java Card platform provides its applications with a logical network interface supporting IP-based protocols such as TCP, TLS, HTTP and HTTPS.

TABLE 1-1 compares the smart card hardware targeted by the Connected Edition of the Java Card technology with the traditional hardware targeted by previous releases of the Java Card technology and by the Classic Edition.

TABLE 1-1 Evolution of Java Card Technology Targeted Device Configurations

Traditional Smart Card Hardware	High-end Smart Card Hardware
8/16 bit CPU	32 bit CPU
~2Kb RAM	24Kb RAM

TABLE 1-1 Evolution of Java Card Technology Targeted Device Configurations

Traditional Smart Card Hardware	High-end Smart Card Hardware
48-64Kb ROM	>256Kb ROM
8-32Kb EEPROM	>128Kb EEPROM
Serial I/O interface	High-speed interfaces
9.6-30 Kb/s	1.5 Mb/s-12 Mb/s
Half duplex	Full duplex

1.1.1.2 Virtual Machine Technology

The Connected Edition of the Java Card virtual machine is based on the CLDC virtual machine, version 1.1. It has been enhanced to meet the requirements of a security device environment: it has been reduced in size, and support for smart card protocols and security has been added. It has also been extended to follow some of the recent advancements of Java™ Platform, Standard Edition (Java™ SE technology) for ease of use and developer friendliness.

The virtual machine technology in the Connected Edition is intended for a 32-bit CPU on a high-end smart card device, while the virtual machine technology in the Classic Edition is suitable for an 8- or 16-bit CPU on a more resource-constrained, traditional smart card device. The virtual machine of the Connected Edition is able to load class files directly, whereas the Classic Edition uses the traditional split-VM technique whereby the loading, linking and verification functions of the Java virtual machine[1] are performed by the off-card Converter tool, which then produces an optimized CAP file format suitable for use on the card by the on-card virtual machine.

The Connected Edition and the Classic Edition feature persistent virtual machines with persistent objects. Multiple applications securely execute on the virtual machine while their objects are isolated from intrusion by a firewall-based context isolation mechanism. The Transaction facility supported by the virtual machines allows an application to maintain a consistent state across card tears and power losses.

The Connected Edition includes the following Java technology features beyond that of the Classic Edition:

- **Multi-threading**: Multiple application threads may execute concurrently to process off-card messages. Applications may start background threads.
- **Class file loading and On-card bytecode verification**: The class files of the application are verified for type safety by the virtual machine on the card. The Connected Edition leverages the stack map attribute information in the class files

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

generated by the Java SE Development Kit (JDK™ software) 1.6 compiler to perform the verification of the application code efficiently using the limited amount of volatile memory available on the card.

- **Automatic Garbage Collection**: Temporary session data is automatically garbage collected when no longer in use.

The core platform libraries in the Connected Edition include the system classes that support these new VM features and its networking capabilities. In addition, they have been extended to support collection and other utility classes to provide a more developer-friendly programming environment. These core platform libraries include the following:

- **Data types**: `char`, `long` and `String`
- **Multi-dimensional array data type**
- **Primitive wrapper classes**: `Boolean`, `Integer` and more
- **String manipulation classes**: `StringTokenizer`, `StringBuffer`, `StringBuilder`
- **Multithreading support classes**: `Thread` and more
- **I/O classes**: `Reader`, `Writer` and `Stream` classes
- **Networking classes from the Generic Connection Framework**: `Connector`, `Connection` and more
- **Collection classes**: `Vector`, `Hashtable`, `Stack`, `Iterator` and more
- **Date and time utility classes**: `Calendar`, `Date`, `TimeZone`
- **Localization and Internationalization support classes**: `Locale`, `ResourceBundle` and more

The Java programming language support in the Connected Edition has been enhanced to provide the advanced features of Java SE technology. These features include:

- **Generics**: Support for *generics* allows code to be reused with varying data types with full type safety.
- **Metadata (Annotation)**: The Connected Edition uses the *Annotation* feature to demarcate transaction protected zones within the code. Annotation uses the `SOURCE` and `CLASS` annotation retention policy directive.
- **Assertions**: This feature enables programmers to test their assumptions about the runtime program state.
- **Enhanced for loop**: This feature allows a program to conveniently iterate over elements of an aggregator object.
- **Auto boxing**: This feature allows a program to use a primitive type as a first class object.
- **Typesafe Enums (Enumerations)**: This Java programing language feature provides a type safe mechanism to define a set of named values for a type.

- **Varargs**: This facility allows a programmer to define methods that accept a variable number of arguments.
- **Static Import**: This facility helps the programmer avoid the "Constant Interface antipattern" problem.

1.2 Application Programming Models

The Java Card Platform, Connected Edition brings the application programming experience closer to that of mainstream Java programming. It not only supports a virtual machine with a richer subset of the Java virtual machine features, such as multithreading and richer data types, but also provides a web application programming environment. Moreover, the applet application model has been extended to benefit from the newly added features, such as multithreading and network connectivity, thereby providing a migration path from classic applet application programming to web application programming. A Java Card platform application developer may also use many of the new or enhanced facilities provided by the platform, such as persistence, transactions, inter-application communication and restartable tasks facilities.

1.2.1 Web Applications

The Connected Edition supports a subset of the *Java™ Servlet API Specification* v2.4 web application model.

Web applications are applications that interact with off-card web clients via HTTP or HTTPS requests and responses. The lifecycle of web applications, as well as the network services over which requests and responses are sent, and the security of access to these applications and their resources, are managed by the Java Card platform's web application container.

A Java Card technology developer typically implements the following components of a web application:

- **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.
- **Static resources such as HTML documents and embedded images or objects**: Static resources constitute static content, which is content that resides in files or the equivalent.

- **Web application deployment descriptor**: The deployment descriptor describes the web application's components and how they are mapped to client requests and their security requirements (which include user authentication and authorization, as well as secure communication requirements).

Each web application is deployed into the web container and is uniquely identified by the specific path at which it is rooted in the web container namespace. That path is used both as the application's unique identifier (application's Uniform Resource Locator (URI)) on the platform, such as for application management and on-card inter-application communications, as well as for dispatching HTTP requests submitted by off-card web clients, based on the queried URL.

The web container is multithreaded and is able to dispatch multiple HTTP requests concurrently.

1.2.2 Applet Applications

The Connected Edition supports both the classic applet application model and the extended applet application model.

Applet applications interact with off-card applet clients via ISO 7816-4 APDU commands and responses. The lifecycle of applet applications is managed by the Java Card platform's applet container[2].

Extended applet applications differ in the following respects from classic applet applications as they are supported in the Classic Edition:

- extended applet application code may comprise multiple packages, while classic applet application code comprises only one package
- extended applet application code may use all facilities and libraries of the Connected Edition, while classic applet applications are restricted to those defined in the Classic Edition.
- extended applet applications may execute concurrently on different threads to process APDU commands received over different I/O interfaces, while classic applet applications execute in a single threaded environment and are not thread aware.

A Java Card platform developer typically implements the following components of an applet application:

- **Applets 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.

2. The concept of application container, which is implicit in the Classic Edition, has been formalized in the Connected Edition to better support the different application models.

- **Applet application deployment descriptor**: The deployment descriptor contains information to uniquely identify each concrete applet class in the application.

Each applet application is deployed into the applet container and is uniquely identified on the card by its applet instance Application Identifier (AID). An applet instance AID has an equivalent URI form (application URI) that is used both as the application's unique identifier on the platform for application management and on-card inter-application communications, as well as for applet selection and dispatching of APDU commands submitted by off-card clients.

1.2.3 Multithreading

The Java Card virtual machine, Connected Edition, supports multithreading and concurrent execution of applications. The Java Card API includes a subset of the Java SE technology thread API, which allows an application to create and handle threads of control.

As described in the previous sections, the web application environment and the extended applet application environment are multithreaded application programming environments. Other facilities of the Java Card Runtime Environment (Java Card RE) may also use different threads to concurrently invoke application entry point methods, such as to asynchronously notify an application of an event. A developer of such applications must account explicitly for multithreading and may even rely on application-managed threads to implement applications.

Threads are not persistent and cannot be resumed after a platform reset. Application-managed threads must be "manually" recreated after each platform reset. To alleviate this issue, the Connected Edition provides a facility for an application to register tasks that are automatically restarted upon a platform reset.

The Connected Edition supports a classic applet application programming environment for backward compatibility with the Classic Edition, which does not support multithreading, but still operates concurrently with the other two application environments.

1.2.4 Persistence

Code and data persistence across card restart events such as card tear (or reset) followed by 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 restart events. Objects may be made persistent as well, under certain conditions.

On the Java Card platform, the memory is divided into volatile memory and non-volatile memory:

- **Volatile memory** (typically DRAM) does not retain its contents across card restart events on the smart card device. *Volatile objects* are objects stored in volatile store and are typically intended to be short-lived or to require frequent updates. Volatile objects are garbage collected on card restart events.
- **Non-volatile memory** (typically ROM, EEPROM and Flash memory) retains its contents across card restart events on the smart card device. Persistent objects are objects stored in non-volatile store and are intended to be long-lived objects. Persistent objects retain their contents across card restart events.

While the persistence of an application's code (its classes) and of some of its data (its objects), such as is required for the management of the application, is ensured by the platform, application-created objects are all initially volatile and may only be made persistent under certain conditions of reachability. The Java Card platform implements a strategy, *persistence by reachability*, for promoting volatile objects to become persistent objects. A newly created object remains volatile as long as it is not referenced by any other persistent object, otherwise it becomes persistent.

All objects, both volatile objects and persistent objects, are garbage collected when no longer referenced by 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 by application code.

1.2.5 Transactions

On the Java Card platform, an application may complete a single logical operation on application data atomically, consistently, and durably within a transaction. The transaction facility of the Java Card platform 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 transaction facility of the Connected Edition extends the "Transactions and Atomicity" subsystem of the Classic Edition to provide the following additional features:

- **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.
- **Better programmer control and program audit of transaction durations**: A method is annotated to explicitly declare its transactional behavior.

The Connected Edition uses annotations (a subset of Java SE technology annotations) to mark classes or methods for transaction demarcation.

1.2.6 Inter-application Communications

In the Connected Edition, inter-application communications have been enhanced beyond the classic *Shareable Interface mechanism* to allow for an application to communicate with another application using the following two facilities:

- **Services**: An application can publish in a central *registry* a *service* it wants to provide to other applications. This facility extends the classic shareable interface mechanism and allows for all application models (applet and web applications) to interact through shareable interface objects in a unified way. In addition to services that may be defined by applications, the specifications for the Connected Edition include a set of predefined standard services, such as for user authentication.
- **Events**: The platform or an application can notify through a central *registry* other applications of a particular condition. When the condition occurs, it is encapsulated in a shareable interface object called an *event* and is passed for handling to an object called an *event listener* that has been registered for notification of the condition. This facility allows for web and applet applications to communicate asynchronously with each other through events. In addition to application-defined events, the specifications for the Connected Edition include a set of predefined platform and standard events, such as for clock resynchronization, application lifecycle and resource lifecycle management.

The use of central registries for services and events allows for a loose coupling between server and client applications. This, and support for application lifecycle events, allows for provisioning of client and server applications independently of each other throughout the lifetime of a card.

1.2.7 Network Communications

In addition to the networking capability intrinsically offered by the web application container (*container-managed connections*), the Java Card platform provides applications with a means for managing network communications by themselves. An application may not only open *server communication endpoints* but may also initiate *client communications* with off-card entities. Thus, applications on the Java Card platform are no longer limited to providing services to off-card clients, but may also themselves be clients of off-card services across a network.

Network connections, both client and server connections, can be managed on the Java Card platform using the Generic Connection Framework. The Generic Connection Framework classes provide a set of related abstractions to request and manage network or I/O connections using various protocols, including both secure and non-secure protocols such as TCP, TLS, HTTP, HTTPS and, optionally, UDP.

1.2.8 File Access

The Java Card platform has optional support for hierarchical file systems of directories and files. An application may use the Generic Connection Framework to access file system objects.

Each application may have a dedicated file system to which it has exclusive access. An application may nevertheless be implemented to act as a file server to other applications. A typical file server application stores the file contents in its own private file system and may implement some access control policy using the security mechanism provided by the platform, such as user authentication and client authentication, to permit or deny access by client applications to certain files.

1.3 Security Model

Beyond the low-level security implemented by the Java Card virtual machine, namely Java programming language bytecode verification, the Connected Edition implements a variety of security mechanisms that provide application-level security and also end-to-end communication security.

1.3.1 Secure Containment of Applications

The Connected Edition of the Java Card technology provides several complementary mechanisms that enforce the security containment of applications.

1.3.1.1 Code Isolation and Java Programming Language Package Access Control

The Java Card platform supports a *code isolation* mechanism. Code isolation ensures that code loaded from one application does not interfere with the code of other applications. Code loaded in this manner cannot override or directly access the code of other applications. This is implemented by defining and enforcing different class namespaces for each loaded application's code.

On the Java Card platform, code isolation is 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. The class loader delegation hierarchy is not exposed to application developers and is strictly defined by the platform.

Additionally, the Java Card platform includes mechanisms to prevent loaded code, applications or libraries, from overriding or extending the set of system classes. It also supports the standard *JAR file package sealing* mechanism, which prevents the classes in a sealed package from being overridden or extended once that package has been loaded.

These mechanisms cater to the secure containment of application code.

1.3.1.2 Context Isolation and Object Sharing

The Java Card platform supports isolation of contexts and applications. *Context isolation* ensures that objects created, and therefore owned, by applications running in the same context cannot be accessed by applications from another context unless the applications owning these objects explicitly provide interfaces for access. Such interfaces are called *Shareable Interfaces* and objects implementing these interfaces, *Shareable Interface Objects*, constitute legal entry points to these applications. Similarly, the Java Card RE executes in a separate context and provides well-defined entry points, *Java Card RE Entry Point Objects*, that can be used by applications to request system services.

Context isolation is enforced by an *application firewall* and protects against any unauthorized access, such as those that may result from developer mistakes and design oversights.

The firewall-enforced context isolation mechanism originally defined in the Classic Edition has been enhanced in the Connected Edition to account for the extended bytecode set and to allow for more effective inter-application communication through Shareable Interfaces. The Connected Edition introduces a new mechanism, the *object ownership transfer mechanism*, which allows for an application to transfer the ownership of an object it owns to another application, thereby transferring exclusive access to the object to that other application.

Context isolation guarantees the secure containment of application execution.

1.3.1.3 Dedicated Application Namespaces

The Connected Edition supports a *Unified Naming* scheme that allows 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. The name of an application is represented as a URI (Uniform Resource Identifier). All of an application's resources, such as web resources, services, events, and files, are named relative to that application's URI.

The Java Card platform enforces that the application has exclusive use of its namespace. For example, an application cannot register a service under a service identifier (URI) that is not rooted in its own namespace.

Assignment of application URIs and namespaces is a key responsibility of card management applications and is essential to the deployment of multi-application provider cards that support post issuance provisioning.

Dedicated application namespace enforcement is yet another mechanism that contributes to the security containment of applications.

1.3.2 Access Control

The Connected Edition supports different complementary access control mechanisms that allow for the definition, configuration and enforcement of platform-wide, as well as per-application security policies.

1.3.2.1 Permission-based Security

Permission-based security allows a card security authority to restrict access to protected system, library and application resources based on some of the characteristics of the application requesting the access, such as the type of application model implemented (web application, extended or classic applet application) and the credentials of the application's code signer.

A *protection domain* is associated with each application group (group context). A protection domain corresponds to the set of permissions granted to an application or group of applications as per the security policy that applies to that group of applications. Each permission in a protection domain represents access to specific protected resources, such as security-sensitive system resources or application resources, such as services provided by other applications.

On the Java Card platform, the *permission-based security policy* is enforced by two types of permission checks:

- **context-switch-triggered permission checks**: Automatically initiated when an application in one context attempts to access an entry point object (a Shareable Interface Object (SIO) for example) in another context.
- **programmatic permission checks**: Programmatically initiated by the system or extension libraries.

Each permission check determines if the permission requested is granted by the protection domain of the application requesting the permission.

Permission-based security is used to restrict access to resources to only those applications which have been specifically granted the permission to access the resource.

Permission-based security provides a powerful means for a card security authority to control, with fine granularity, the resources accessed by an application or group of applications.

1.3.2.2 Role-based Security

Role-based security allows an application's security policy to restrict access to protected application resources including web resources, SIO-based services and events. The restrictions are based on some of the characteristics of the application requesting the access, such as its identity, and the identity of the user on whose behalf the access is requested.

An application developer may express the logical security requirements of the application either declaratively in the deployment descriptor of a web application or programmatically through the implementation of programmatic security checks within the code that manages access to certain resources. These *declarative security constraints* and programmatic security checks name the client or user 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 are mapped by a deployer (for example, the application provider) to actual user identities and client application identities or characteristics on the targeted platform.

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.

Role-based security provides a flexible means for an application developer to define and implement the security requirements of his application while allowing for the actual security policy to only be configured upon deployment.

1.3.2.3 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, such as required by role-based security, for accessing protected resources including SIO-based and event-based services, as well as web resources.

On the Java Card platform, users are categorized as either a *card holder*, the primary user of the card, or as *other users*, such as a remote card administrator. Each type of user has a different user identity on the card.

User authentication is implemented by *authenticators,* specialized authentication services that can use a variety of schemes to authenticate a user, such as a password, a PIN or a biometric template. These services can be invoked both by an application for *application-managed authentication* and by the web container for *container-managed authentication*.

Because several conversational sessions can be established simultaneously between on-card web applications and web clients, web user authentication is tracked on a per-session basis. *Session-scoped authentication* contrasts with the *global authentication* scope typically implemented on a classic platform in that it prevents a user authenticated in a conversational session under one identity to gain unauthorized access to protected resources authorized to another, simultaneously authenticated, identity.

Additionally, the Java Card platform distinguishes applications that are accessible locally and safely through a *card holder-facing client*, from applications that may be accessed remotely and which have stronger security requirements. Such a remotely accessible application may be an administrative user application or a card management application. To be accessed, such applications require the explicit authorization of the card holder.

1.3.2.4 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, when accessed through one of the services it exposes, may initiate and check authentication of a client application programmatically by naming the client security role permitted to access the service. That client security role is associated to the credentials of authorized application clients to which it has been mapped at during deployment. These credentials may be symmetric or asymmetric cryptographic key or certificate materials.

A client application may have to authenticate with several different server applications. The authentication of the client application with each of these server applications is tracked independently.

On-card client authentication provides a flexible and robust mechanism for managing the security policy of a server application. Instead of relying solely on the identification of an application, such as based on its identifier on the platform to grant access to its services, a server application can use credentials to authenticate its trusted clients.

1.3.3 Network Communication Security

On the Java Card platform applications may interact with off-card peers (for example, the mobile phone the card is embedded in or a server located on the Internet) through secure network communications over Secure Sockets Layer (SSL) or Transport Layer Security (TLS). These communications may be established over web container-managed HTTPS server connections, such as between an off-card client and a web application, or over application-managed client or server SSL/TLS or HTTPS connections.

The security of such network communications, also referred to as *end-to-end security*, relies on secure network protocols and cryptography services that ensure the confidentiality and the integrity of the data transmitted, as well as the authentication of the peers.

1.3.4 Key and Trust Management

Application developers may configure the security requirements and characteristics for secure communications with peers either programmatically in the application's code or declaratively in the deployment descriptor of a web application. The security requirements for a secure communication include peer authentication, integrity and confidentiality of the data transmitted.

Additionally, an application developer may use a credential manager object to manage the credentials used to establish secure communication with peers. A credential manager is used both for managing the key material that 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.

The Java Card platform allows an application to delegate to a card manager application the management of the keys and trust decisions for the secure connections that it specifically opens or that are opened on its behalf by the web container. Such a delegated model of credential management is essential for supporting card management frameworks, such as Global Platform.

1.3.4.1 Secure Hosting of Web Applications

A web application developer may declare requirements for content integrity and confidentiality in the deployment descriptor of a web application. The application developer or the application provider can also require that the application be hosted on a dedicated secure port. The web container of the Java Card platform enforces a web application's requirements for content integrity and confidentiality by only accepting requests for protected resources of that application, over HTTPS connections open on that application's dedicated secure port.

As in the case of application-managed secure communications, the security characteristics of connections from web clients on that application-dedicated HTTPS port are negotiated using the credential manager that applies to that application.

Therefore, each web application that requires protection is securely hosted on its own dedicated secure port and uses its own security requirements and credentials to establish secure communications with web clients.

1.3.4.2 Extensible Cryptography Framework

The Connected Edition supports an extensible cryptography framework that allows platform implementers to include for each cryptography service a variety of algorithms and implementations. An application developer or provider can discover all the various algorithms and implementations a card supports. Then the developer can select the cryptography algorithm and implementation that best suits the needs of his application and the constraints of the operating environment the application is deployed to. Such a selection can be made based, for example, on the provider of a particular cryptographic algorithm implementation.

1.4 Application Development and Deployment Model

The development lifecycle of applications for the Connected Edition differentiates the roles and responsibilities of the participants in the process, from the development of an application, to its assembly and distribution, and then to its deployment to a card. These roles and responsibilities are described in Section 2.3, "Application Development Participants" on page 2-3.

Throughout this development process, application descriptors and distorting and deployment units are the key artifacts that participants operate on. For this purpose and according to their roles and responsibilities, participants will use specific development tools but may also use generic development tools and will eventually use the card management facility available on the targeted card.

1.4.1 Application Descriptors

Central to an application's development lifecyle are the application's descriptors. These descriptors are documents that describe the structure, configuration and deployment information of an application, and convey this information to application developers, application assemblers, and deployers.

In the Connected Edition descriptors are of three types:

- **Application model-specific deployment descriptors**: The web application deployment descriptor and the applet application deployment descriptor. A deployment descriptor describes the elements and configuration information of an application that are dependent on the application model.
- **Java Card platform-specific application descriptor**: A Java Card platform-specific application descriptor describes the elements and configuration information of an application that are specific to the Java Card platform.
- **Runtime descriptor**: A runtime descriptor describes the configuration and deployment information of an application that are specific to an operating environment to which the application will be deployed.

The use of descriptors, especially to convey the security configuration requirements of applications, alleviates the need for hard-coding such configurations and better leverages the security mechanisms built in the platform. The security evaluation of applications for their reuse and deployment in different operating environments are, therefore, facilitated.

1.4.2 Distribution and Deployment Units

The Connected Edition supports three types of distribution and deployment units:

- The application module distribution format JAR file encapsulates one application, either a web application or an applet application:
  - The web application distribution format is a web archive (.war) file with a runtime descriptor and a Java Card platform-specific application descriptor as additional metadata files.

- The applet application distribution format contains the classes for the applet application module and the applet application model deployment descriptor file. The runtime descriptor and Java Card platform-specific application descriptor files are also included as additional metadata files.

  The classic applet application distribution format also contains the applet CAP file components (*.cap) inside the JAR file.

- The extension library JAR file is a standard library JAR file containing Java class files. Extension library classes are accessible to all applications on the card.
- The classic library JAR file is a standard JAR library format containing Java class files as well as the library CAP file components (*.cap). Classic library classes are only accessible to the classic applications on the card.

The inclusion of classic applet and library CAP file components in classic deployment units ensures that these deployment units can be deployed on platforms implementing the Classic Edition without any changes. That is not only the backward compatibility is ensured for these applications, but also the interoperability with both the Connected Edition and the Classic Edition is guaranteed.

1.4.3 Application Deployment and Card Management

The card manager in the Connected Edition supports the following post-issuance functions after proper authorization of the requesting off-card client:

- Loading of deployment units, meaning application modules and libraries
- Creation of application instances
- Deletion of application instances
- Unloading of deployment units

The Connected Edition defines a card management SPI for these post-issuance functions to enable industry-defined card management applications to be integrated with the platform. The Global Platform Card Management layers are currently being developed for the Connected Edition.

1.4.4 Application Development Tools

The application developer can use an off-the-shelf IDE or the NetBeans IDE to develop an application for the Connected Edition. The NetBeans IDE includes support specifically for Java Card 3 applications. Command-line tools are available with the Development Kit for the Java Card 3 Platform, Version 3.0.2, Connected Edition, for packaging the application binary file into the distribution unit format described above. Tools are also available in the development kit for validating and deploying onto the reference implementation of the platform.

A web application that uses the Connected Edition subset of the Java EE programming environment may be developed and run on a standard IDE. Both the servlet API as well as the binary web archive format of a web application for the Connected Edition are compatible with that of the Java EE platform.

CHAPTER 2

Programming Notes Introduction

These programming notes cover a selected set of features provided by the Java Card 3 Platform, Connected Edition. This chapter describes the features covered in this book, explains how each of the following chapters is organized, and introduces the sample used throughout this book, the Transit application.

Each chapter focuses on one of these features and is organized to cover the different phases of the development cycle from an application's design to its configuration and deployment. The chapters of these programming notes may be read in any order. Also, within a specific chapter the reader may choose to jump directly to the section that corresponds to his or her role in the application's development lifecycle.

These programming notes are not a tutorial on the Java Card 3 Platform. They complement the tutorials and programming notes for the different Java technologies that are integrated in the Java Card 3 Platform. The reader should refer to those tutorials and programming notes for other programming guidelines and best practises that may apply.

2.1 Features Described in This Book

These programming notes describe how to use some features of the Java Card 3 Platform that may be difficult to grasp at first for one of the following reasons:

- They are new and specific to the Java Card platform. No programming notes or tutorials for them exist yet, therefore, they require explanation in terms that are more practical and application developer-friendly than in the specification, which is platform developer-oriented.
- They are a subset of an existing Java technology that has been adapted to the specifics of the Java Card platform. There may be existing programming notes or tutorials, but the limitations, when compared to the full feature set, or the specific way they interact with other Java Card technology features can benefit from being described from a programmer's point of view.

These features include:

- **Authentication and authorization of application users** - the Java Card platform supports a mechanism for user authentication and authorization that integrates the user role-based security model of web applications with that of on-card SIO-based services and with the variety of authentication schemes supported on the Java Card platform.
- **Authentication and authorization of on-card applications** - the mechanism for on-card application authentication and authorization supported by the Java Card platform is based on a novel client role-based security model that uses schemes of different strength for applications to authenticate to each other.
- **End-to-end security of network connections** - the Java Card platform supports a mechanism for setting up the security of end-to-end connections that extends that of the Java ME™ technology-based Generic Connection Framework.
- **Internationalization and localization of applications** - internationalization and localization support on the Java Card platform is a limited subset of that for Java SE™ technology.
- **Migration of classic applet application to the new extended environment** - while classic applet applications may run unchanged in a strict Java Card Platform, v2.2.2 runtime emulation environment, with minimum precaution, their code may also be ported to use the richer feature set of the Java Card platform, Connected Edition
- **Packaging, distribution and deployment of applications and libraries** - the Java Card platform supports a packaging, distribution and deployment model of applications and libraries that is consistent across all the application models: web application, extended applet application and classic applet application models.

These notes give programming guidelines for each of these features. They are by no means exhaustive or limiting. Each of these features is addressed in separate, yet sometimes related, chapters.

2.2 Organization Of Each Chapter

Each chapter addresses programming issues for one of the features of the Java Card platform identified in Section 2.1, "Features Described in This Book" on page 2-1 as requiring additional guidance for application programmers. The term *application programmer* here designates all the actors in a Java Card application's development lifecycle. An application programmer is not limited to the sole application's programmer or coder. Section 2.3, "Application Development Participants" on page 2-3 describes in more detail the roles and responsibilities of the participants in the process, from the development of an application, to its assembly and distribution, and then to its deployment to a card.

Most the chapters of these programming notes have been organized according to the typical development lifecycle of Java Card applications as follows:

- **Designing the application** - this section of the chapter is addressed to the application developer. It highlights the design choices that an application developer must make to leverage the feature of the Java Card platform addressed in the chapter.
- **Implementing the application** - this section of the chapter is also addressed to the application developer. It describes how to program to the feature of the Java Card platform addressed in the chapter and how to implement different design choices.
- **Configuring and deploying the application** - this section of the chapter is addressed to the application assembler and deployer. It describes the configuration elements of the Java Card platform feature addressed in the chapter and how to set up the application for deployment in a target operating environment.

2.3 Application Development Participants

The development lifecycle of applications for the Connected Edition differentiates the roles and responsibilities of the participants in the process, from the development of an application, to its assembly and distribution, and then to its deployment to a card.

1. **Application developer**: The application developer creates the application. The output of an application developer is a set of application classes and resources, and supporting libraries 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.

2. **Application assembler**: The 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 for the application. The output of the application assembler is an application archive conforming to one of the distribution formats supported by the Java Card platform.

3. **Application deployer**: The application deployer takes one or more application archive files provided by an application developer and deploys the application onto 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. 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. The application deployer usually corresponds to the application provider.

Note that these roles are only logical roles. An actual participant, such as an application provider, may perform several of these roles. A Java Card technology application provider may, for example, buy from one or several application developers an application or the components from which he may assemble his application. The application provider may then provision or deploy the application to cards in the field.

2.4 Transit Sample Application Overview

Throughout this book the Transit reference application is used as an example to illustrate programming guidelines and best practices. The Transit reference application is a suite of samples that exercises many of the use cases that the Java Card 3 Platform targets. The Transit application is a Java Card application for mobile ticketing. Such an application could be provisioned onto cards issued to the users of an urban transit system.

The Transit sample application is available at

`http://kenai.com/projects/javacard/pages/JC3Transit`

although it must be run from within the NetBeans IDE.

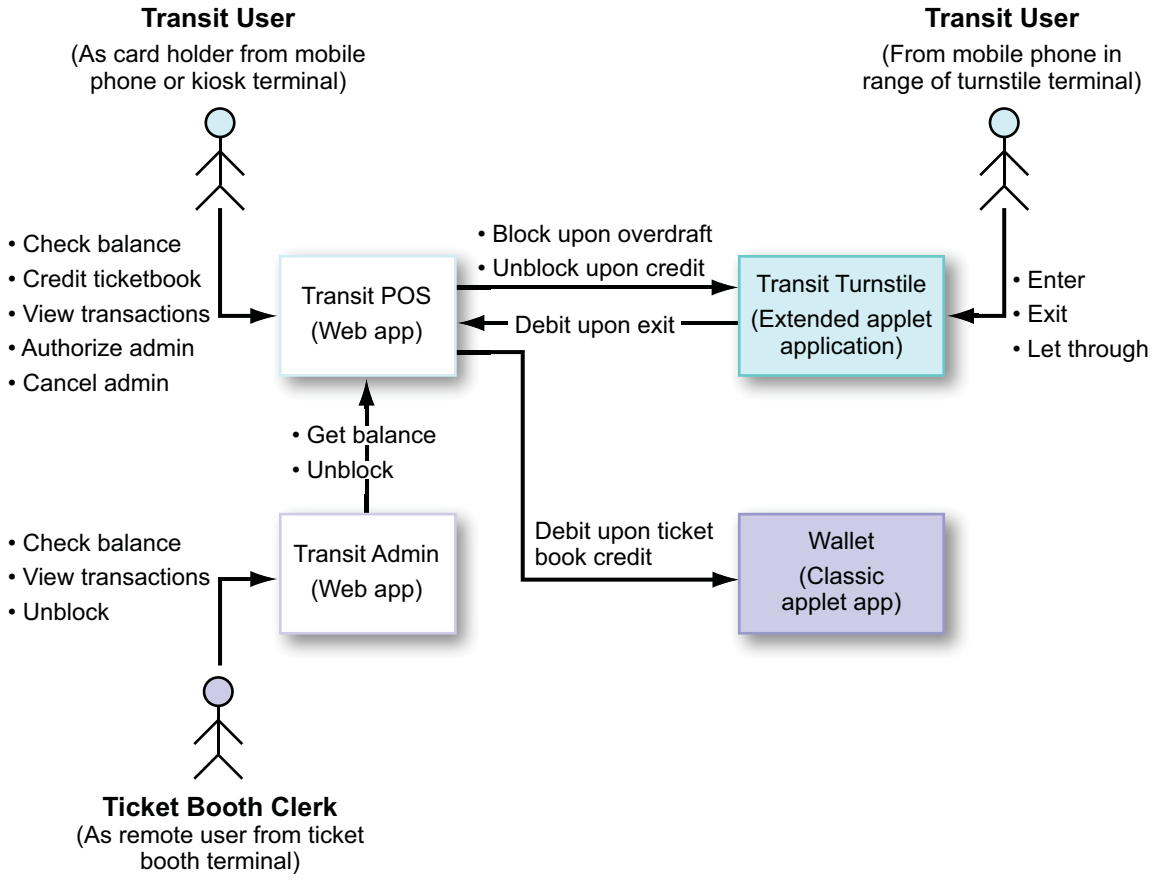
The following sections describe in detail the architecture and use cases of the Transit application, and highlights the specific features of the Java Card platform that it exercises. For information on the Transit application's use cases, see Section 2.5, "Use Cases of the Transit Application" on page 2-8.

2.4.1 Architecture Of The Transit Application

The Transit reference application is composed of three interoperating applications, *Transit Point of Sale*, *Transit Admin*, and *Transit Turnstile*. In addition, the Transit application makes use of the *Wallet* application from the Classic Edition.

The collaboration diagram in FIGURE 2-1 summarizes the architecture and the interactions between the different applications that comprise the Transit application.

FIGURE 2-1 Architecture and Interactions of the Transit Application Suite



2.4.1.1 Transit Point Of Sale (POS) Web Application

This application is a web application that interacts with the card holder (the transit system user) through a web front-end and also interoperates with other trusted on-card applications through a Shareable Interface Object-based service.

The web front-end of the Transit POS application allows the card holder to perform the following ticketing account operations:

- get the ticket book balance
- credit the ticket book
- get the transaction history (time and amount of debits and credits)
- authorize remote administration of the ticket book
- cancel remote administration of the ticket book

Access to the Transit POS web application is only allowed from a card holder-facing terminal, such as the owner's mobile phone. Moreover, access to the ticketing account operations require authentication of the user. The protected web content of the Transit POS web application is hosted on a secure port that requires confidentiality and integrity of the connections from web clients.

The ticket book state and operations are encapsulated in a *TicketBook* object. The Transit POS web application exposes a *TicketBook* service that gives to other trusted on-card applications restricted access to the following operations of the ticket book:

- get the ticket book balance
- debit the ticket book
- get the transaction history (time and amount of debits and credits)
- unblock the ticket book

The ticket book also fires events to notify other trusted on-card applications of the following conditions:

- ticket book overdraft
- ticket book credited
- ticket book unblocked

Access to the operations exposed and the events fired by the *TicketBook* service require the Transit Turnstile applet application to authenticate to the Transit POS application with credentials, whereas the Transit Admin web application is trusted based on its on-card identifier.

2.4.1.2 Transit Admin Web Application

This application is a web application that interacts with a remote ticket booth clerk through a web front-end. It also interoperates with the Transit POS web application through the *TicketBook* service that application exposes.

The web front-end of the Transit Admin application allows a ticket booth clerk to perform the following administrative operations on the owner’s ticketing account:

- get the ticket book balance
- get the transaction history (time and amount of debits and credits)
- unblock the ticket book

The Transit Admin web application delegates these operations to the *TicketBook* service exposed by the Transit POS application.

Access to the Transit Admin web application is allowed from a non-card holder-facing terminal, such as a ticket booth terminal. Access to the Transit Admin web application must be explicitly authorized by the card holder. Moreover, access to the ticketing account operations require the user (the ticket booth clerk) to

authenticate. The protected web content of the Transit Admin web application is hosted on a secure port that requires confidentiality and integrity of the connections from web clients.

2.4.1.3 Transit Turnstile Applet Application

This application is an extended applet application that interacts over the contactless interface with Transit Turnstile Terminals. It also interoperates with the Transit POS web application through the *TicketBook* service that application exposes.

The contactless APDU applet processes the following commands from Transit Turnstile Terminals:

- enter at a transit zone through the turnstile gate - this operation is only allowed if the Transit user has a positive ticket book balance.
- exit at a transit zone through the turnstile gate - this operation is only allowed if the Transit user has a ticket book balance sufficient to pay for the ride. Otherwise the ticket book is blocked.
- let through a turnstile gate - this operation may be requested by a ticket booth clerk to let the Transit user exit through the turnstile gate despite his/her insufficient ticket book balance.

The Transit Turnstile applet application delegates ticket book operations to the *TicketBook* service exposed by the Transit POS application. The Transit Turnstile applet application also monitors the ticket book's overdraft, credited and unblocked events to keep the Transit user's "clearance" up-to-date.

The interaction between a turnstile terminal and the Transit Turnstile applet is secured by an ad hoc mutual authentication protocol. Additionally, the let-through privileged operation requires the ticket booth clerk to authenticate with a PIN.

2.4.1.4 Related Wallet Application

The Transit POS web application interacts with yet another, fourth application, the Wallet application, which is not formally part of the Transit application but which is one of the sample applications for the Java Card Platform, Classic Edition. This classic applet application has been enhanced to allow for on-card interaction with other applications, specifically the Transit POS web application, through a Shareable Interface Object-based service. The interaction between the Transit POS web application and the Wallet applet application is performed through a helper application, the Wallet Assist applet application:

- **Wallet applet application** - this application is a classic applet application that interacts over a contacted or contactless interface with a Card Acceptance Device (CAD). The Wallet applet application stores electronic money and supports credit,

debit and check-balance operations when connected to a CAD. It also exposes the *WalletAccess* service to other on-card applications, which allows for the following operation:

- debit the wallet

The Transit POS web application delegates to the Wallet applet application the financial transaction required to credit the ticket book.

- **WalletAssist applet application** - this application is an extended applet application. It exposes a Shareable Interface Object-based service, namely the *WalletAssist* service, that allows for the Transit POS web application to indirectly lookup and operate on the *WalletAccess* service. This proxying through an extended applet application is necessary because web applications cannot directly look up services exposed by classic applet applications. The *WalletAccess* service allows for the following operation:
  - look up and debit the wallet

2.5 Use Cases of the Transit Application

This section describes the main use cases of the Transit application. These use cases represent the majority of the functions the Transit application is programmed to perform.

2.5.1 Point Of Sale Use Cases

▼ Logging In

1. **Pre-conditions**
   - The Transit POS application is installed (loaded and instantiated)
   - The card is physically connected to a trusted terminal (the user's mobile phone or a POS kiosk)
   - The Transit System user (the card holder) is not logged in the Transit POS application
2. **Main Flow of Events**
   a. The user opens his browser on the Transit POS web application's URL (`http://<card>:8019/transit/pos`)
   b. The unprotected Home page of the Transit POS web application is displayed

c. The user clicks the [`Continue`] button

d. The browser is redirected to the protected page displaying the ticket book balance, which requires HTTP Basic authentication

e. The browser displays the HTTP Basic authentication dialog asking for a user name and password

f. The user enters his user name (`owner-pos`) and PIN (`8888`)

g. The credentials are validated by the web container

h. The ticket book balance page is displayed

3. **Alternate Flows**
   - The user enters wrong credentials (user name or PIN) - the user is prompted again
4. **Special Requirements**
   - A secure HTTP connection (HTTPS) must be used for user authentication
5. **Post-conditions**

   a. The user is logged in as `owner-pos`

   b. The browser displays the ticket book balance page

   c. The browser maintains the `owner-pos` user's authentication credentials to be sent upon each request to Transit POS web application's URLs

▼ Logging Out

1. **Pre-conditions**
   - The Transit System user (the card holder) is logged in the Transit POS application (see Logging In)
2. **Main Flow of Events**

   a. The user clicks the [`LOGOUT`] link

   b. The web container ends the current HTTP session and cancels the remote administration authorization, if authorized

   c. The browser displays the HTTP Basic authentication dialog asking for a user name and password

   d. The user clicks the [`Cancel`] button without entering any credentials

   e. The web container logs the user out and rejects the request

f. The browser displays an error page with error code 401 and message "UNAUTHORIZED"

3. **Alternate Flows**
   - N/A
4. **Special Requirements**
   - The user's authentication credentials maintained by the browser must be cleared to effectively logout
   - The remote administration authorization must be cancelled upon logout
5. **Post-conditions**

   a. The user is logged out

   b. The browser has cleared the user's authentication credentials it maintained locally

▼ Logging In As Guest User

1. **Pre-conditions**
   - The Transit POS application is installed (loaded and instantiated)
   - The card is physically connected to a trusted terminal (the user's mobile phone or a POS kiosk)
   - The Transit System guest user (the card holder) is not logged in the Transit POS application
2. **Main Flow of Events**

   a. The user opens his browser on the Transit POS web application's URL (`http://<card>:8019/transit/pos`)

   b. The unprotected Home page of the Transit POS web application is displayed

   c. The user clicks the [`Continue`] button

   d. The browser is redirected to the protected page displaying the ticket book balance which requires HTTP Basic authentication

   e. The browser displays the HTTP Basic authentication dialog asking for a user name and password

   f. The user enters his user name (`guest-pos`) and PIN (`8888`)

   g. The credentials are validated by the web container

h. A **restricted** ticket book balance page is displayed (providing a restricted view of the balance information)

3. **Alternate Flows**
   - The user enters wrong credentials (user name or PIN) - the user is prompted again
4. **Special Requirements**
   - A secure HTTP connection (HTTPS) must be used for user authentication
5. **Post-conditions**
   a. The user is logged in as `guest-pos`
   b. The browser displays a restricted ticket book balance page
   c. The browser maintains the `guest-pos` user's authentication credentials to be sent upon each request to Transit POS web application's URLs

▼ Access Attempt From Non-Card Holder Facing Client

1. **Pre-conditions**
   - The Transit POS application is installed (loaded and instantiated)
   - The card is either physically connected to ticket booth clerk's terminal or, while being physically connected to a trusted terminal (the user's mobile phone or a POS kiosk), the card is accessed remotely (by the ticket booth clerk)
2. **Main Flow of Events**
   a. The user opens his browser on the Transit POS web application's URL (`http://<card>:8019/transit/pos`)
   b. The web container rejects the request
   c. The browser displays an error page with error code 403 and message "`Not Accessible From Client`"
3. **Alternate Flows**
   - N/A
4. **Special Requirements**
   - The Transit POS application is not remotely accessible and must systematically reject any access from non-card holder facing clients
5. **Post-conditions**
   - N/A

▼ Changing Locale

1. **Pre-conditions**
   - The Transit POS application is installed (loaded and instantiated)
   - The card is physically connected to a trusted terminal (the user's mobile phone or a POS kiosk)
   - The Transit System user (the card holder) is logged out of the Transit POS application
2. **Main Flow of Events**
   a. The user changes the preferred language for display pages in his browser, for example to French
   b. The user logs in the Transit POS application (see Logging In)
3. **Alternate Flows**
   - None of the user's accepted languages is supported - the default platform and/or application language is then used
4. **Special Requirements**
   - N/A
5. **Post-conditions**
   - The pages of the application are displayed in the user's preferred language

▼ Checking Ticket Book Balance

1. **Pre-conditions**
   - The Transit System user (the card holder) is logged in the Transit POS application (see Logging In)
   - The ticket book balance page is displayed
2. **Main Flow of Events**
   a. The user clicks the [`REFRESH BALANCE`] button
   b. The ticket book balance page is displayed
3. **Alternate Flows**
   - N/A
4. **Special Requirements**
   - The ticket book balance operation is protected by authentication and requires a secure HTTP connection (HTTPS)

5. **Post-conditions**
   - An up-to-date ticket book balance page is displayed

▼ Crediting Ticket Book

1. **Pre-conditions**
   - The Wallet application is installed (loaded and instantiated)
   - The WalletAssist application is installed (loaded and instantiated)
   - The Transit System user (the card holder) is logged in the Transit POS application (see Logging In)
   - The ticket book balance page is displayed
2. **Main Flow of Events**
   a. The user clicks the [`CREDIT`] button
   b. The ticket book credit page is displayed
   c. The user enters a number of ride tickets to credit and hits the [`SUBMIT`] button
   d. The wallet is debited by the amount of money corresponding to the requested number of ride tickets and the ticket book is reliably credited
   e. The ticket book balance page is displayed
3. **Alternate Flows**
   - A tear occurs during any of the processing steps and/or an abort transaction is performed after any operation
   - The wallet balance is not sufficient - the crediting of the ticket book fails
4. **Special Requirements**
   - Transaction protection is used to ensure the integrity of the ticket book balance. In the event of a rollback the balance is not changed
   - The ticket book credit operation is protected by authentication and requires a secure HTTP connection (HTTPS)
5. **Post-conditions**
   a. The wallet has been debited
   b. The ticket book has been credited
   c. An up-to-date ticket book balance page is displayed

▼ Viewing Ticket Book Transaction History

1. **Pre-conditions**
   - The Transit System user (the card holder) is logged in the Transit POS application (see Logging In)
   - The ticket book balance page is displayed
2. **Main Flow of Events**
   a. The user clicks the [`HISTORY`] button
   b. The ticket book transaction history page is displayed
3. **Alternate Flows**
   - N/A
4. **Special Requirements**
   - The ticket book transaction history operation is protected by authentication and requires a secure HTTP connection (HTTPS)
5. **Post-conditions**
   - N/A

▼ Authorizing Ticket Booth Clerk Remote Administration

1. **Pre-conditions**
   - The Transit System user (the card holder) is logged in the Transit POS application as `owner-pos` (see Logging In)
   - Remote access is not yet authorized: the [`AUTHORIZE`] button is enabled and the [`CANCEL`] button is disabled
2. **Main Flow of Events**
   a. The user clicks the [`AUTHORIZE`] button
   b. The browser displays the HTTP Basic authentication dialog asking for a user name and password
   c. The user enters his user name (`owner-admin`) and PIN (`8888`)
   d. The credentials are validated by the web container
   e. The remote administration page is displayed, showing the message "`Remote administration authorized`"
3. **Alternate Flows**

- The user enters wrong credentials (user name or PIN) - the user is prompted again

4. **Special Requirements**
   - Remote administration authorization is protected by authentication and requires a secure HTTP connection (HTTPS)
5. **Post-conditions**
   a. Remote administration is authorized (see Section 2.5.2, "Remote Administration Use Cases" on page 2-17)
   b. The user is logged out from the `owner-pos` user account
   c. The user is logged in as `owner-admin`
   d. The browser has overwritten the `owner-pos` user's authentication credentials it maintains locally with the `owner-admin` user's authentication credentials
   e. The user must click on the [`HOME`] link and log in again as `owner-pos` (see Logging In) to proceed with POS operations

▼ Cancelling Ticket Booth Clerk Remote Access

1. **Pre-conditions**
   - The Transit System user (the card holder) is logged in the Transit POS application as `owner-pos` (see Logging In)
   - Remote access has been authorized (see Authorizing Ticket Booth Clerk Remote Administration): the [`AUTHORIZE`] button is disabled and the [`CANCEL`] button is enabled
2. **Main Flow of Events**
   a. The user clicks the [`CANCEL`] button
   b. The remote administration page is displayed, showing the message "`Remote administration denied`"
   c. The user clicks on the [`HOME`] link
   d. The Home page of the Transit POS web application is displayed
   e. The user clicks the [`Continue`] button
   f. The ticket book balance page is displayed
3. **Alternate Flows**
   - N/A

4. **Special Requirements**
   - Remote administration authorization cancelling is protected by authentication and requires a secure HTTP connection (HTTPS)
5. **Post-conditions**
   - Remote administration is denied (see Section 2.5.2, "Remote Administration Use Cases" on page 2-17): the [AUTHORIZE] button is enabled and the [CANCEL] button is disabled

▼ Crediting Ticket Book To Unblock After Issuance or After Overdraft

1. **Pre-conditions**
   - The Wallet application is installed (loaded and instantiated)
   - The WalletAssist application is installed (loaded and instantiated)
   - The Transit System user (the card holder) is logged in the Transit POS application (see Logging In)
   - The ticket book is blocked and the ticket book balance page is displayed with a null balance (such as immediately after the card has been issued to the user, see Entering) or a negative balance (such as when the last ride of the user was worth more ride tickets than what he had, see Exiting)
2. **Main Flow of Events**
   a. The user credits his ticket book (see Crediting Ticket Book)
   b. The ticket book balance page is displayed with a null or positive balance
3. **Alternate Flows**
   - A tear occurs during any of the processing steps and/or an abort transaction is performed after any operation
   - The wallet balance is not sufficient - the crediting of the ticket book fails and the ticket book remains blocked
4. **Special Requirements**
   - Transaction protection is used to ensure the integrity of the ticket book balance. In the event of a rollback the balance is not changed.
   - The ticket book credit operation is protected by authentication and requires a secure HTTP connection (HTTPS)
5. **Post-conditions**
   a. The wallet has been debited

b. The ticket book has been credited

c. The ticket book is unblocked

d. An up-to-date ticket book balance page is displayed

2.5.2 Remote Administration Use Cases

▼ Access Attempt Without Card Holder Authorization

1. Pre-conditions

- The Transit POS application is installed (loaded and instantiated)
- The Transit Admin application is installed (loaded and instantiated)
- The card is either physically connected to the ticket booth clerk's terminal or, while being physically connected to a card holder facing terminal (the user's mobile phone or a POS kiosk) the card is accessed remotely by the ticket booth clerk

2. Main Flow of Events

a. The user opens his browser on the Transit Admin web application's URL (`http://<card>:8019/transit/admin`)

b. The web container rejects the request

c. The browser displays an error page with error code 401 and message "`Need Card Holder authorization`"

3. Alternate Flows

- N/A

4. Special Requirements

- The Transit Admin application is remotely accessible and requires card holder authorization

5. Post-conditions

- N/A

▼ Logging In

1. **Pre-conditions**
   - The Transit POS application is installed (loaded and instantiated)
   - The Transit Admin application is installed (loaded and instantiated)
   - The card is either physically connected to the ticket booth clerk's terminal or, while being physically connected to a card holder facing terminal (the user's mobile phone or a POS kiosk) the card is accessed remotely by the ticket booth clerk
   - The card holder (the Transit System user) has authorized remote administration (see Authorizing Ticket Booth Clerk Remote Administration)
2. **Main Flow of Events**
   a. The user opens his browser on the Transit Admin web application's URL (`http://<card>:8019/transit/admin`)
   b. The unprotected Home page of the Transit Admin web application is displayed
   c. The user clicks the [`Continue`] button
   d. The browser is redirected to the protected page displaying the ticket book balance. The protected page requires form-based, Java Card platform-specific authentication
   e. The browser displays the login page asking for a user name and PIN
   f. The user enters his user name (`remote-admin`) and PIN (`8888`)
   g. The credentials are validated by the web container
   h. The ticket book balance page is displayed
3. **Alternate Flows**
   - The user enters wrong credentials (user name or PIN) - the user is prompted again
4. **Special Requirements**
   - A secure HTTP connection (HTTPS) must be used for user authentication
5. **Post-conditions**
   a. The user is logged in as `remote-admin`
   b. The browser displays the ticket book balance page

▼ Logging Out

1. **Pre-conditions**
   - The ticket booth clerk is logged in the Transit Admin application (see Logging In)
2. **Main Flow of Events**
   a. The user clicks the [LOGOUT] link
   b. The web container logs the user out
   c. The unprotected Home page of the Transit Admin web application is displayed
3. **Alternate Flows**
   - N/A
4. **Special Requirements**
   - N/A
5. **Post-conditions**
   a. The user is logged out
   b. The unprotected Home page of the Transit Admin web application is displayed

▼ Checking Ticket Book Balance

1. **Pre-conditions**
   - The ticket booth clerk is logged in the Transit Admin application (see Logging In)
   - The ticket book balance page is displayed
2. **Main Flow of Events**
   a. The user clicks the [REFRESH BALANCE] button
   b. The ticket book balance page is displayed
3. **Alternate Flows**
   - N/A
4. **Special Requirements**
   - The ticket book balance operation is protected by authentication and requires a secure HTTP connection (HTTPS)
5. **Post-conditions**

- An up-to-date ticket book balance page is displayed

▼ Viewing Ticket Book Transaction History

1. **Pre-conditions**
   - The ticket booth clerk is logged in the Transit Admin application (see Logging In)
   - The ticket book balance page is displayed
2. **Main Flow of Events**
   a. The user clicks the [HISTORY] button
   b. The ticket book transaction history page is displayed
3. **Alternate Flows**
   - N/A
4. **Special Requirements**
   - The ticket book transaction history operation is protected by authentication and requires a secure HTTP connection (HTTPS)
5. **Post-conditions**
   - N/A

▼ Unblocking Ticket Book

1. **Pre-conditions**
   - The ticket booth clerk is logged in the Transit Admin application (see Logging In)
   - The ticket book is blocked and the ticket book balance page is displayed with a null balance (such as immediately after the card has been issued to the user, see Entering) or a negative balance (such as when the last ride of the user was worth more ride tickets than what he had, see Exiting)
2. **Main Flow of Events**
   a. The user clicks the [UNBLOCK] button
   b. The ticket book transaction history page is displayed
3. **Alternate Flows**
   - N/A
4. **Special Requirements**

- N/A

5. **Post-conditions**
   - The ticket book is unblocked

2.5.3 Turnstile Use Cases

▼ Entering

1. **Pre-conditions**
   - The Transit POS application is installed (loaded and instantiated)
   - The Transit Turnstile application is installed (loaded and instantiated)
   - The Transit System user (the card holder) has brought his card in range of the turnstile terminal

2. **Main Flow of Events**

   a. The user brings his card in range of the contactless terminal of the entry turnstile

   b. The card is activated

   c. A secure session is initiated

   d. The ticket book is checked – it must not be blocked

   e. The entry zone ID is checked – it must be null

   f. The entry zone ID is reliably recorded

   g. The entry event is logged

   h. The secure session is terminated

   i. The turnstile opens

   j. The user moves his card out of the reader's range (tear)

3. **Alternate Flows**
   - A tear occurs during any of the processing steps
   - An abort transaction is performed after any operation
   - The ticket book is blocked - the entry is denied
   - The ticket book balance is null or negative - the entry is denied

- The entry zone ID is not null – a transit is already in progress or the user attempts to perform multiple “illegal” entries - the entry is denied

4. **Special Requirements**
   - A secure session is used for the transaction, to authenticate both the card and the terminal application to each other
   - Transaction protection is used to ensure the integrity of the data on the card. In the event of a rollback, the entry zone ID on the card is not changed
   - The transaction must complete within N milliseconds because of the use of a contactless terminal
5. **Post-conditions**
   - The card is deactivated
   - If the turnstile was opened, the entry zone ID was recorded
   - If the turnstile was not opened, the entry zone ID was not changed

▼ Exiting

1. **Pre-conditions**
   - The Transit POS application is installed (loaded and instantiated)
   - The Transit Turnstile application is installed (loaded and instantiated)
   - The Transit System user (the card holder) has brought his card in range of the turnstile terminal
2. **Main Flow of Events**

   a. The user brings his card in range of the contactless terminal of the exit turnstile

   b. The card is activated

   c. A secure session is initiated

   d. The ticket book is checked – it must not be blocked

   e. The entry zone ID is checked – it must not be null

   f. The ticket book balance is reliably debited by the ticket number corresponding to the fare for the ride (computed from the entry zone and the exit zone)

   g. The entry zone ID is reset to null

   h. The entry event is logged

   i. The secure session is terminated

   j. The turnstile opens

k. The user moves his card out of the reader's range (tear)

3. **Alternate Flows**
   - A tear occurs during any of the processing steps
   - An abort transaction is performed after any operation
   - The entry zone ID is null – signifying that the user entered the Transit System "illegally" - the exit is denied
   - The ticket book is blocked - the exit is denied
   - The ticket book balance becomes negative after debiting the fare for the ride (overdraft) - the exit is denied

4. **Special Requirements**
   - A secure session is used for the transaction, to authenticate both the card and the terminal application to each other
   - Transaction protection is used to ensure the integrity of the data on the card. In the event of a rollback the entry zone ID on the card is not changed
   - The transaction must complete within N milliseconds because of the use of a contactless terminal

5. **Post-conditions**
   - The card is deactivated
   - If the turnstile was opened, the ticket book balance is updated, the entry zone ID is reset, and the transaction is logged on the card
   - If the turnstile was not opened, the ticket book balance was not changed and the entry zone ID was not changed

▼ Let Through

1. **Pre-conditions**
   - The Transit POS application is installed (loaded and instantiated)
   - The Transit Turnstile application is installed (loaded and instantiated)
   - The Transit System user (the card holder) has brought his card in range of the turnstile terminal
   - The user was denied exiting because of a ticket book overdraft, see alternate flows of Exiting

2. **Main Flow of Events**

   a. The user brings his card in range of the contactless terminal of the exit turnstile

   b. The card is activated

c. A secure session is initiated

d. The ticket booth clerk commands the turnstile terminal to let the user exit through the gate

e. The turnstile terminal authenticates with the card as the user `remote-admin` using a PIN (`8888`)

f. The turnstile application verifies the provided credentials

g. The entry zone ID is checked – it must not be null

h. The entry zone ID is reset to null

i. The secure session is terminated

j. The turnstile opens

k. The user moves his card out of the reader's range (tear)

3. Alternate Flows

- A tear occurs during any of the processing steps
- An abort transaction is performed after any operation
- The entry zone ID is null – signifying that the user entered the Transit System "illegally" - the let-through operation fails

4. Special Requirements

- A secure session is used for the transaction, to authenticate both the card and the terminal application to each other
- Transaction protection is used to ensure the integrity of the data on the card. In the event of a rollback the entry zone ID on the card is not changed
- The transaction must complete within N milliseconds because of the use of a contactless terminal

5. Post-conditions

- The card is deactivated
- If the turnstile was opened, the entry zone ID is reset
- If the turnstile was not opened, the entry zone ID was not changed

CHAPTER 3

Authenticating and Authorizing Application Users

User security on the Java Card platform is supported through *user authentication* and *user authorization*. User authentication is the process by which a user proves his or her identity to the card. Whereas user authorization is the process by which access to protected resources, such as web resources and SIO-based and event-based services, may be granted to a user based on his or her authenticated identity.

The Java Card platform defines two categories of users, the *card holder*, the primary user of the card, or *other users*, such as a remote card administrator. To these two *user categories* correspond distinct privileges and identities on the card.

User authentication is implemented by *authenticators,* specialized authentication services that can use a variety of schemes to authenticate a user, such as a password, a PIN or a biometric template. These services can be invoked both by an application for *application-managed authentication* and by the web container for *container-managed authentication*.

Because several conversational sessions can be established simultaneously between on-card web applications and web clients, web user authentication is tracked on a per-session basis. *Session-scoped authentication* contrasts with *global authentication* typically implemented on a classic platform in that it allows for two different users concurrently accessing the platform to be authenticated independently.

Additionally, the Java Card platform distinguishes applications that are accessible locally and safely through a *card holder-facing client* from applications that may be accessed remotely and which have stronger end-to-end security requirements. Such a remotely accessible application may be an administrative user application or a card management application. To be accessed remotely, such applications require the explicit authorization of the card holder.

On the Java Card platform, user authorization is mainly supported through user role-based security. User role-based security is used to restrict access to protected application resources, including web resources, SIO-based services and events, strictly to those users who have been granted a particular security role. A *user*

security role is a logical grouping of users defined by the application developer or assembler. When the application is deployed, user roles are mapped by a deployer (for example, the application provider) to actual user identities on the targeted platform. Role-based security provides a flexible means for an application developer to define and implement the security requirements of his application while allowing for the actual security policy to be configured upon deployment.

An application developer may express the logical security requirements of an application either declaratively in the deployment descriptor of a web application or programmatically through the implementation of programmatic security checks within the code that manages access to certain resources. These *declarative security constraints* and programmatic security checks name the user security roles permitted to access the protected resources.

In some cases, application developers may also directly use the authentication services without resorting to role-based security. This may be required for applet applications, especially classic applet applications for which container-managed authentication is not available and which have no access to the new role-based security API.

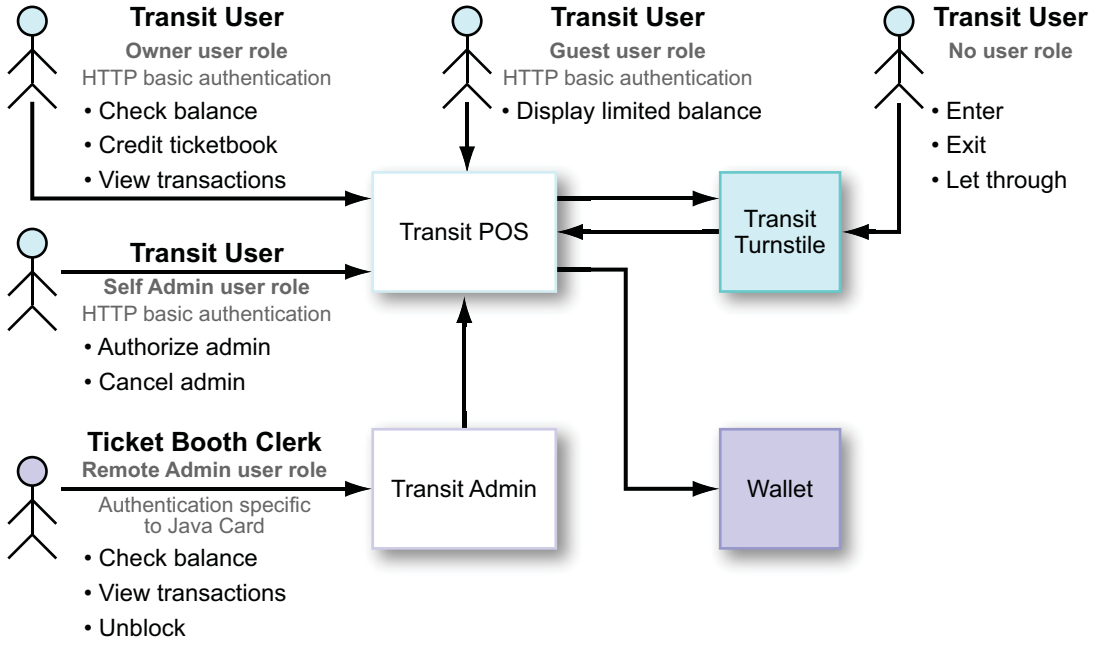
3.1 Designing An Application's User Security

To take full advantage of the user security facilities offered by the Java Card platform, you, as an application developer, must first analyze the user security requirements of your application and gather the following information:

- *Accessibility Requirements and Categories of Users* - if the application will be locally accessible or remotely accessible and the categories of user roles (card holder or other users) that will have access to the application.
- *User Roles and Protected Functions* - whether certain functions of the application must be protected so they can be accessed exclusively by users in certain roles.
- *Authentication Methods* - what scheme and strength of authentication is required for the application.

The diagram in FIGURE 3-1 shows the use cases of the Transit application with highlights on the functional user roles and authentication methods.

FIGURE 3-1 Functional User Roles and Authentication Methods



3.1.1 Determining The Accessibility Requirements

To allow for securing (granting or denying) remote access to web applications, the Java Card platform implements a mechanism that requires the card holder to authorize remote access to web applications.

Note – This mechanism does not apply to applet applications.

Therefore, you, as a web application developer, must decide whether your application will be accessible locally through a *card holder-facing client*, or if it will be accessed remotely from a *non-card holder-facing client*:

- If your web application will be accessed remotely, it requires card holder authorization. That is, access to your web application should only be granted by the web container if the card holder is currently authenticated in a role that authorizes the remote access.
- If your web application will be accessed locally from a card holder-facing client, there is no restriction because the sole expected user is the card holder himself.

Note – A remotely accessible web application can also be accessed locally from a card holder-facing client, in which case the access will not be subjected to card holder authorization because the sole expected user is the card holder himself.

TABLE 3-1 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 3-1 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

You can implement card holder authorization requirements by defining card holder authorization constraints. See Section 3.2.3.4, "Defining Card Holder Authorization Constraint" on page 3-30.

For example, the Transit Admin web application is intended to be accessed by a ticket booth clerk when the transit user has blocked his card (such as because of insufficient ticket book credit). The ticket booth clerk will be accessing the Transit Admin web application through a terminal, which does not qualify as a card holder-facing client. Therefore, such an access will require the explicit authorization of the card holder, the transit user. The transit user will authorize the access by the ticket booth clerk to the Transit Admin web application by authenticating in a dedicated role.

The Transit POS web application and the Transit Admin web application are intended to be used by both the card holder and other users, namely ticket booth clerks. TABLE 3-2 lists for both of these applications the categories of users and the functions or resources they are intended to access.

TABLE 3-2 User Categories and Protected Functions or Resources for the Transit POS and Transit Admin Web Applications

Application	User Category	Protected Function(s) or Resource(s)
Transit POS web application	Card holder	Ticket book balance operation Ticket book history operation Ticket book credit operation
	Card holder	Owner authentication for card holder authorization
	Other user (ticket booth clerk)	Ticket book unblock operation
Transit Admin web application	Card holder	Authorizing remote access to the application
	Other user (ticket booth clerk)	Ticket book balance operation Ticket book history operation Ticket book unblock operation

3.1.2 Defining The User Roles

If there are certain functions or resources within your application that require specific privileges, you must decide the user roles permitted access to each function or resource. Such a user role is a logical grouping of users that does not designate any specific user at this point in the development lifecycle. The specific users will be identified later when your application is deployed into an actual operating environment.

Note – The card holder authorization roles mentioned in Section 3.1.1, "Determining The Accessibility Requirements" on page 3-3 are additional examples of user security roles.

Coding with user roles instead of with actual user identities gives you more flexibility when creating the application's configuration and facilitates the reuse of the application or of some of its components in a different setting.

For example, the Transit POS web application and the Transit Admin web application require various user security roles to be defined. TABLE 3-3 lists for both of these applications the user security roles and the functions or resources they protect.

TABLE 3-3 User Security Roles and Protected Functions or Resources for the Transit POS and Transit Admin Web Applications

Application	User Security Role	User Category	Protected Function(s) or Resource(s)
Transit POS web application	`OWNER-POS`	Card holder	Ticket book balance operation Ticket book history operation Ticket book credit operation
	`OWNER-ADMIN`	Card holder	Owner authentication for card holder authorization
	`ADMIN`	Other user	Ticket book unblock operation
Transit Admin web application	`OWNER-ADMIN`	Card holder	Authorizing remote access to the application
	`ADMIN`	Other user	Ticket book balance operation Ticket book history operation Ticket book unblock operation

3.1.3 Determining The Authentication Method

The authentication services on the Java Card platform support a variety of authentication schemes:

- PIN-based authentication
- Password-based authentication
- Biometric authentication through a variety of optional schemes such as fingerprint, iris scan, and voice print.[1]

Additional schemes may be supported by specific platform implementations.

The authentication method you choose for your application impacts its implementation when authentication is application managed, see Section 3.2.2.2, "Coding Application-managed Authentication" on page 3-12. In such a situation, your application has to manage programmatically both the prompting of the user for its credentials and the validation of those credentials with the scheme-specific authentication service that corresponds to the identity being authenticated.

1. An exhaustive list is given in the `javacardx.biometry.BioBuilder` class

If your application is a web application that uses container-managed authentication, your choice of the authentication scheme may have little impact on its implementation. Moreover, the authentication scheme may be configured later during deployment. Therefore, when possible, you should favor container-managed authentication over application-managed authentication for the additional deployment flexibility and code reusability it provides.

3.2 Implementing An Application's User Security

Based on your design decisions, implementing your application's user security may encompass the following:

- Choosing whether your application will manage the authentication of users itself (at the application level) or whether it will leverage the user authentication facility provided by the web container. And, accordingly implementing your application's user authentication either by coding application-managed authentication or by configuring container-managed authentication.
- Implementing your application's user authorization by configuring the declarative user authorization constraints, if your application is a web application that relies on container-managed authentication, or by coding the programmatic user role-based security checks that will guard the resources or functionalities that you want to protect. In some cases where role-based security does not apply, you may resort to coding application-specific security checks.
- Configuring a card holder authorization constraint for your remotely accessible web application so that access to your web application will only be granted if the card holder is currently authenticated in a role that authorizes the remote access.

3.2.1 Choosing Between Container-managed and Application-managed Authentication

Container-managed authentication is only available for web applications. As a web application developer, you should rarely have to implement user authentication at the application level and should always favor container-managed authentication. You should choose application-managed authentication over container-managed authentication only if you seek either to better control the authentication protocol and the interaction with the user or to overcome the limitation that a web application can only be configured to use a single authentication method.

If your application is an applet application, you must manage authentication of your application's users at the application level or rely on application frameworks that do so. See Section 3.2.2, "Implementing User Authentication" on page 3-8.

Container-managed and application-managed authentications can be performed with global authenticators as well as session authenticators. Some restrictions nevertheless apply on application-managed authentication with respect to user role-based security. TABLE 3-4 summarizes the cases where user role-based security applies.

TABLE 3-4 Applicability of User Role-based Security

	Container-managed Authentication	Application-managed Authentication
Global authentication	Applicable	Applicable
Session authentication	Applicable	Not applicable*

* The Java Card Platform does not currently provide an API for programmatically binding an authenticated identity to a session and to the threads handling requests or commands within that session. The role-based security API only applies to globally authenticated identities and to the session-scoped identity bound to thread executing invoking that API.

3.2.2 Implementing User Authentication

As the application developer, you must gather the following information on the platforms targeted for deployment as well as on the operating environments. This information is usually part of the initial requirements for your application and includes:

- The authentication schemes supported on the targeted platforms or required by the targeted operating environments.
- The user identities, their authentication schemes and categories, defined on the targeted platforms.

In most cases, this information is not relevant until the configuration of your application for deployment. But, in some rare cases, an application may have to implement specific behavior based on this information.

Note – User authentication can only be performed by, or on behalf of, user-facing applications.

Note – For security reasons, when developing a web application, you should always favor session-scope authentication over global authentication.

3.2.2.1 Configuring 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)

You, as a web application developer, must configure the authentication mechanism to be used when accessing protected web resources in the deployment descriptor of your web application. The configured authentication mechanism will be used by the web container to authenticate any user attempting to access a protected resource of your web application. Only one such mechanism can be configured for a web application. This mechanism will apply for all user identities regardless of their categories or their authentication scopes. That is, it will apply for both global authentication of card-holder-users and session-scoped authentication of card-holder-users and other-users.

Note – The authentication mechanism configured in the deployment descriptor of a web application is not used for any other resources but web resources. If the application is exposing some SIO-based service to other on-card applications this mechanism does not apply.

For example, the Transit POS application is configured to require HTTP Basic Authentication for any user accessing one of its protected web resources.

EXAMPLE 3-1 Login Configuration of the Transit POS Web Application (`WEB-INF/web.xml`)

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>transit</realm-name>
</login-config>
```

Configuring The Java Card Platform-specific Authentication

In addition to the standard password-based authentication methods you are used to as a web application developer, the Java Card platform provides a mechanism that supports various authentication schemes, such as biometric, which have requirements beyond those of simple user name and password.

Support for biometric authentication schemes is optional on Java Card platform implementations, but the Java Card-specific authentication mechanism can also be used for PIN and password authentication. For PIN and password authentication, the Java Card-specific authentication mechanism provides services strictly equivalent to that of standard form-based authentication.

As an example, the Transit Admin application is configured to perform PIN authentication using Java Card-specific authentication mechanism for any user accessing one of its protected web resources.

EXAMPLE 3-2 contains an example of an HTML form for PIN-based authentication.

EXAMPLE 3-2 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" />
  <input type="submit" value="Login" />
</form>
```

EXAMPLE 3-3 contains an example of web application deployment descriptor configuration for PIN-based authentication.

EXAMPLE 3-3 Web Application Deployment Descriptor Configuration For PIN-based Authentication

```
<login-config>
  <auth-method>JC-pin</auth-method>
  <realm-name>transit</realm-name>
  <form-login-config>
    <form-login-page>/pin-based-login.html</form-login-page>
    <form-error-page>/login-error.html</form-error-page>
  </form-login-config>
</login-config>
```

Securing User Authentication

As a web application developer, you must be aware of the respective weaknesses of the authentication methods supported by the web container:

- HTTP Basic Authentication is not a secure authentication protocol. User passwords are sent in clear (only base64 encoded). Additionally, HTTP Basic credentials are stored within web browsers and are sent again to the server upon every single request to the same server and security realm. HTTP Basic Authentication does not provide any mechanism for explicit session expiration. It may require exiting the web browser in order to terminate an authentication session.

- HTTP Digest Authentication is similar to HTTP Basic Authentication but transmits the password in an encrypted form.
- Form-based Authentication allows you to control the look and feel of the authentication procedure but still the user name and password are transmitted in clear as request parameters. It is strongly advised that you use HTTP POST method to submit these authentication credentials.
- Java Card platform-specific Authentication allows you to control the look and feel of the authentication procedure and the collection of the biometric credentials. It is strongly advised to use HTTP POST method to submit these authentication credentials.

It is strongly advised that you perform authentication over secure connections unless an eavesdropping-proof point-to-point network configuration is assumed (such as between the card and its embedding device). You may additionally configure secure connections with clients to require the authentication of the clients, resulting in a security-strengthened user authentication protocol. See Chapter 2 for more details on off-card client authentication.

For example, the Transit POS web application specifies user data constraints for each protected resource that require confidentiality of the data transmitted over the connections from the web client.

EXAMPLE 3-4 Authorization Constraint of the Transit POS Web Application (`WEB-INF/web.xml`)

```
<security-constraint>
    <display-name>Card Holder Access</display-name>
    <web-resource-collection>
        <web-resource-name>CreditAndBalance</web-resource-name>
        <description>Credit and Balance URLs</description>
        <url-pattern>/balance</url-pattern>
        <url-pattern>/credit</url-pattern>
        <url-pattern>/history</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <description>Card Holder PIN</description>
        <role-name>OWNER-POS</role-name>
    </auth-constraint>
    <user-data-constraint>
        <description>Encryption</description>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
<security-constraint>
```

3.2.2.2 Coding Application-managed Authentication

In some cases, user authentication has to be performed programmatically by an application. Applet applications, extended or classic, must manage user authentication programmatically in all cases. Web applications may sometimes resort to application-managed authentication depending on the use cases to be implemented.

To programmatically authenticate a user using a particular authentication scheme, you, as the application developer, must proceed as follows:

- Retrieve the authenticator from the service registry
- Initiate an authentication session:
  - Check that the authenticator indeed implements the expect scheme and cast it to the expected scheme-specific authenticator interface (`SharedPINAuth`, `SharedPasswordAuth` or `SharedBioTemplateAuth`). Additionally, in the case of a biometric authenticator, the type of the biometric authentication method may be checked by calling the `SharedBioTemplateAuth.getBioType` method.
  - Validate the user's credentials.

To terminate an authentication session, you must reset the authentication status of the authenticator.

If your application implements application-managed authentication it is responsible for securely collecting the user's credentials according to some standard or custom protocol. The same security advises apply to application-managed authentication as apply to container-managed authentication.

EXAMPLE 3-5 shows how the Transit Turnstile Applet application manages programmatically the authentication of the turnstile administrator. At initialization time, it retrieves the authenticator. The applet can only be selected if the PIN has not already been blocked by too many failed authentication attempts. When receiving a `VERIFY` APDU command from the turnstile, it verifies the provided PIN; this marks the beginning of an authentication session.

Subsequent turnstile requests will be processed as long as the turnstile user is authenticated. When deselected, the applet resets the authentication status and, therefore, ends the authentication session. This code makes no assumption on the type of authenticator being used, either session-scoped or global. A session-scoped authenticator is nevertheless best suited as it will avoid the sharing of the

authentication status with other applications concurrently running on the card. The authenticator to be used may be configured through the setting of an application property in the runtime deployment descriptor of the application.

EXAMPLE 3-5 Application-managed Authentication for the Transit Turnstile Applet Application (`TurnstileApplet.java`)

```
static final String ADMIN_MAPPING_PROPERTY = "ADMIN-URI";
private SharedPINAuth adminAuthenticator;

protected TransitApplet(byte[] bArray, short bOffset, byte
bLength) {
    ...
    String pinAuthURI
      = JCSystem.getAppProperty(ADMIN_MAPPING_PROPERTY);
    if (pinAuthURI != null) {
        ServiceRegistry registry
          = ServiceRegistry.getServiceRegistry();
        Authenticator authenticator
          = (Authenticator) registry.lookup(pinAuthURI);
        if (authenticator != null
              && authenticator instanceof SharedPINAuth) {
            adminAuthenticator = (SharedPINAuth) authenticator;
        }
    }
    ...
}
public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    if (buffer[ISO7816.OFFSET_INS] == VERIFY) {
        verify(apdu);
    } else {
        ...
    }
}
```

EXAMPLE 3-5 Application-managed Authentication for the Transit Turnstile Applet Application (`TurnstileApplet.java`)

```
public boolean select() {
    // Decline to be selected if the PIN authenticator is blocked.
    if (adminAuthenticator.getTriesRemaining() == 0) {
        return false;
    }
    return true;
}
public void deselect() {
    // Reset the PIN authenticator at the end of the session
    adminAuthenticator.reset();
    ...
}
private void verify(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    if (adminAuthenticator == null
        || adminAuthenticator.check(buffer,
             ISO7816.OFFSET_CDATA, numBytes) == false) {
        ISOException.throwIt(SW_VERIFICATION_FAILED);
    }
}
```

Note that authenticators are SIO-based authentication services that may be either registered by the Java Card RE or by an authorized application such as a card management application. When an authenticator is registered, and therefore owned, by an other application, an application which implements application-managed user authentication must transfer the ownership of the user's credentials (or a copy thereof) to be checked (e.g. a byte array containing a PIN or a `String` object containing a password) to that other application (unless it is, as in the case of an APDU byte array, a global array). Otherwise, that other application will be unable to accessed the provided user's credentials. EXAMPLE 3-6 provides an example of how this can be handled.

EXAMPLE 3-6 Transferring A Credential Object to An Authenticator

```
SharedPINAuth adminAuthenticator = ...;
String adminPIN = ...;
byte[] pin = (adminPIN).getBytes();
if (JCSystem.getServerURI(adminAuthenticator) != null) {
  JCSystem.transferOwnership(pin, adminAuthenticator);
}
if (adminAuthenticator.check(pin, (short) 0, (byte) pin.length)) {
  ...
}
```

Programmatically Terminating A Container-managed Authentication Session

In some situations, if you are developing a web application you may rely on container-managed authentication to initiate an authenticated session but may resort to controlling the termination of the authenticated session programmatically as follows:

1. Resetting the HTTP session tracking. The HTTP session must be invalidated by calling the `HttpSession.invalidate` method. This will indirectly result in the followings:
   - The termination of the binding to the HTTP session of the session-scoped authenticator instance that was used for authentication
   - The invalidation of any authentication session tracking information associated with the HTTP session such those of Form-based or Java Card-specific authentication schemes.
2. Invalidating the authenticated identity:
   - If the authenticator corresponding to the authenticated identity is global, that authenticator must be reset by calling the `Authenticator.reset` method.
   - If the authenticator corresponding to the authenticated identity is session-scoped, the binding to the HTTP session of the particular authenticator instance that was used for authentication is terminated when invalidating the HTTP session. See previous step.

Caution – When using HTTP Basic authentication, the credentials of the authenticated user are kept in the web client, typically a web browser. To terminate an HTTP Basic authentication session and clear the user's credentials maintained by a web browser, the web browser must either be shut down or the application has to force the user through a logout procedure, such as by login as a "dummy user," also known as an "exit user."

You may use an `HttpSessionListener` that will not only get invoked when a session is programmatically invalidated by a call to the `HttpSession.invalidate` method, but will also be invoked when the session times out.

EXAMPLE 3-7 shows how the Transit POS application programmatically manages the termination of the card holder authorization session by resetting the authentication status of the associated global card holder user. The authenticator is reset when the session is invalidated either upon time out or programmatically.

EXAMPLE 3-7 Application-managed Authentication for the Transit Turnstile Applet Application (`web.xml`, `POSServlet.java` `SessionListener.java`)

```
<listener>
  <description>Performs global card holder authorization
logout</description>
  <listener-class>
    com.sun.jcdemo.transit.pos.SessionListener
  </listener-class>
</listener>
<session-config>
  <session-timeout>1</session-timeout>
</session-config>
```

```
private Authenticator cardHolderAuthenticator;

public void doPost(HttpServletRequest request, HttpServletResponse
response) {
  String path = request.getServletPath();
  HttpSession session = request.getSession(true);
```

```
  if (path.equals("/cancel-remote-admin")) {
    if (cardHolderAuthenticator != null) {
      cardHolderAuthenticator.reset();
    }
    ...
  } else if (path.equals("/logout")) {
    if (session != null) {
      session.invalidate();
    }
    ...
  }
}
```

```
public class SessionListener implements HttpSessionListener {

  public void sessionDestroyed(HttpSessionEvent hse) {
    ServletContext context
      = hse.getSession().getServletContext();
    Authenticator authenticator = (Authenticator) context
      .getAttribute(POSServlet.GLOBAL_CH_AUTHENTICATOR_ATTR);
    if (authenticator != null) {
      authenticator.reset();
    }
  }
}
```

3.2.2.3 Implementing An Application-defined Authenticators

Authenticators are plain SIOs. Registration and invocation of authenticators are subject to the same access control policies as other SIOs. They may implement user or on-card client application authorization as any other SIO (see Section 3.2.3.1, "Coding Programmatic User Role-based Security Checks" on page 3-22 and Section 4.2.2.1, "Coding Programmatic Peer Application Security Checks" on page 4-11).

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 `owner-admin` user `sio:///standard/auth/user/session/transit/owner-admin/pin`, for example, may only be authorized through the `/transit/pos` web application if the `owner-pos` card-holder-user `sio:///standard/auth/holder/global/transit/owner-pos/pin` is already authenticated. The administrative functions of an authenticator can also be protected in a similar way.

Authenticators must nevertheless implement specific behavior depending on whether they are used for global authentication or session-scoped authentication.

Implementing A Global Authenticator

For a global authenticator, you should implement a `ServiceFactory` that returns the same `Authenticator` instance when invoked by the service registry. This single authenticator instance will encapsulate the same credentials and the common global

authentication state. EXAMPLE 3-8 shows an example of a global card holder PIN authenticator. You may easily apply the same design pattern to other types of authenticators, such as to a password or a fingerprint authenticator..

EXAMPLE 3-8 A Global Card Holder PIN Authentication Service Factory

```
@TransactionType(TransactionTypeValue.NOT_SUPPORTED)
public class GlobalPINAuthFactory implements ServiceFactory {
  private final OwnerPIN ownerPIN;//Common credentials and authentication state
  private final GlobalPINAuth auth;

  public GlobalPINAuthFactory(byte[] pin, int tryLimit) {
    ownerPIN = new OwnerPIN((byte) tryLimit, (byte) pin.length);
    ownerPIN.update(pin, (byte) 0, (byte) pin.length);
    auth = new GlobalPINAuth();
  }

  public Shareable create(String serviceURI, Object parameter)
   throws SecurityException {
    return auth; // Returns the same instance each time
  }

  public OwnerPIN getOwnerPIN() {
    return ownerPIN;
  }
```

EXAMPLE 3-8 A Global Card Holder PIN Authentication Service Factory

```
  // This is a wrapper/proxy that provides synchronized access to the common
  // credentials and authentication state (ownerPIN).
  @TransactionType(TransactionTypeValue.SUPPORTS)
  private class GlobalPINAuth implements SharedPINAuth {

    @TransactionType(TransactionTypeValue.NOT_SUPPORTED)
    public synchronized boolean check(byte[] pin, short offset, byte length) {
      return ownerPIN.check(pin, offset, length);
    }

    public synchronized byte getTriesRemaining() {
      return ownerPIN.getTriesRemaining();
    }

    public synchronized boolean isValidated() {
      return ownerPIN.isValidated();
    }

    @TransactionType(TransactionTypeValue.NOT_SUPPORTED)
    public synchronized void reset() {
      ownerPIN.reset();
    }
  }
}
```

Implementing A Session Authenticator

For a session-scoped authenticator, you should implement a `ServiceFactory` that returns a different `Authenticator` instance upon each call from the service registry. Each authenticator instance must encapsulate the same credentials but a distinct authentication state. They must also share a common *try limit counter* so that failed

attempts may be counted globally. EXAMPLE 3-9 shows an example of a session password authenticator. Here also, you may easily apply that same design pattern to other types of authenticators, such as to a PIN or a fingerprint authenticator.

EXAMPLE 3-9 A Session Password Authentication Service Factory

```
@TransactionType(TransactionTypeValue.NOT_SUPPORTED)
public class SessionPasswordAuthFactory implements ServiceFactory {

  private final OwnerBioTemplate ownerPassword; // Common credentials

  public SessionPasswordAuthFactory(String password, int tryLimit) {
    ownerPassword
      = BioBuilder.buildBioTemplate(BioBuilder.PASSWORD, (byte) tryLimit);
    byte[] pwdBytes = password.getBytes();
    ownerPassword.init(pwdBytes, (byte) 0, (byte) pwdBytes.length);
    ownerPassword.doFinal();
  }

  public Shareable create(String serviceURI, Object parameter)
   throws SecurityException {
    return new SessionPasswordAuth(); // Returns a new instance each time
  }

  public OwnerBioTemplate getOwnerPassword() {
    return ownerPassword;
  }
```

EXAMPLE 3-9 A Session Password Authentication Service Factory

```
  // This is a wrapper/proxy that provides synchronized access to the common
  // credentials (ownerPassword) but maintains a distinct authentication state.
  @TransactionType(TransactionTypeValue.SUPPORTS)
  private class SessionPasswordAuth implements SharedPasswordAuth {
    private TransientReference<Boolean> validated
     = new TransientReference<Boolean>(false);

    @TransactionType(TransactionTypeValue.NOT_SUPPORTED)
    public boolean check(String password) {
      validated.set(false);
      synchronized (ownerPassword) {
        try {
          byte[] pwdBytes = password.getBytes();
          ownerPassword.initMatch(pwdBytes, (byte) 0, (byte) pwdBytes.length);
          validated.set(ownerPassword.isValidated());
        } catch (BioException be) {
        } finally {
          ownerPassword.reset();
        }
      }
      return validated.get();
    }

    public byte getTriesRemaining() {
      synchronized (ownerPassword) {
        return ownerPassword.getTriesRemaining();
      }
    }

    public boolean isValidated() {
      return validated.get() != null && validated.get();
    }

    @TransactionType(TransactionTypeValue.NOT_SUPPORTED)
    public void reset() {
      if (validated.get() != null && validated.get()) {
        validated.set(false);
      }
    }
  }
}
```

Registering An Authenticator

Authenticators are regular SIO-based services that must be registered with the service registry. Authenticators registered under the namespace `sio:///standard/auth/holder/global/` must support global card-holder-user authentication. Authenticators registered under the namespaces `sio:///standard/auth/holder/session/` and `sio:///standard/auth/user/session/` must support session authentication. Any application may register an authentication service provided it has been granted the permission to do so.

EXAMPLE 3-10 shows how to register an authenticator service.

EXAMPLE 3-10 Registering A Global Card Holder PIN Authentication Service Factory

```
ServiceRegistry registry = ServiceRegistry.getServiceRegistry();

String authenticatorURI
  = JCSystem.getAppProperty(OWNER_ADMIN_MAPPING_PROPERTY);
String adminAuthPIN
  = JCSystem.getAppProperty(OWNER_ADMIN_PIN_PROPERTY);

registry.register(authenticatorURI,
  new GlobalPINAuthFactory(adminAuthPIN.getBytes(), 3));
```

3.2.3 Implementing User Authorization

When developing a web application, you should always favor declarative security over programmatic security for protected web resources. Security-aware web applications may use programmatic security checks when declarative security alone is not sufficient to express the security model of the application.

Note – Declarative security for SIO-based resources is not currently supported by the Java Card platform.

3.2.3.1 Coding Programmatic User Role-based Security Checks

Use the following methods to implement programmatic user role-based security checks:

- the `isUserInRole` method of the `JCSystem` class determines if an authenticated identity (either associated to the current session or global) is in a specified security role as defined in the Java Card Platform-specific application descriptor for SIO and event resources. User role-based security for SIO-based services and events may only be implemented using programmatic security.
- the `isUserInRole` method of the `HttpServletRequest` interface determines if an authenticated identity associated with the current HTTP request is in a specified security role as defined in the web application's deployment descriptor for web resources.

Programmatic Security Checks for Web Resources

Invoke `HttpServletRequest.isUserInRole` to check if the authenticated identity associated with the current HTTP request is in a specified security role as defined in the web application's deployment descriptor for web resources. The identity being checked is the identity of the web application user that submitted the request. If the user is not authenticated, this method returns false. This method can only be invoked within the scope of an HTTP request handling method.

EXAMPLE 3-11 gives an example of the usage of the `HttpServletRequest.isUserInRole` to support a guest user with limited access to the balance information. The security constraints set for the `/balance` web resource grants access to both the main user role and the guest role. However, the guest user only has a restricted and limited view of the balance information.

EXAMPLE 3-11 Fictitious Example For the Transit POS Application That Would Allow For A Guest User (`WEB-INF/web.xml, ServletPOS.java`)

```
<security-constraint>
    <web-resource-collection>
        <url-pattern>/balance</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>OWNER-POS</role-name>
        <role-name>GUEST-POS</role-name>
    </auth-constraint>
    ...
<security-constraint>
```

```
private static String USER_ROLE = "OWNER-POS";
private static String GUEST_USER_ROLE = "GUEST-POS";

public void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
  String path = request.getServletPath();

  if (path.equals("/balance")) {

    int balance = ticketBook.getBalance();
    if (request.isUserInRole(GUEST_ROLE)) {
      // Display limited balance information
    } else if (request.isUserInRole(USER_ROLE)) {
      // Display full balance information
    } else {
      // Misconfiguration
    }
  }
  ...
}
```

Programmatic Security Checks for SIO-based Resources

Use the `JCSystem.isUserInRole` method to determine if a globally authenticated identity or a session-scoped authenticated identity is in a specified security role as defined in the Java Card platform-specific application descriptor. Because some identities are global, this method may be invoked from outside the scope of any

particular HTTP request handling method. However, role-based security on a session-scoped identity can only be checked with this method from within the scope of an HTTP request method[2], where the session-scoped identity is valid. For example, if invoked from within the `init` method of a servlet, this method would only return true if there is a global identity authenticated in the role being checked.

EXAMPLE 3-12 shows the programmatic user role-based security checks performed by the Transit POS web application to protect access to some of the ticket book service functions.

2. Note that the Java Card platform does not currently provide means for binding session-scoped identities to applet selection session. Therefore, invoking the `JCSystem.isUserInRole` method on role solely mapped to a session-scoped identity from within the scope of an APDU command handling method will always return false.

EXAMPLE 3-12 Programmatic User Role-based Security Checks of the Ticket Book Service (TicketBook.java)

```
private static final String ADMIN_USER_ROLE = "ADMIN";
private static final String USER_ROLE = "OWNER-POS";

public int getBalance() throws SecurityException {
    if (JCSystem.getClientURI() != null) {
      if (!JCSystem.isClientInRole(ADMIN_CLIENT_ROLE, serviceURI)
            || !JCSystem.isUserInRole(ADMIN_USER_ROLE)) {
        throw new SecurityException();
      }
    }
    return balance;
}

public long[][] getHistory(long[][] buffer, int start) {
    if (JCSystem.getClientURI() != null) {
      if (!JCSystem.isUserInRole(USER_ROLE)
          && !(JCSystem.isUserInRole(ADMIN_USER_ROLE) &&
               JCSystem.isClientInRole(ADMIN_CLIENT_ROLE,
"/ticketbook"))) {
          throw new SecurityException();
      }
    }
    ...
    return buffer;
}

public void unblock() {
    if (!(JCSystem.isClientInRole(ADMIN_CLIENT_ROLE, serviceURI)
          && JCSystem.isUserInRole(ADMIN_USER_ROLE))) {
        throw new SecurityException();
    }
    ...
}
```

3.2.3.2 Defining Declarative User Authorization Constraints

If you are developing a web application, you may use declarative security to express security constraints on certain resources of your web application. Web security constraints are declared in the deployment descriptor of web applications for constraints on web resources.

For example, the Transit POS web application and the Transit Admin web application define various user authorization constraints in their web application deployment descriptors. TABLE 3-3 lists for both of these applications the protected web resources and the user security roles permitted to access these resources.

TABLE 3-5 Protected Web Resources of the Transit POS and Transit Admin Web Applications and the User Security Roles Permitted to Access These Resources

Application	Protected Web Resource(s)	User Security Role
Transit POS web application	`/balance` (balance operation) `/history` (history operation) `/credit` (credit operation)	OWNER-POS
	`/OwnerAuthorization.html` (card holder authorization operation)	OWNER-ADMIN
Transit Admin web application	`/balance` (balance operation) `/history` (history operation) `/unblock` (unblock operation)	ADMIN

EXAMPLE 3-13 shows an authorization constraint that protects with the user security role `OWNER-POS` the `/balance`, `/history` and `/credit` web resources when accessed via a GET or a POST HTTP method.

EXAMPLE 3-13 Authorization Constraint of the Transit POS Web Application (`WEB-INF/web.xml`)

```
<security-constraint>
    <display-name>Card Holder Access</display-name>
    <web-resource-collection>
        <web-resource-name>CreditAndBalance</web-resource-name>
        <description>Credit and Balance URLs</description>
        <url-pattern>/balance</url-pattern>
        <url-pattern>/credit</url-pattern>
        <url-pattern>/history</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <description>Card Holder PIN</description>
        <role-name>OWNER-POS</role-name>
    </auth-constraint>
    ...
<security-constraint>

<security-constraint>
    <display-name>OwnerAuthorization</display-name>
    <web-resource-collection>
        <web-resource-name>Owner Authorization
Resource</web-resource-name>
        <description>Require Card Holder Authorization for remote
access</description>
        <url-pattern>/OwnerAuthorization.html</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <description>Card Holder Global Auth</description>
        <role-name>OWNER-ADMIN</role-name>
    </auth-constraint>
    ...
<security-constraint>
```

Caution – The Java Card platform version of the Servlet specification imposes the following restriction on security constraints: the same `url-pattern` and `http-method` value pair must not appear in multiple constraints. This restriction also applies to security constraints which do not have an `http-method` element because it stands for all the possible values of the `http-method` element. Therefore, the web container on a Java Card Platform implementation rejects applications

declaring in their deployment descriptors multiple security constraints with the same `url-pattern` and `http-method` value pair, as this is considered as a conflicting declaration. For example, adding the security constraint from EXAMPLE 3-13 to those listed in EXAMPLE 3-14 or EXAMPLE 3-15 would result in the application being rejected since there is more than one security constraint for the URL `/balance` and HTTP method `GET`.

EXAMPLE 3-14 Fictitious Authorization Constraint That Would Conflict If Added To Those of EXAMPLE 3-13 (`WEB-INF/web.xml`)

```
<security-constraint>
    <display-name>Card Holder Guest Access</display-name>
    <web-resource-collection>
        <web-resource-name>Balance</web-resource-name>
        <description>Balance URL</description>
        <url-pattern>/balance</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <description>Card Holder PIN</description>
        <role-name>GUEST-POS</role-name>
    </auth-constraint>
<security-constraint>
```

EXAMPLE 3-15 Fictitious Authorization Constraint That Would Conflict If Added To Those of EXAMPLE 3-13 (`WEB-INF/web.xml`)

```
<security-constraint>
    <display-name>Card Holder Guest Access</display-name>
    <web-resource-collection>
        <web-resource-name>Balance</web-resource-name>
        <description>Balance URL</description>
        <url-pattern>/balance</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <description>Card Holder PIN</description>
        <role-name>GUEST-POS</role-name>
    </auth-constraint>
<security-constraint>
```

3.2.3.3 Coding Application-managed Authorization

To determine if a card-holder-user is globally authenticated, you must locate the corresponding authenticator and check its authentication status by invoking its `Authenticator.isValidated` method.

In the case of a session-scoped authenticated identity, locating the authenticator from the registry and checking its authentication status would not work since a different instance is returned by the registry each time it is looked up. Therefore, when using session-scoped authentication, you must maintain a reference to the authenticator instance in your application's code and you must check the authentication status on that instance.

EXAMPLE 3-16 shows how the Transit Turnstile Applet application programmatically grants access to the "Let Through" turnstile operation to the turnstile administrator (the ticket book clerk). This relies on the turnstile administrator being previously authenticated as illustrated in EXAMPLE 3-5.

EXAMPLE 3-16 Application-managed Authorization for the Transit Turnstile Applet Application (`TurnstileApplet.java`)

```
private SharedPINAuth adminAuthenticator;

private void processRequest(APDU apdu) {
  byte[] buffer = apdu.getBuffer();
  switch (buffer[TLV_TAG_OFFSET]) {
  case PROCESS_ENTRY:
    ...
  case PROCESS_EXIT:
    ...
  case PROCESS_LET_THROUGH:
    offset = processLetThrough(buffer, TLV_VALUE_OFFSET,
buffer[TLV_LENGTH_OFFSET]);
    break;
  default:
    ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
  }
  ...
}

private void processLetThrough(byte[] buffer, short offset, short
length) {
  // Check access authorization
  if (!adminAuthenticator.isValidated()) {
    ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
  }
  ...
}
```

3.2.3.4 Defining Card Holder Authorization Constraint

If your web application is intended to be accessed remotely and therefore requires card holder authorization, you must include in the Java Card Platform-specific application descriptor of your application a `card-holder-authorization`

element. That element must have one or more `role-name` sub-elements designating role names. During the configuration of your application for deployment onto an actual platform, these role names will have to be mapped to card-holder-user identities.

For example, the Java Card Platform-specific application descriptor (`META-INF/javacard.xml`) of the Transit Admin web application includes the element listed in EXAMPLE 3-17.

EXAMPLE 3-17 Card Holder Authorization Constraint of the Transit Admin Web Application (`META-INF/javacard.xml`)

```
<card-holder-authorization>
    <role-name>OWNER-ADMIN</role-name>
</card-holder-authorization>
```

This element of configuration of the Transit Admin web application declares that this application is accessible from a non-card holder-facing client (a remote client) under the condition that the card holder is currently authenticated and that the corresponding card-holder-user identity is in the `OWNER-ADMIN` role.

Implementing Proactive Card Holder Authorization

In its simplest implementation the mechanism for card holder authorization requires that the card holder must already (proactively) be authenticated. Therefore, the card holder must have been previously authenticated in a role required for the web application to be accessed. If the card holder is not already authenticated in any of the required roles, an error is returned to the remote client.

Proactive card holder authorization only requires that you map the card holder authorization role to a global card holder user. That global card holder user may be authenticated prior to the remote user accessing the remotely accessible application through another locally accessible application.

For example, the card holder user that authorizes the access to the Transit Admin web application is authenticated by the container when the card holder accesses a dedicated resource (the `/OwnerAuthorization.html` page) of the Transit POS web application, see EXAMPLE 3-18.

EXAMPLE 3-18 Authorization Constraint of the Transit POS Web Application (`WEB-INF/web.xml`)

```
<security-constraint>
  <display-name>OwnerAuthorization</display-name>
  <web-resource-collection>
    <web-resource-name>Owner Authorization
Resource</web-resource-name>
    <description>Require Card Holder Authorization for remote
access</description>
    <url-pattern>/OwnerAuthorization.html</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>Card Holder Global Auth</description>
    <role-name>OWNER-ADMIN</role-name>
  </auth-constraint>
  <user-data-constraint>
    <description>encryption</description>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

In a mobile ticketing environment, the typical scenario goes as follows:

1. The transit card holder has blocked his card or is experiencing some difficulties with his card. The transit card holder requests the help of the ticket booth clerk.
2. The ticket booth clerk requests the authorization from the transit card holder to access the Transit Admin application from his terminal (a non-card holder-facing terminal).
3. The transit card holder grants the authorization to the ticket booth clerk by accessing and authenticating to the `/OwnerAuthorization.html` page of the Transit POS application from his own mobile (the trusted card holder-facing terminal).
4. The ticket booth clerk can access and authenticate to the Transit Admin application from his terminal.
5. The transit card holder signs off from the card holder authorization identity, such as through one of the mechanisms described in Programmatically Terminating A Container-managed Authentication Session, to end the authorization session, hence preventing any other remote access.

Proactive card holder authorization is easy to set up but suffers from the use of a global identity. Once the card holder authorization is granted for an application:

- It is granted for any potential remote user concurrently accessing that application.
- It is also granted for any other applications that may be using that same identity for card holder authorization.
- It may interfere with any other application that may be concurrently running and using that same identity.

One way to alleviate the two latter issues is for you to use a global card holder identity dedicated to your application.

Implementing Reactive Card Holder Authorization

A card holder authorization role may be mapped to a session-scoped card holder user to allow for the card holder authorization request to be reactive, meaning only to be triggered when a remote user is attempting to access the application, and to limit this authorization to that particular access. This configuration, though, is not as easy to set up as that for proactive card holder authorization.

You may implement a special-purpose, privileged, application named a *user-facing card holder authentication delegate* that registers an authenticator SIO for a specific card holder identity. This delegate, when its `isValidated` method is invoked, may prompt the card holder if he is not already authenticated as this identity. You may implement such an application, for example, as a Java ME MIDP application or as a web application using Dynamic HTML (DHTML) that can pro-actively prompt the end-user.

If a card holder authorization role name of a non-card holder-facing web application is mapped to an authenticator URI managed by such an authentication delegate, an attempt to access the web application by a remote user will result in the delegate application being invoked through the `isValidated` method of its authenticator SIO, and the card holder being prompted for authentication if he is not already authenticated as the required identity. If the card holder is already authenticated or successfully authenticates, the `isValidated` method of the authenticator SIO will return true and the card holder authorization will be granted for the duration of the HTTP session with the remote end-user. If the authenticator is registered as a session authenticator, the authorization will only be granted to the current HTTP session.

EXAMPLE 3-19 shows a simple example of an authentication delegate for a SIM card on a Java ME MIDP phone. This example extends the examples from EXAMPLE 3-8 and EXAMPLE 3-10.

EXAMPLE 3-19 A PIN Authentication Delegate

```
public class PINAuthenticationDelegate implements SharedPINAuth {
  private SharedPINAuth actual;

  public PINAuthenticationDelegate(SharedPINAuth actual) {
    this.actual = actual;
  }

  public boolean isValidated() {
    if (!actual.isValidated() && actual.getTriesRemaining() > 0) {
      Connection c
       = Connector.open("ssl://embeddingPhone:22222");
      // 1- MIDlet is activated by Push registry
      // 2- initiate dialog with MIDlet
      // 3- MIDlet prompts user for PIN
      // 4- MIDlet sends PIN back through secure connection
      actual.check(pin, ...);
    }
    return actual.isValidated();
  }
}
```

In the same mobile ticketing environment as described in the previous section, the typical scenario goes as follows:

1. The transit card holder has blocked his card or is experiencing some difficulties with his card. The transit card holder requests the help of the ticket booth clerk.
2. The ticket booth clerk attempts to access the Transit Admin application from his terminal (a non-card holder-facing terminal).
3. The access attempt from the ticket booth clerk triggers the prompting of the transit card holder for his authorization and associated credentials directly on his own mobile (the trusted card holder-facing terminal).
4. The transit card holder authenticates and grants the authorization to the ticket booth clerk.
5. When the ticket booth clerk signs off, the authorization session is automatically invalidated.

3.2.3.5 Declaring User Roles

In order to facilitate the security evaluation and the configuration by the application deployer of your application, you, as the application developer, are required to declare all the user roles your application uses in declarative web security constraints and in programmatic security checks:

- You must declare the roles referenced by declarative web user authorization constraints (security constraints) in your web application's deployment descriptor.
- You must declare the roles referenced by web programmatic user role-based security checks in your web application's deployment descriptor.
- You must declare the roles referenced by SIO-based programmatic user role-based security checks in your application's Java Card platform-specific application descriptor.
- You must declare the roles referenced by card holder authorization constraints in your application's Java Card platform-specific application descriptor.

For example, the roles referenced by the declarative web user authorization constraints of the Transit POS application, namely `OWNER-POS` and `OWNER-ADMIN` (see EXAMPLE 3-13), are declared at the end of that web application's deployment descriptor. See EXAMPLE 3-20.

EXAMPLE 3-20 Web User Roles Declaration for the Transit POS Web Application (`WEB-INF/web.xml`)

```
<security-role>
    <description>Card Holder</description>
    <role-name>OWNER-POS</role-name>
</security-role>
<security-role>
    <description>For card holder authorization</description>
    <role-name>OWNER-ADMIN</role-name>
</security-role>
```

The roles referenced by the SIO-based programmatic security checks of the Transit POS application, namely `OWNER-POS` and `ADMIN` (see EXAMPLE 3-12), are declared at the end of that web application's Java Card platform-specific application descriptor, see EXAMPLE 3-21.

EXAMPLE 3-21 All User Roles Declaration for the Transit POS Web Application (`META-INF/javacard.xml`)

```
<security-role>
    <role-name category="USER">ADMIN</role-name>
</security-role>
<security-role>
    <role-name category="USER">OWNER-POS</role-name>
</security-role>
<security-role>
    <role-name category="USER">OWNER-ADMIN</role-name>
</security-role>
```

The roles referenced by the card holder authorization constraint, namely `OWNER-ADMIN` (see EXAMPLE 3-17), are declared at the end of that web application's Java Card platform-specific application descriptor, see EXAMPLE 3-22.

EXAMPLE 3-22 All User Roles Declaration for the Transit Admin Web Application (`META-INF/javacard.xml`)

```
<security-role>
    <role-name category="USER">OWNER-ADMIN</role-name>
</security-role>
<security-role>
    <role-name category="USER">ADMIN</role-name>
</security-role>
```

Declaring Programmatic User Security Role References

The deployment descriptor of a web application allows for user role names passed as parameters to the `HttpRequest.isUserInRole` method to only be treated as indirect references to the security roles actually declared. This allows for more flexibility in changing role names in an application without having to recompile the servlet or component making the call. This feature is mostly used when a web application is assembled from reusable (already compiled) components that may use conflicting role names in their call to the `HttpRequest.isUserInRole` method. But a web application assembler is not required to use it because the web container defaults to treating the user role names passed as parameters to the `HttpRequest.isUserInRole` method as actual role names.

You may typically use this feature when assembling a Java Card web application from already certified components.

3.3 Configuring An Application's User Security

When your application is to be deployed, you, as the application deployer, must configure your application as follows:

- you must configure or re-configure, if needed, the container-managed authentication to account for the specifics of the deployment environment
- you must map the roles declared by the application developer to actual user identities on the targeted card.

3.3.1 Configuring Container-managed Authentication

If your application is a web application that uses container-managed authentication, you, as the application deployer, must configure the following sub-elements of the `login-config` element in your application's deployment descriptor:

- The `auth-method` element must be set to one of the authentication schemes supported on the targeted platforms or required by the targeted operating environments.
- The `realm-name` element must be set to the authentication realm for the set of user identities defined on the targeted platforms.

In most cases, the developer of the application will have already configured these parameters in the web application's deployment descriptor. A re-configuration of these parameters should only be done if the web application's code has no direct dependency on particular authentication scheme and authentication realm. See Section 3.2.2.1, "Configuring Container-managed Authentication" on page 3-9.

If your web application does not define a login configuration or authentication method in its deployment descriptor (`login-config` and `login-config/auth-method` elements), active authentication by the container in any of the role that your web application declared will not be performed.

See EXAMPLE 3-1 for an example of configuration of the `login-config/auth-method` and `login-config/realm-name` elements in Transit POS Web Application's deployment descriptor.

3.3.2 Mapping User Roles To User Authenticators

The user security roles declared in the Java Card platform-specific application descriptor or in the deployment descriptor of web applications (see Section 3.2.3.5, "Declaring User Roles" on page 3-35) must be mapped to authenticator URIs in the runtime descriptor of the application.

A user security role can be mapped to one or more authenticator URIs. Each authenticator URI encodes the following information:

- the category of the user: card holder or other user
- the scope of the authentication: global or session-scoped
- the authentication realm: an arbitrary path that designates a group of users
- the name of the user
- the authentication scheme: PIN, password or one of the biometric types.

An authenticator URI has one of three following formats:

`sio:///standard/auth/holder/[global|session]/[<realm>/]<user>/<scheme>`

`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.

Each authenticator URI corresponds to a user identity on the targeted platform. The authenticator URIs to which an application's security role can be mapped may be constrained by the application's design, implementation and/or configuration:

- If a user security role is intended for active authentication and authorization checking, the mapping may be constrained by the following:
  - the authentication realm configured in `login-config/realm-name` element of your web application's deployment descriptor
  - the categories of users as determined by the accessibility requirements for your application, see Section 3.1.1, "Determining The Accessibility Requirements" on page 3-3
  - the authentication method configured in the `login-config/auth-method` element of your web application's deployment descriptor.

  Additionally, the authenticator URIs to which a user security role is mapped cannot be designated by a path-prefix URI pattern.
- If a user security role is only intended for authorization checking, the mapping is not constrained and additionally, session-scoped authenticator URIs to which a user security role may be mapped may be designated by path-prefix URI patterns.

Note – The use of path-prefix URI patterns for user identities intended to be actively authenticated by the web container are not permitted because the web container would be unable to locate the one authenticator to be used for authentication. The use of path-prefix URI patterns for global user identities even intended only for authorization checking by the web container or the Java Card RE are not permitted because it would require searching the service registry for all matching authenticators.

If you do not map to an authenticator URI a user security role that exclusively protect one of the resources of your web application, no user will be able to authenticate in that role and therefore access to the resource protected by that role will never be authorized. This can be used on purpose to disable certain parts of an application when deployed in a particular operating environment where you want to limit the functionalities of your application.

If you map a user security role to an authenticator URI with a <*scheme*> or <*realm*> path component that does not match the authentication method or realm name[3] (`login-config/auth-method` and `login-config/realm-name` elements) declared for your web application, active authentication in that role by the container will not be performed. Still, if you have mapped the user security role 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.

TABLE 3-6 lists for each of the web authentication methods the corresponding <*scheme*> authenticator URI component.

TABLE 3-6 Web Authentication Methods And Corresponding Authentication Schemes

Web Authentication Method (`auth-method` element)		Authentication Scheme(s) (<*scheme*> component)	
Description	**Value**	**Description**	**Value(s)**
HTTP Basic	`BASIC`	PIN or password	`pin` `password`

3. When the `login-config/realm-name` element is not present an empty realm name is assumed.

TABLE 3-6 Web Authentication Methods And Corresponding Authentication Schemes

Web Authentication Method (`auth-method` element)		Authentication Scheme(s) (*<scheme>* component)	
HTTP Digest	`DIGEST`	PIN or password	`pin` `password`
Form-based	`FORM`	PIN or password	`pin` `password`

Java Card Specific	`JC-`*<scheme>*	PIN, password or biometric	`pin`
			`password`
			`body-odor`
			`dna-scan`
			`ear-geometry`
			`facial-feature`
			`finger-geometry`
			`fingerprint`
			`gait-style`
			`hand-geometry`
			`iris-scan`
			`keystrokes`
			`lip-movement`
			`palm-geometry`
			`retina-scan`
			`signature`
			`thermal-face`
			`thermal-hand`
			`vein-pattern`
			`voice-print`

For example, the roles declared in both the web application deployment descriptor (see EXAMPLE 3-20) and in the Java Card platform-specific application descriptor (see EXAMPLE 3-21) of the Transit POS and Transit Admin web applications are mapped in their respective runtime descriptors to specific user identities that are consistent across the two applications. The mapping of the `ADMIN` role to a plain (possibly remote) user authenticator URI does not signify that the Transit POS web application is remotely accessible. This role is for the use of the administrative functions exposed

by the ticket book service. These functions are intended to be accessed by a remote user indirectly through the Transit Admin web application, which is itself remotely accessible..

EXAMPLE 3-23 User Roles Mapping for the Transit POS Web Application (`META-INF/MANIFEST.MF`)

```
User-Role-List: OWNER-POS, OWNER-ADMIN, ADMIN
OWNER-ADMIN-Mapped-To-Auth-URI:
    sio:///standard/auth/holder/global/transit/owner-admin/pin
OWNER-POS-Mapped-To-Auth-URI:
    sio:///standard/auth/holder/session/transit/owner-pos/pin
ADMIN-Mapped-To-Auth-URI:
    sio:///standard/auth/user/session/transit/remote-admin/pin
```

EXAMPLE 3-24 User Roles Mapping for the Transit Admin Web Application (`META-INF/MANIFEST.MF`)

```
User-Role-List: OWNER-ADMIN, ADMIN
OWNER-ADMIN-Mapped-To-Auth-URI:
    sio:///standard/auth/holder/global/transit/owner-admin/pin
ADMIN-Mapped-To-Auth-URI:
    sio:///standard/auth/user/session/transit/remote-admin/pin
```

Note – The Transit POS Web and Transit Admin Web are using the same names for certain user security roles (`OWNER-ADMIN` and `ADMIN`) but they could have very well used totally unrelated names. This could have happened if, for example, the two applications had been developed by two different parties. The responsibility of the deployer is to make sure that names that represent the same functional and security roles are indeed mapped to the same group of user identities (authenticator URIs) on the platform.

CHAPTER 4

Authenticating and Authorizing On-Card Applications

Inter-application communication security on the Java Card platform is supported through *peer application authentication* and *authorization*. Peer application authentication is the process by which an application proves its identity to another application. Whereas peer application authorization is the process by which access to protected resources, such as SIO-based services and events, may be granted to another application based on its authenticated identity.

Note that an event is simply a special kind of service that is used for asynchronously notifying other applications of an occurring condition. In this chapter, the term "services" designates both proper SIO-based services, as well as SIO-based events.

On the Java Card platform, an application may be authenticated through two different paths: through the application authentication policy implemented by the card management and through peer-to-peer application authentication:

- **Card management authentication policy** - One of the responsibilities of card manager applications is to securely load and install applications according to a well-defined policy. This encompasses the following operations:

  a. authenticating an application's code to determine if it was issued by a trusted source and binding the loaded application to a protection domain that grants certain permissions

  b. assigning application identifiers (application URIs) to instances of that application upon request from a trusted card management authority

- **Peer-to-peer application authentication** - Depending on the card management authentication policy in place and other business and security requirements, an application's URI may not be sufficient to assert its identity. The application developer may therefore resort to a peer-to-peer application authentication technique. A server application can use credentials to authenticate the clients it trusts. These credentials may be symmetric or asymmetric cryptography materials, such as secret keys and public key certificates. Currently, the Java Card platform

only supports *on-card client authentication*, but clients of a server application may also authenticate the server by defining a simple application-level protocol between that server application and its clients.

When implementing peer-to-peer application authentication, an application may use a *credential manager* object to manage the credentials used for authentication. A credential manager is used both for managing the key material that is used to authenticate with peer applications and for managing the trust material that is used when making trust decisions, such as deciding whether credentials presented by a peer application should be accepted. The actual management of the authentication credentials depends on the *mode of operation* of the application, server and client. An application developer can either implement and set his own credential manager, an *application-defined credential manager*, or it can also delegate to a credential manager assigned by the card manager application, a *card manager-assigned credential manager*.

On the Java Card platform, peer application authorization is mainly supported through role-based security. Currently, the Java Card platform only supports role-based security for client applications, *on-card client role-based security*. On-card client role-based security is used by a server application to restrict access to protected application resources, including SIO-based services and events, to only those client applications that have been granted a particular security role. An *on-card client security role* is a logical grouping of client applications defined by the application developer or assembler. When the application is deployed, on-card client roles are mapped by a deployer (for example, the application provider) to actual client applications or groups of applications on the targeted platform. Role-based security provides a flexible means for an application developer to define and implement the security requirements of his application while allowing for the actual security policy to be configured upon deployment.

4.1 Designing Inter-application Communication Security

An application engages in inter-application communication whenever it uses services provided by another application or whenever it offers services to other applications.

You, as the developer of such an application, must start by analyzing the inter-application communication security requirements of your application and must gather the following information, depending on your application's mode or modes of operation:

- **Server Mode** - If your application operates in a server mode, as a server application that provides services to other applications, you must gather the following information:

- **Client Application Roles and Protected Functions** - whether certain functions of your application directly or indirectly exposed through its services must be protected so they can be accessed exclusively by certain client applications, depending on their functional roles or privileges.
- **Client Authentication and Identification Strategies** - whether a client application's URI is sufficient to assert its identity or whether client-to-server authentication is required.

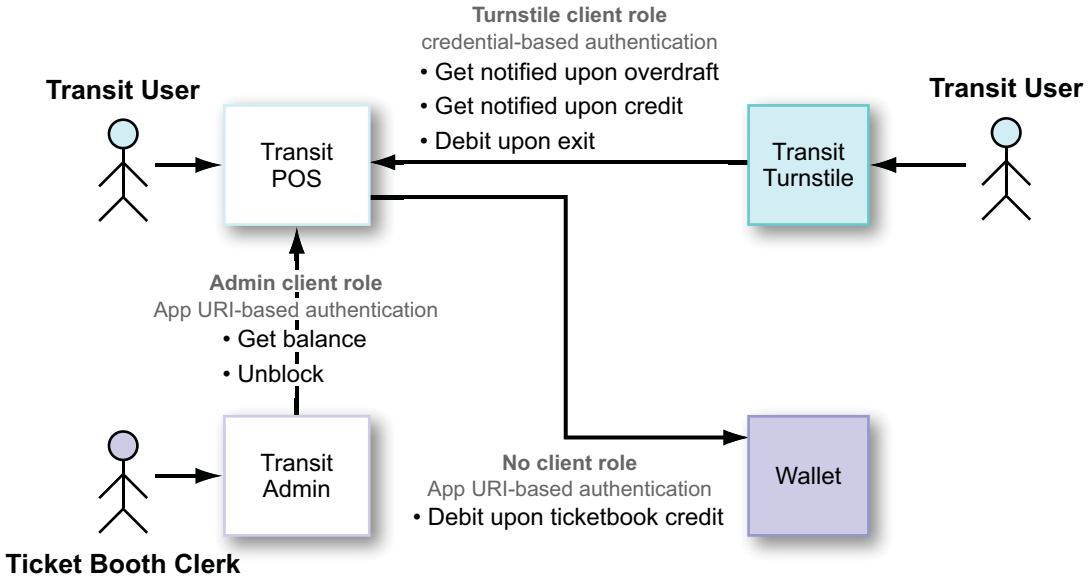
- **Client Mode** - If your application operates in a client mode, as a client application that uses services provided by other applications, you must gather the following information:
  - **Server's Client Authentication and Identification Strategies** - the strategies used by the server application to authenticate and/or identify its clients, see Server Mode above.
  - **Server Authentication and Identification Strategies** - whether the server application's URI is sufficient to assert its identity or whether server-to-client authentication is required.

When your application requires its peers to authenticate, you must additionally determine the following:

- **Authentication credential management model** - if the management of the authentication credentials must be performed by the card management-assigned credential manager (delegated model) or if the management of the authentication credentials must be performed by the application itself (ad hoc model).
- **Authentication session duration** - the type of authentication session duration required based on:
  - the sensitivity of the protected functions
  - whether the authentication credentials may be changed or revoked during the application lifetime.

The diagram in FIGURE 4-1 shows the use cases of the Transit application with highlights on the functional client roles and strategies for authentication and identification.

FIGURE 4-1 Functional Client Roles and Strategies for Authentication and Identification



4.1.1 Defining Client Application Roles

If there are certain functions or resources within your server application that require client applications to have specific privileges, you must decide the client application roles permitted access to each function or resource. Such a client application role is a logical grouping of client applications that does not designate any specific application until later in the development lifecycle when the application is deployed into an actual operating environment.

Similarly to user roles (see Section 3.1.2, "Defining The User Roles" on page 3-5), using or coding with client application roles instead of with actual client application identities (application URI) allows more flexibility in your application's configuration and will facilitate the reuse of your application or some of its components in a different setting.

For example, the Transit POS web application requires various client application security roles to be defined. TABLE 4-1 lists these client application security roles and the functions or resources they protect.

TABLE 4-1 Client Application Security Roles and Protected Functions or Resources for the Transit POS Application

Application	User Security Role	Protected Function(s) or Resource(s)
Transit POS web application	TURNSTILE-CLIENT	Ticket book debit operation
	ADMIN-CLIENT	Ticket book balance operation
		Ticket book history operation
		Ticket book unblock operation

4.1.2 Determining Peer Application Authentication and Identification Strategies

When designing a server application, you must determine how and on what basis your application will decide that it trusts other applications (client applications) that will attempt to access protected functions of the service it exposes.

Similarly, when designing a client application, you may sometimes be required to determine how and on what basis your application will decide that it trusts the server application that provides the service it wants to use.

Your application may use one of the following strategies to decide whether it trusts a peer application (client or server application):

- Based on the peer application's URI or protection domain assigned by the trusted card management application:
  - Your application may trust another application on the basis of that other application's URI. You, therefore, trust this application URI to establish the identity of that other application in accordance with a certain card management policy in place.
  - Your application may trust another application on the basis of that other application's protection domain's name. In this case you trust this protection domain's name to establish a certain form of identity or group identity in accordance with a certain card management policy in place.
- Based on peer-to-peer authentication:
  - Your application may trust another application on the basis of certain authentication credentials that it uses to establish its identity.

Currently the Java Card platform only provides an API for authenticating client applications. Nevertheless, provided the definition of some simple application-level protocol between a server application and its clients, server and also mutual authentication can be implemented. See Coding Programmatic Server Application Role-based Security Checks With A Role-Reversing CallBack SIO for a description of this design pattern.

4.1.3 Determining The Authentication Credential Management Model

The choice of an authentication credential management model is usually dictated by the operating and business environment to which an application will be deployed.

By default, the platform assumes a delegated model. In this model the management of the authentication credentials is delegated to a card management-assigned credential manager.

If the management of the authentication credentials must be performed by your application directly (ad hoc model), you must implement an ad hoc credential manager (application-defined credential manager), see Section 4.2.1, "Implementing Peer-to-peer Application Authentication and Identification" on page 4-8.

4.1.3.1 Defining Authentication Credential Aliases

If the server application you are developing is required to use authentication credentials to establish the trustworthiness of its clients, you must determine or define the way these authentication credentials are or should be named.

For client authentication, the role name passed as parameter to the `JCSystem.isClientInRole` method must be mapped to credential aliases. These credential aliases are then passed to the `getTrustedCredentials` method of the server application's applicable credential manager to retrieve the actual credentials to be used for authentication.

The credential aliases that you use in the implementation of the `getTrustedCredentials` method are part of the configuration information that you must communicate to the application deployer, see Section 4.3.2, "Configuring Client Application Authentication" on page 4-19. When the application is deployed, the application deployer may define a mapping of the declared client roles to certain or all of these credential aliases that is specific to the operating and business environment to which an application is deployed.

4.1.4 Determining Authentication Session Durations

When developing a server application, you must determine the type of authentication session duration for each of the security roles protecting functions or the resources of your application. The choice of the type of the authentication session duration may depend on the sensitivity of the security role, as well as on whether the authentication credentials of both your server application and its clients are managed dynamically or not. Another factor that may influence this choice is performance or response time constraints, such as for contactless applications.

The Java Card platform supports three types of authentication session duration:

- **Duration of the access** - authentication in a particular role is performed for each invocation of the `JCSystem.isClientInRole` method
- **Duration of the card session** - authentication in a particular role is performed once for a specific client application and specific server application during the card power up/reset session
- **Duration/lifetime of the inter-operating applications** - authentication in a particular role is performed once for a specific client application and specific server application during their common lifetime

4.2 Implementing Inter-application Communication Security

Based on your design decisions, implementing your application's inter-application communication security consists on the following:

- Implementing the way your application will authenticate and identify the peers it trusts
- Implementing the security checks that will authorize access to your application's resources to only those peers that have been granted access based on their authenticated identity

---

Note – Authentication and authorization may depend on the resources (SIO-based services and events) being accessed. These resources are designated with URIs both in the code and in descriptors. To maximize portability and reusability, you should name or refer to the application's resources both in code and descriptors using relative URIs.

4.2.1 Implementing Peer-to-peer Application Authentication and Identification

Client application authentication requires a `CredentialManager` instance be set for the mode(s) of communication (client or server) that corresponds to that of your application. Depending on that or these modes of communication, the `CredentialManager` instance has the following responsibilities:

- **Server mode** - select the authentication credentials that will determine if the authentication credentials presented by the client application should be trusted
- **Client mode** - select the authentication credentials to be presented for authentication with the server application

If you have opted for the delegated authentication credential management model, you rely on the default credential manager set by the card management application that will load your application.

If you have opted for an ad hoc authentication credential management model, you must implement an application-defined credential manager. An instance of that class must be set at runtime for the mode(s) of communication that correspond to that of your application.

4.2.1.1 Coding Ad hoc Credential Management

An application-defined credential manager must be a subclass of the `CredentialManager` class.

Authentication Credential Management By Server

For a server application, you must provide a concrete implementation of the `CredentialManager.getTrustedCredentials` method.

As mentioned earlier, the credential aliases that you may use in the implementation of this method will be part of the configuration information that you must communicate to the application deployer.

For example, the Transit POS web application, which exposes the Ticket Book service to the Transit Turnstile applet application, uses an application-defined credential manager to authenticate the Transit Turnstile application, see EXAMPLE 4-1. The

authentication credential is a DES secret key, pre-shared with the Transit Turnstile application, and is initialized with key specification data retrieved from the TURNSTILE-KEY application property.

EXAMPLE 4-1 Application-defined Credential Manager Implemented by the Transit POS Application (POSCredentialManager.java)

```
public class POSCredentialManager extends CredentialManager {
    static final String TURNSTILE_KEY_PROPERTY = "TURNSTILE-KEY";
    private DESKey turnstileKey;

    public POSCredentialManager() {
        turnstileKey = (DESKey) KeyBuilder.buildKey(
          KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES, false);
        String keyData
          = JCSystem.getAppProperty(TURNSTILE_KEY_PROPERTY);
        if (keyData != null) {
            byte[] keyDataBytes = keyData.getBytes();
            turnstileKey.setKey(keyDataBytes, (short) 0);
        }
    }
    ...
    public Object[] getTrustedCredentials(String[] aliases,
            String endpointURI, byte mode) {
        if (endpointURI.endsWith("/ticketbook")) {
            for (String alias : aliases) {
                if (alias.equals("TURNSTILE_CREDENTIAL")) {
                    return new Object[]{turnstileKey};
                }
            }
        }
        return null;
    }
}
```

An instance of this CredentialManager subclass is set for the mode CredentialManager.MODE_SIO_SERVER when the application is instantiated and before the TicketBook service is registered, see EXAMPLE 4-2.

EXAMPLE 4-2 Application-defined Credential Manager Implemented by the Transit POS Application (ContextListener.java)

```
CredentialManager.setCredentialManager(
  new POSCredentialManager(),
  CredentialManager.MODE_SIO_SERVER);
TicketBook ticketBook = new TicketBook(TICKETBOOK_URI);
ServiceRegistry registry = ServiceRegistry.getServiceRegistry();
registry.register(TICKETBOOK_URI,
  new TicketBookFactory(TicketBook.createProxy(ticketBook)));
...
```

Authentication Credential Management By Client

For a server application, you must provide a concrete implementation of the `CredentialManager.getCredentials` method.

For example, the Transit Turnstile applet application, which is a client of the Ticket Book service exposed by the Transit POS web application, uses an application-defined credential manager to authenticate with the Transit POS web application, see EXAMPLE 4-3. The authentication credential is a DES secret key, pre-shared with the Transit POS application, and which is initialized with key specification data retrieved from the `TURNSTILE-KEY` application property.

An instance of this `CredentialManager` subclass is set for the mode `CredentialManager.MODE_SIO_CLIENT` when the application is instantiated and before the TicketBook service is looked up, see EXAMPLE 4-4.

EXAMPLE 4-3 Application-defined Credential Manager Implemented by the Transit Turnstile Application (`TurnstileCredentialManager.java`)

```
public class TurnstileCredentialManager extends CredentialManager
{
    static final String TURNSTILE_KEY_PROPERTY = "TURNSTILE-KEY";
    private DESKey turnstileKey;

    public TurnstileCredentialManager() {
        turnstileKey = (DESKey) KeyBuilder.buildKey(
          KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES, false);
        String keyData
          = JCSystem.getAppProperty(TURNSTILE_KEY_PROPERTY);
        if (keyData != null) {
            byte[] keyDataBytes = keyData.getBytes();
            turnstileKey.setKey(keyDataBytes, (short) 0);
        }
    }
    ...
    public Key[] getCredentials(byte[] types, String endpointURI,
            byte mode) {
        if (endpointURI.endsWith("/ticketbook")) {
            for (byte type : types) {
                if (type == turnstileKey.getType()) {
                    return new Key[]{turnstileKey};
                }
            }
        }
    }
}
```

EXAMPLE 4-4 Application-defined Credential Manager Implemented by the Transit Turnstile Application (`ContextListener.java`)

```
CredentialManager.setCredentialManager(
  new TurnstileCredentialManager(),
  CredentialManager.MODE_SIO_CLIENT);

ServiceRegistry serviceRegistry =
ServiceRegistry.getServiceRegistry();
SharedTicketBook _ticketBook = (SharedTicketBook)
    serviceRegistry.lookup(transitPOSAppURI, "ticketbook");
```

4.2.2 Implementing Peer Application Authorization

Peer application authorization consists of coding programmatic security checks that guard access to the protected resources of your application. These security checks may use API methods that simply allow it to retrieve the identity of a peer or may use the client role-based security API.

4.2.2.1 Coding Programmatic Peer Application Security Checks

The Java Card platform provides various API methods to code programmatic security checks for client as well as server applications:

- For programmatic client application security checks, you may use the following methods:
  - the `getClientURI` method of the `javacardx.framework.JCSystem` class returns the URI identifying the proper client of an SIO invoked through a shareable interface method. If the shareable interface method was called by another application from within the same group context as that of the server application, the client corresponds to that calling application. Otherwise the client corresponds to the previously active application as determined by a call to the `JCSystem.getPreviousURI` method
  - the `getPreviousURI` method of the `javacardx.framework.JCSystem` class returns the URI of the previously active application, meaning the application URI that was associated with the current thread before it entered the current group context
  - the `getPreviousContextAID` method of the `javacard.framework.JCSystem` class returns the AID associated with the previously active applet context
  - the `isClientInRole` method of the `javacardx.framework.JCSystem` class determines if a client application as determined by the `getClientURI` method is in a specified security role as defined in the Java Card platform-specific application descriptor for SIO and event resources

These methods are typically used by a server application, while executing a shareable interface method to determine the identity or role-membership of its clients and, thereby, to control access privileges.

In addition, in an event producing application, which is a particular kind of server application, you may use the following method:

- the `notifyListenersInRole` method of the `javacardx.facilities.EventRegistry` class notifies the listeners of client applications that are in a specified security role. Note that the listeners registered by the server application itself are always notified.

- For programmatic server application security checks, you may use the following method:
  - the `getServerURI` method of the `javacardx.framework.JCSystem` class returns the URI identifying the server application owning a specified Shareable Interface Object.

  This method is typically used by a client application to determine the identity of the application providing a service and, thereby, make some trust decisions.

You should always favor role-based security checks implemented with the `isClientInRole` and `notifyListenersInRole` methods over simple security checks implemented with `getClientURI`, `getServerURI`, `getPreviousURI` and `getPreviousContextAID`. Role-based security provides additional deployment flexibility, code reusability and security tuning[1].

Coding Programmatic Client Application Role-based Security Checks

When developing a server application, you may use the `JCSystem.isClientInRole` and `EventRegistry.notifyListenersInRole` methods to determine if a client application is in a specified security role as defined in the Java Card platform-specific application descriptor.

1. Security checks implemented with the `isClientInRole` method are equivalent to security checks implemented with the `getClientURI` method when roles are mapped to application URIs.

EXAMPLE 4-5 shows the programmatic application client role-based security checks performed by the Transit POS web application to protect access to some of the ticket book service functions and events.

EXAMPLE 4-5 Programmatic Application Client Role-based Security Checks of the Ticket Book Service (`TicketBook.java`)

```
static final String TURNSTILE_CLIENT_ROLE = "TURNSTILE_CLIENT";
static final String ADMIN_CLIENT_ROLE = "ADMIN_CLIENT";

public int getBalance() throws SecurityException {
    if (!JCSystem.isSIOCallerInSameContext()) {
        if (!(JCSystem.isClientInRole(ADMIN_CLIENT_ROLE,
serviceURI)
            && JCSystem.isUserInRole(ADMIN_USER_ROLE))) {
            throw new SecurityException();
       }
    }
    return balance;
}

public int credit(int count) {
    if ((balance + count) > 0) {
        eventRegistry.notifyListenersInRole(
          new Event("ticketbook/credited", null),
          TURNSTILE_CLIENT_ROLE);
    }
    balance += count;
    ...
    return balance;
}
```

EXAMPLE 4-5 Programmatic Application Client Role-based Security Checks of the Ticket Book Service (`TicketBook.java`)

```
public int debit(int count) throws SecurityException {
    if (!JCSystem.isClientInRole(TURNSTILE_CLIENT_ROLE,
serviceURI)) {
        throw new SecurityException();
    }
    balance -= count;
    if (balance <= 0) {
        eventRegistry.notifyListenersInRole(
          new Event("ticketbook/overdraft", null),
          TURNSTILE_CLIENT_ROLE);
    }
    ...
    return balance;
}

public long[][] getHistory(long[][] buffer, int start) {
    if (!JCSystem.isSIOCallerInSameContext()) {
        if (!(JCSystem.isUserInRole(USER_ROLE)
            || (JCSystem.isUserInRole(ADMIN_USER_ROLE) &&
                JCSystem.isClientInRole(ADMIN_CLIENT_ROLE,
                                        serviceURI))) {
            throw new SecurityException();
        }
    }
    ...
    return buffer;
}

public void unblock() {
    if (!JCSystem.isSIOCallerInSameContext()) {
        if (!(JCSystem.isClientInRole(ADMIN_CLIENT_ROLE,
                                      serviceURI)
            && JCSystem.isUserInRole(ADMIN_USER_ROLE))) {
            throw new SecurityException();
       }
    }
    eventRegistry.notifyListenersInRole(
      new Event("ticketbook/unblocked", null),
      TURNSTILE_CLIENT_ROLE);
}
```

Coding Programmatic Server Application Role-based Security Checks With A Role-Reversing CallBack SIO

Currently, the Java Card platform does not support a role-based security API that would allow a client application to authenticate the server application that provides the service it is attempting to use.

Nevertheless, you, as a client application developer, may agree with the developer of the server application that provides the service you intend to use on a simple application-level protocol that will allow your client application to authenticate that server application.

Such a protocol requires a client application *C* to call a server application *S* with a dedicated SIO that the server application must call back. This results in a role reversal as the server *S* becomes the client and the client *C* becomes the server. Upon this call by server application *S*, the client application *C*'s SIO may perform a role-based security check.

EXAMPLE 4-6, EXAMPLE 4-7 and EXAMPLE 4-8 illustrate a possible example that shows programmatic application server role-based security checks performed by the Transit Turnstile applet application to ensure that the ticket book service is provided by an application it trusts.

EXAMPLE 4-6 Helper Classes For Programmatic Application Server Role-based Security Checks

```
public interface RoleReversingCallBack extends Shareable {
    boolean check();
}

public interface IdentifiableService extends Shareable {
    void identify(RoleReversingCallBack o);
}

public class AccessChecker implements RoleReversingCallBack {
    private String roleName;
    private String uri;
    private IdentifiableService service;
    private granted = false;

    public AccessChecker(String roleName, String uri,
      IdentifiableService service) {
        this.uri = uri;
        this.service = service;
        this.roleName = roleName;
    }

    public synchronized boolean isGranted() {
        granted = false;
        service.indentify(this);
        return granted;
    }

    public synchronized void check() {
        if (!JCSystem.getClientURI()
          .equals(JCSystem.getServerURI(service))) {
           throw new SecurityException();
        }
        granted = JCSystem.isClientInRole(roleName, uri);
    }
}
```

EXAMPLE 4-7 Ticket Book Service Identification For Application Programmatic Application Server Role-based Security Checks

```
public class TicketBook implements SharedTicketBook,
SharedTicketBookControl, IdentifiableService {

    public void identify(RoleReversingCallBack o) {
       o.check();
    }
    ...
}
```

EXAMPLE 4-8 Turnstile Applet Application Invocation Of Application Programmatic Application Server Role-based Security Checks

```
public class TurnstileApplet extends Applet implements AppletEvent
{
    static final String POS_SERVER_ROLE = "POS_SERVER";

    private SharedTicketBook lookupTicketBookService() {
        SharedTicketBook _ticketBook = (SharedTicketBook)
          serviceRegistry.lookup(transitPOSAppURI, "ticketbook");
        String uri = "sio:" + JCSystem.getURI() + "/ticketbookAC";
        AccessChecker accessChecker
          = new AccessChecker(POS_SERVER_ROLE, uri, _ticketBook);
        if (!accessChecker.isGranted()) {
           throw new SecurityException();
       }
    }
}
```

4.2.2.2 Declaring Client Application Roles

To facilitate the security evaluation and the configuration by the application deployer of your application, you, as the application developer, are required to declare all the client application roles your application uses in programmatic security checks.

Roles referenced by client application role-based security checks are declared in the Java Card platform-specific application descriptor.

Caution – If you fail to declare a role referenced by a programmatic security check in your application's code or in any library code that your application may invoke, that role may not be mapped by the application deployer to an actual client and access to the resources it protects will be systematically denied.

The roles referenced by programmatic client role-based security checks of the Transit POS application, namely `TURNSTILE_CLIENT` and `ADMIN_CLIENT` (see EXAMPLE 4-5), are declared at the end of that web application's Java Card platform-specific application descriptor. See EXAMPLE 4-9.

EXAMPLE 4-9 All Client Application Roles (On-card Client Roles) of the Transit POS Web Application (`META-INF/javacard.xml`)

```
<security-role>
    <role-name category=
"ON-CARD-CLIENT">TURNSTILE_CLIENT</role-name>
</security-role>
<security-role>
    <role-name category="ON-CARD-CLIENT">ADMIN_CLIENT</role-name>
</security-role>
```

4.3 Configuring Inter-application Communication Security

When your application is to be deployed, you, as the application deployer, must configure your application as follows:

- you must map the roles declared by the application developer to actual client identities on the targeted card
- you must also configure the peer-to-peer authentication parameters (mainly the session duration) and for those roles that have been mapped to authentication credentials.

4.3.1 Mapping Client Application Roles To Applications

The client application security roles declared in the Java Card platform-specific application descriptor of your application (see Section 4.2.2.2, "Declaring Client Application Roles" on page 4-17) 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 authentication credential aliases, designating the credentials of authorized application clients.

Note – If you, as an application deployer, fail to map a role, that role cannot be granted to any client and, therefore, resources protected by that role will not be accessible. You can use this explicitly to disable certain parts of your application when deployed in a particular operating environment.

For example, the `TURNSTILE_CLIENT` and `ADMIN_CLIENT` roles declared in the Java Card platform-specific application descriptor (see EXAMPLE 3-21) of the Transit POS are mapped in the runtime descriptors to the Transit Turnstile application's authentication credentials and to the Transit Admin application URI, respectively.

EXAMPLE 4-10 User Roles Mapping for the Transit POS Web Application (`META-INF/MANIFEST.MF`)

```
On-Card-Client-Role-List: TURNSTILE_CLIENT, ADMIN_CLIENT
TURNSTILE_CLIENT-Mapped-To-Credential: TURNSTILE-CREDENTIAL
ADMIN_CLIENT-Mapped-To-Client-URI: /transit/admin
```

If the card management policy in place enforces a strict management of the names given to the protection domains to which applications are assigned, all applications from the same application provider (as determined from the application's signer certificate, for example) could be assigned to application protection domains that are given the same name. For example, all Transit applications could be assigned to protection domains named `"Transit"`. In such a case, the `ADMIN_CLIENT` role of the Transit POS could be mapped to the application protection domain name `pd:Transit`. Note though, that this would not only permit the Transit Admin application but also any other application issued by the Transit application provider (and therefore bound to a protection domain named `"Transit"`) to access the administrative functionalities of the POS application. EXAMPLE 4-11 illustrates this fictitious example.

EXAMPLE 4-11 User Role Mapping to Protection Domain Name for the Transit POS Web Application (`META-INF/MANIFEST.MF`)

```
ADMIN_CLIENT-Mapped-To-Client-URI: pd:Transit
```

4.3.2 Configuring Client Application Authentication

For each of the client application roles that you have mapped to authentication credentials you must configure the authentication session duration. The duration of an authentication session determines the validity span of authentication status between your server application and its clients and, therefore, the frequency with which the client identities must be authenticated.

The authentication session duration must be set in the runtime descriptor of the application, either globally, by default, for all its client security roles with the `On-Card-Clients-Credential-Auth-Duration` attribute and/or individually for each of its client security roles with the *<Client-Role>*`-Credential-Auth-Duration` attribute. TABLE 4-2 lists the possible values for these attributes.

TABLE 4-2 Authentication Session Duration Types

Attribute Value	Description
`access`	authentication in a particular role is performed for each invocation of the `JCSystem.isClientInRole` method
`card-session`	authentication in a particular role is performed once for a specific client application and specific server application during the card session
`client-lifetime`	authentication in a particular role is performed once for a specific client application and specific server application during their common lifetime

For example, the Transit POS web application, which exposes the Ticket Book service to the Transit Turnstile applet application, is configured to use a card session-bound authentication session duration. This is a compromise between the finer security of access-bound duration and the reduced performance impact of application lifetime-bound duration. The Transit Turnstile applet application will authenticate only once during a contactless-card session but each time the transit user uses his card to enter or exit the transit system through a turnstile gate. If the response time is still an issue, the authentication session duration could be set to client lifetime to avoid the overhead of on-card client authentication during contactless-card sessions.

EXAMPLE 4-12 Authentication Credential Configuration of the Transit POS Web Application (`META-INF/MANIFEST.MF`)

```
On-Card-Client-Role-List: TURNSTILE_CLIENT, ADMIN_CLIENT
TURNSTILE_CLIENT-Mapped-To-Credential: TURNSTILE-CREDENTIAL
TURNSTILE_CLIENT-Credential-Auth-Type: client-only
TURNSTILE_CLIENT-Credential-Auth-Duration: card-session
ADMIN_CLIENT-Mapped-To-Client-URI: /transit/admin
```

Note – Currently, the Java Card platform only supports application client authentication and the only supported value for the `On-Card-Clients-Credential-Auth-Type` and *<Client-Role>*`-Credential-Auth-Type` attributes is `client-only`.

CHAPTER 5

Securing Connections End-to-end

On the Java Card platform, applications may interact with off-card peers (for example, the mobile phone the card is embedded in or a server located on the Internet) through secure network communications over Secure Sockets Layer (SSL) or Transport Layer Security (TLS). These communications may be established over web container-managed HTTPS server connections between an off-card client and a web application, or over application-managed client or server SSL/TLS or HTTPS connections.

The security of such network communications, also referred to as *end-to-end security*, relies on secure network protocols and cryptography services that ensure the confidentiality and the integrity of the data transmitted, as well as the authentication of the peers.

Application developers may configure the security requirements and characteristics for secure connections with off-card peers either programmatically in the application's code or declaratively in the deployment descriptor of a web application. The security requirements for a secure connection include peer authentication, integrity and confidentiality of the data transmitted.

Additionally, an application may use a *credential manager* object to manage the credentials used to establish secure connections with peers. A credential manager is used both for managing the key material that 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. The actual management of the authentication credentials depends on the *mode of operation* of the application, server and client. An application developer can implement and set his own credential manager, an *application-defined credential manager*, hence implementing an *ad hoc model* of credential management. Or, it can delegate to a credential manager assigned by the card manager application, a *card manager-assigned credential manager,* the management of the keys and trust decisions for the secure connections that it opens or that are opened on its behalf by the web container. Such a *delegated model* of credential management is essential for supporting card management frameworks, such as Global Platform.

A web application developer may declare requirements for content integrity and confidentiality in the deployment descriptor of a web application. The application developer or the application provider can also require that the application be hosted on a dedicated secure port. The web container enforces a web application's requirements for content integrity and confidentiality by accepting requests for protected resources of that application only over HTTPS connections open on that application's dedicated secure port. As for any other secure connection, the security characteristics of connections from web clients on that application-dedicated HTTPS port are negotiated using the credential manager that applies to that application.

Therefore, each web application that requires protection is securely hosted on its own dedicated secure port and uses its own security requirements and credentials to establish secure connections with web clients.

5.1 Designing End-to-end Security

On the Java Card platform, applications may interact with off-card peers through 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.

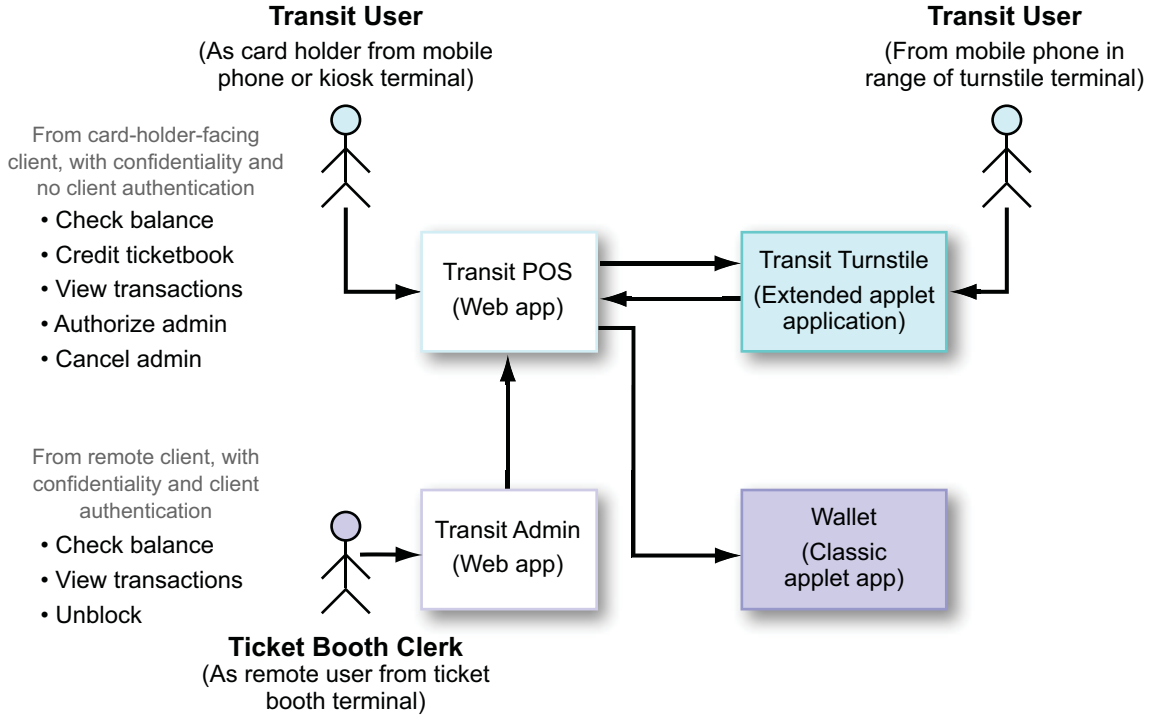
These connections may be set up with distinct security characteristics, including confidentiality, integrity and peer authentication, and may use different cryptographic technologies (symmetric or asymmetric) for peer authentication.

Therefore, you, as an application developer, must first analyze the requirements for each of the secure connections your application must engage in and gather the following information:

- *Security Requirements* - whether the connection peer must be authenticated and whether the integrity and confidentiality of the data transmitted over the connection must be guaranteed. Whether specific cryptography technologies and algorithms must be used.
- *Authentication Credentials and Security Requirements Management Model* - whether the management of the authentication credentials and security requirements must be performed by the card management-assigned credential manager (delegated model) or whether the management of the authentication credentials and security requirements must be performed by the application itself (ad hoc model).

The diagram in FIGURE 5-1 shows the use cases of the Transit application with highlights on security requirements for communication with off-card clients.

FIGURE 5-1 Security Requirements for Communication With Off-card Clients



5.1.1 Determining Security Requirements

The security requirements for a secure connection may consist of the following:

- peer authentication
- integrity of the data transmitted
- confidentiality of the data transmitted
- optionally, specific cipher suites that may be used.

Your choice of a cipher suite not only determines the cryptographic algorithms that will be used for guaranteeing the integrity and confidentiality of the data transmitted but also determines the cryptographic technology used for peer authentication (symmetric cryptography or asymmetric cryptography).

The cryptographic technology used for peer authentication, either symmetric or asymmetric cryptography, determines the type of credentials to be used for authentication. The cryptographic technology impacts the way these credentials will be provisioned and managed on the card. Examples include:

- Pre-shared Keys (PSK), secret keys that must be shared between all the peers beforehand.
- Public Key Certificates whose issuance and validity checking require a Public Key Infrastructure (PKI).

If your application is a web application, then in addition to the security requirements that you must determine for each of the application-managed secure connections your application may open through the GCF, you must further analyze the security requirements, if any, of the connections the web container will manage to host your application.

5.1.1.1 Determining The Security Requirements of a Web Application

If your application is a web application, you must determine whether certain or all resources of your web application have specific transport guarantee requirements such as confidentiality and integrity. The transport guarantee requirements for a specific web resource determine the security requirements of the connection over which that resource will be serviced. Unless there is a requirement for exclusive secure access of all your web application's content, resources that have no transport guarantee requirements may be serviced over plain HTTP connections.

As a web application developer, you must determine the following requirements:

- Whether any resource is to be serviced exclusively over secure connections.
- Whether only certain resources are to be serviced exclusively over secure connections.
- Whether the whole application's contents (all resources) are to be serviced exclusively over secure connections.
- Whether the secure port on which to host the protected content of your web application may be dynamically allocated or statically allocated. You should opt for static port allocation only if you have the guarantee that this will not conflict with other applications, such as in a closed deployment environment. Otherwise you should favor dynamic port allocation.
- Whether the web clients attempting to access your web application's protected content must first authenticate.

You may typically choose to service content that is not sensitive, such as images and other static resources, over plain/non-secure connections for better performance and response time.

Note also that user authentication, hence exchanges of user credentials, should always be performed over secure connections, see Securing User Authentication.

For example, the Transit POS web application and the Transit Admin web application have specific security requirements for the connections over which their web resources are serviced. They both require web clients to authenticate and require integrity and confidentiality of the exchanges with web clients. TABLE 5-1 lists for both of these applications the user security roles and the functions or resources they protect.

TABLE 5-1 Protected Functions or Resources of the Transit POS and Transit Admin Web Applications and their Security Requirements

Application	Protected Resource(s)	Transport Guarantee Requirement	Client Authentication
Transit POS web application	Ticket book balance operation Ticket book history operation Ticket book credit operation Owner authentication for card holder authorization Ticket book unblock operation	`Confidentiality`	`No`
	Index and error pages	`None`	
Transit Admin web application	Ticket book balance operation Ticket book history operation Ticket book unblock operation Authorizing remote access to the application	`Confidentiality`	`Yes`
	Index and error pages	`None`	

The Transit POS web application and the Transit Admin web application additionally rely on public key certificates to authenticate web clients and/or to authenticate to web clients.

5.1.2 Determining The Authentication Credentials and Security Requirements Management Model

The choice of an authentication credential management model is usually dictated by the operating and business environment to which an application will be deployed.

By default, the platform assumes a delegated model, which means the management of the authentication credentials is delegated to a card management-assigned credential manager.

If the management of the authentication credentials must be performed by your application directly (ad hoc model), you must implement an ad hoc credential manager (application-defined credential manager). See Section 5.2.2.1, "Coding Ad Hoc Credential Management" on page 5-11.

5.2 Implementing End-to-end Security

Based on your design decisions, implementing your application's end-to-end security may encompass the following:

- Setting up the security requirements for some or all of the secure connections that your application will open:
  - **Declaratively** - if your application is a web application and you want to discriminatively configure the user data constraints (that state the transport guarantee requirements) for your application's web resources.
  - **Programmatically** - if you opted for an ad hoc configuration of the security requirements.
- Implementing peer-to-peer authentication for the secure connections that your application will open if you opted for an ad hoc management of your application's authentication credentials.

Note – The Java Card platform does not provide built-in support for securing communications over APDU. You, as an applet developer, may rely on third parties frameworks such as Global Platform for managing secure channels with applet applications, or you may choose to implement your own ad hoc solution. The Transit Turnstile applet application provides an example of securing contactless exchanges between the applet and the turnstile terminal.

5.2.1 Setting Up Security Requirements For Secure Connections

You may set up the security requirements for some or all of the secure connections that your application will open as follows:

- **Declaratively** - if your application is a web application and you need to configure distinct security requirements for distinct sets of your application's web resources.

- **Programmatically** - if your application opens secure connections using the GCF or if you want to more precisely configure the security requirements of the connections to your web applications.

5.2.1.1 Defining A Web Application's Declarative User Data Constraints

If your application is a web application, you may define distinct confidentiality and integrity requirements for distinct sets of your application's web resources. These transport guarantee requirements must be defined in the deployment descriptor of your web application using user data constraints (`user-data-constraint` elements). A user data constraint establishes a requirement that requests the constrained resource be received over a protected transport layer connection. The strength of the required protection is defined by the value of the transport guarantee (`user-data-constraint/transport-guarantee` element). TABLE 5-2 lists the possible values of transport guarantee.

TABLE 5-2 Transport Guarantee Values

Element Value	Description
`INTEGRAL`	used to define a requirement for content integrity
`CONFIDENTIAL`	used to define a requirement for confidentiality
`NONE`	used to indicate that the container must accept the constrained requests when received on any connection including an unprotected one

For example, all the dynamic web resources of the Transit Admin web application must be serviced over an HTTPS connection with a `CONFIDENTIAL` transport guarantee, see EXAMPLE 5-1.

EXAMPLE 5-1 User Data Constraint of the Transit Admin Web Application (`WEB-INF/web.xml`)

```
<security-constraint>
  <display-name>RemoteAdminAccess</display-name>
  <web-resource-collection>
    <web-resource-name>UnblockAndBalance</web-resource-name>
    <description>Unblock and Balance URLs</description>
    <url-pattern>/unblock</url-pattern>
    <url-pattern>/balance</url-pattern>
    <url-pattern>/history</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>Remote Admin</description>
    <role-name>ADMIN</role-name>
   </auth-constraint>
   <user-data-constraint>
     <description>Encryption</description>
     <transport-guarantee>CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

5.2.1.2 Coding Ad Hoc Security Requirement Management

If you have opted for the delegated management model, then you rely on the default security requirements set by the card management application that will load your application.

Otherwise, if you have opted for an ad hoc security requirements management model, you must implement a subclass of the `SecurityRequirements` class. An instance of that class must be set at runtime for the mode(s) of communication that corresponds to that of your application.

The methods of the `SecurityRequirements` are applicable to more than one of the modes of operation `MODE_WEB_SERVER`, `MODE_GCF_SERVER`, or `MODE_GCF_CLIENT`. A `SecurityRequirements` instance set for the mode `MODE_DEFAULT` may only be

invoked for a mode if there is no instance set for that specific mode. TABLE 5-3 summarizes for each of the methods of the `SecurityRequirements` class the modes of operation for which they are invoked.

TABLE 5-3 Invocations of the Methods of the `SecurityRequirements` Class

Methods of the SecurityRequirements Class	Modes of Operation
`isClientAuthRequired`	`MODE_WEB_SERVER` `MODE_GCF_SERVER`

`isIntegrityRequired`	`MODE_WEB_SERVER`
	`MODE_GCF_SERVER`
	`MODE_GCF_CLIENT`

`isConfidentialityRequired`	`MODE_WEB_SERVER`
	`MODE_GCF_SERVER`
	`MODE_GCF_CLIENT`

`chooseCipherSuites`	`MODE_WEB_SERVER`
	`MODE_GCF_SERVER`
	`MODE_GCF_CLIENT`

For web container-managed secure connections, implementing a `SecurityRequirements` subclass is optional as the security requirements for these connections can be configured declaratively in the deployment descriptor and runtime descriptor of the application, see Section 5.2.1.1, "Defining A Web Application's Declarative User Data Constraints" on page 5-7. Nevertheless, you may still have to implement a `SecurityRequirements` subclass if you need to restrict the cipher suites used for the connections. Note that the `SecurityRequirements` base class provides a default implementation of the `isClientAuthRequired`, `isIntegrityRequired` and `isConfidentialityRequired` methods that reflects the requirement for client authentication and the overall requirements for content integrity and confidentiality declared in the web application's runtime descriptor and deployment descriptor, respectively. This allows you to create subclasses of the `SecurityRequirements` base class that only overwrite the deployment configuration where needed.

Note – You should favor declarative configuration of the security requirements through a web application's descriptors for the additional deployment flexibility and code reusability it provides.

EXAMPLE 5-2 shows the application-defined `SecurityRequirements` implemented by the Transit Admin web application to filter the cipher suites used for the secure connections with its web clients. An instance of this `SecurityRequirements` subclass is set for the mode `CredentialManager.MODE_WEB_SERVER` when the application is instantiated and before any request is dispatched.

EXAMPLE 5-2 Application-defined Security Requirements Implemented by the Transit Admin Application (`AdminSecurityRequirements.java`)

```
public class AdminSecurityRequirements
 extends CredentialManager.SecurityRequirements {

  public String[] chooseCipherSuites(
   String[] supportedCipherSuites,
   String endpointURI, byte mode) {
    if (mode == CredentialManager.MODE_WEB_SERVER) {
      Vector<String> selectedCipherSuites =
       new Vector<String>(supportedCipherSuites.length);
      for (String cipherSuite : supportedCipherSuites) {
        if (cipherSuite.startsWith("TLS_RSA")) {
          selectedCipherSuites.addElement(cipherSuite);
        }
      }
      String[] cipherSuites
       = new String[selectedCipherSuites.size()];
      selectedCipherSuites.copyInto(cipherSuites);
      return cipherSuites;
    }
    return null;
  }
}
```

EXAMPLE 5-3 Application-defined Security Requirements Implemented by the Transit Admin Application (`AdminServlet.java`)

```
CredentialManager.setSecurityRequirements(
  new AdminSecurityRequirements(),
  CredentialManager.MODE_WEB_SERVER);
```

5.2.2 Implementing Peer-to-peer Authentication

Managing authentication with off-card peers requires a `CredentialManager` instance to be set for each of the modes of communication that may be applicable. Depending on these modes of communication, the `CredentialManager` instance has the following responsibilities:

- **Server mode** - select the authentication credentials that will determine if the authentication credentials presented by the off-card client should be trusted

- **Client mode** - select the authentication credentials to be presented for authentication with the off-card server

If you have opted for the delegated authentication credential management model, then you rely on the default credential manager set by the card management application that will load your application.

If you have opted for an ad hoc authentication credential management model, you must implement an application-defined credential manager. An instance of that application-defined credential manager class must be set at runtime for the mode of communication that corresponds to that of your application .

5.2.2.1 Coding Ad Hoc Credential Management

An application-defined credential manager must be a subclass of the `CredentialManager` class.

Depending on the type of cryptographic material your application must use for peer authentication, you must provide concrete implementations of distinct sets of methods of the `CredentialManager` class, one for public key cryptography (PKI) and one for secret key cryptography (PSK). If your application uses both types of cryptography material, you must implement both sets of methods.

Also, some of these methods are applicable to more than one of the modes of operation `MODE_WEB_SERVER`, `MODE_GCF_SERVER`, or `MODE_GCF_CLIENT`. A `CredentialManager` instance set for the mode `MODE_DEFAULT` may only be invoked for a mode if there is no instance set for that specific mode.

TABLE 5-4 summarizes for each cryptography material type the methods of the `CredentialManager` class and the modes of operation for which they are invoked.

TABLE 5-4 Invocations of the Methods of the `CredentialManager` Class Based on the Type of Cryptography Material Used for Authentication

Cryptography Material Type	Methods of the CredentialManager Class	Modes of Operation

Public key cryptography (PKI) authentication	`checkTrusted`	`MODE_WEB_SERVER`
		`MODE_GCF_SERVER`
		`MODE_GCF_CLIENT`

	`chooseCertificateChain`	`MODE_WEB_SERVER`
		`MODE_GCF_SERVER`
		`MODE_GCF_CLIENT`

	`getAcceptedIssuers`	`MODE_GCF_SERVER` `MODE_WEB_SERVER`

	`getMatchingPrivateKey`	`MODE_WEB_SERVER`
		`MODE_GCF_SERVER`
		`MODE_GCF_CLIENT`

TABLE 5-4 Invocations of the Methods of the `CredentialManager` Class Based on the Type of Cryptography Material Used for Authentication

Cryptography Material Type	Methods of the CredentialManager Class	Modes of Operation

Secret key cryptography (PSK) authentication	`choosePreSharedKey`	`MODE_WEB_SERVER`
		`MODE_GCF_SERVER`
		`MODE_GCF_CLIENT`

	`getPSKIdentityHint`	`MODE_WEB_SERVER` `MODE_GCF_SERVER`
	`getPSKIdentity`	`MODE_GCF_CLIENT`

EXAMPLE 5-4 shows the application-defined credential manager implemented by the Transit POS web application to authenticate with its web clients. The authentication credentials are public key cryptography materials. The Transit POS web application uses a public key certificate and its paired private key to authenticate to its clients. It also uses a public key certificate to authenticate its trusted clients. These public key cryptography materials are retrieved as servlet context attributes in a Java Card RI-dependent way, see Section 5.3.2, "Provisioning Or Generating Authentication Credentials" on page 5-17. An instance of this `CredentialManager` subclass is set for the mode `CredentialManager.MODE_WEB_SERVER` when the application is instantiated and before any request is dispatched, see EXAMPLE 5-5. Note that this is the same `CredentialManager` instance that is used for `MODE_SIO_SERVER`, see also EXAMPLE 4-2.

EXAMPLE 5-4 Application-defined Credential Manager Implemented by the Transit POS Application (`POSCredentialManager.java`)

```
public class POSCredentialManager extends CredentialManager {
  private Certificate[] webServerCertChain;
  private PrivateKey webServerPrivateKey;
  private Certificate[] webClientCertIssuers;

  public POSCredentialManager(Certificate[] webServerCertChain,
   PrivateKey webServerPrivateKey,
   Certificate[] webClientCertIssuers,
   String onCardClientKeyData) {
    this.webServerCertChain = webServerCertChain;
    this.webServerPrivateKey = webServerPrivateKey;
    this.webClientCertIssuers = webClientCertIssuers;
    ...
  }
```

EXAMPLE 5-4 Application-defined Credential Manager Implemented by the Transit POS Application (`POSCredentialManager.java`)

```
  public void checkTrusted(Certificate[] certificateChain,
   String authType, String endpointURI, byte mode)
   throws CertificateException {
    if (mode == MODE_WEB_SERVER) {
      for (Certificate certificate : certificateChain) {
        for (Certificate trustedCertificate :
webClientCertIssuers) {
          if (certificate.equals(trustedCertificate)) {
            return;
          }
        }
      }
      throw new CertificateException(
          certificateChain[certificateChain.length - 1],
          CertificateException.UNRECOGNIZED_ISSUER);
    }
    throw new CertificateException(
        certificateChain[certificateChain.length - 1],
        CertificateException.VERIFICATION_FAILED);
  }

  public Certificate[] getAcceptedCertificateIssuers(
    String endpointURI, byte mode) {
    if (mode == MODE_WEB_SERVER) {
      return webClientCertIssuers;
    }
    return null;
  }

  public Certificate[] chooseCertificateChain(String[] types,
    String[] issuers, String endpointURI, byte mode) {
    if (mode == MODE_WEB_SERVER) {
      return webServerCertChain;
    }
    return null;
  }
```

EXAMPLE 5-4 Application-defined Credential Manager Implemented by the Transit POS Application (`POSCredentialManager.java`)

```
  public PrivateKey getMatchingPrivateKey(Certificate
certificate,
    String endpointURI, byte mode) {
    if (mode == MODE_WEB_SERVER) {
      if (webServerCertChain[0].equals(certificate)) {
        return webServerPrivateKey;
      }
    }
    return null;
  }
}
```

EXAMPLE 5-5 Application-defined Credential Manager Implemented by the Transit POS Application (`ContextListener.java`)

```
static final String WEB_SERVER_CERTS_ATTR
  = "com.sun.javacard.credentials.webServerCerts";
static final String WEB_SERVER_PRIV_KEY_ATTR
  = "com.sun.javacard.credentials.webServerPrivKey";
static final String WEB_CLIENT_CERTS_ATTR
  = "com.sun.javacard.credentials.webClientTrustedCerts";

Certificate[] webServerCerts = (Certificate[]) sce
  .getServletContext().getAttribute(WEB_SERVER_CERTS_ATTR);
PrivateKey webServerPrivateKey = (PrivateKey) sce
  .getServletContext().getAttribute(WEB_SERVER_PRIV_KEY_ATTR);
Certificate[] webClientCerts = (Certificate[])
  sce.getServletContext().getAttribute(WEB_CLIENT_CERT_ATTR);
CredentialManager credentialManager = new POSCredentialManager(
  webServerCerts, webServerPrivateKey, webClientCerts, ...);
if (webServerCerts != null && webServerPrivateKey != null) {
  CredentialManager.setCredentialManager(credentialManager,
  CredentialManager.MODE_WEB_SERVER);
}
CredentialManager.setCredentialManager(
  credentialManager, CredentialManager.MODE_SIO_SERVER);
```

5.3 Configuring End-to-end Security

When your application is to be deployed, you, as the application deployer, may configure your application as follows:

- you may provision, if necessary, the credentials your application will use to authenticate off-card peers and authenticate to off-card peers. These authentication credentials may differ depending on the targeted operating environment.
- if your application is a web application, you may additionally configure some of the secure hosting parameters, such as the secure port your application will respond on, according to the targeted operating environment.

5.3.1 Configuring Secure Hosting Of A Web Application

There are two parameters in the runtime descriptor that you may use to configure the secure hosting of your web application:

- **Secure port number -** The optional `Web-Secure-Port-Number` attribute designates the static port number over which the web container will exclusively service the protected content of your application. If you do not specify this attribute, the web container will dynamically assign a free port number to your web application. The secure content of your web application will then be exclusively serviced over that dynamically allocated secure port. Protected content is in both cases indirectly reachable from the default port thanks to secure port redirection. Secure port redirection applies both to statically and dynamically allocated ports, but especially allows for web clients to be unaware of the specific port that was dynamically allocated by the web container. It therefore facilitates the deployment of your application and improves its usability from generic web clients such as regular web browsers. Unless the targeted deployment environment is a closed environment, the use of a static port requires some allocation policy or standardization to avoid conflicts between deployed applications.

Note – You should favor dynamic port allocation over static port allocation for the improved flexibility it provides unless you want to avoid the additional overhead of port redirection by allowing web clients to directly connect on the static port.

- **Secure access requirement** - The optional `Web-Secure-Access-Only` attribute indicates whether the whole content of your application is to be serviced exclusively over secure connections regardless of the specific user data constraints in the deployment descriptor of your application, or whether only the application's resources that are protected by user data constraints are to be serviced exclusively over secure connections.

TABLE 5-5 summarizes how a web application instance is hosted depending on its overall transport guarantee requirements, its secure port allocation requirement and its secure access requirement.

TABLE 5-5 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 the requested dedicated secure port	
	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 the requested 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

In addition to the secure port and secure access requirements, you may configure the requirements for web client authentication. The optional `Web-Client-Auth-Required` attribute indicates whether authentication of web clients will be required when connecting to the secure port on which the protected content of your web application is hosted. The value of this attribute is returned by the default implementation of the `SecurityRequirements` base class and may, therefore, be programmatically overridden by an application-defined or the card management-assigned `SecurityRequirements` instance.

For example, the Transit Admin web application is configured to use a dedicated static secure port and to allow unprotected content to be directly serviced over a plain HTTP connection.

EXAMPLE 5-6 Authentication Credential Configuration of the Transit POS Web Application (`META-INF/MANIFEST.MF`)

```
Application-Type: web
Web-Context-Path: /transit/admin
Web-Secure-Port-Number: 49201
```

5.3.2 Provisioning Or Generating Authentication Credentials

The Java Card platform does not define a standard means for provisioning authentication credentials[1]. However, the Java Card Reference Implementation supports two mechanisms for provisioning these credentials:

- **Provisioning of authentication credentials through the card manager** - the application's authentication credentials are passed to the card manager as deployment parameters. These credentials include a server public key certificate and its paired private key, and a trusted client or client CA certificate. These credentials are made available through the application's card manager-assigned credential manager, which applies to all communication modes.
- **Provisioning of authentication credentials to a web application** - the web application's authentication credentials are packaged along with the application. These credentials may include a server public key certificate chain and its paired private key, and trusted client or client CA certificates. These credentials are made available to the web application as `ServletContext` attributes. The application may retrieve these credentials and make them available through an application-defined credential manager for the applicable communication modes. The web credentials are provisioned in files under the web module's `WEB-INF/credentials` directory:
  - **`WEB-INF/credentials/server.key`** - server's RSA private key file containing the key's hexadecimal-encoded modulus and exponent, available to the web application as a `javacard.security.RSAPrivateKey` object through the `com.sun.javacard.credentials.webServerPrivKey` context attribute
  - **`WEB-INF/credentials/server.crt`** - the server's X509 public key certificate file, in Base64-encoded format, available to the web application as an array of `java.microedition.pki.Certificate` objects through the `com.sun.javacard.credentials.webServerCerts` context attribute
  - **`WEB-INF/credentials/*.crt`** - the trusted client or CA's X509 public key certificate file, in Base64-encoded format, available to the web application as an array of `java.microedition.pki.Certificate` objects through the `com.sun.javacard.credentials.webClientTrustedCerts` context attribute.

For example, the authentication credentials of the Transit Admin web application are packaged along with the application as per a provisioning scheme that is specific to the Java Card Reference Implementation, see EXAMPLE 5-7. These authentication credentials are stored in files under the `WEB-INF/credentials` directory. These

1. The Java Card Specification does not currently define an API for generating `Certificate` objects from an external representation. This is currently implementation dependent.

credentials are loaded by the card manager when the application is deployed and made available to the web application as `ServletContext` attributes, see Section 5.2.2.1, "Coding Ad Hoc Credential Management" on page 5-11.

EXAMPLE 5-7 Authentication Credentials Provisioned Along With The Transit Admin Web Application (`AdminWeb.war`)

```
AdminWeb.war/
  + META-INF/
  - WEB-INF/
      + classes/
      - credentials/
          server.crt
          server.key
          trustedCA.crt
        AdminHistory-1.html
        AdminHistory-2.html
        AdminHistory-Footer.html
        AdminMain-1.html
        AdminMain-Footer.html
        web.xml
    error.html
    index.html
```

CHAPTER 6

Internationalizing and Localizing Applications

On the Java Card platform, a *locale* designates an application's user's language and country. *Internationalization* of an application consists in designing and implementing the application so that it can be adapted to various languages and countries without requiring code changes and recompilation. The result of internationalization is a locale-independent application. *Localization* consists in the adaptation of such a locale-independent application for a specific language and country by adding locale-specific components: translated text, culturally-acceptable images and symbols, etc. The terms internationalization and localization are often abbreviated to *i18n* and *l10n*, respectively.

On the Java Card platform, internationalization and localization of applications are supported by the following platform features:

- character sets and encodings
- locale representation
- locale-specific resource bundle
- mechanism to retrieve the default platform locale and character encoding
- mechanism to retrieve the preferred locales and character encoding of a web application's client.

Caution – The Java Card platform provides only a limited API for internationalization and localization. It neither supports locale-specific formatting of numbers, dates, times and currency amounts, nor does it support locale-specific comparison (collation) of strings.

Internationalization and localization of applications are key to address international deployment of applications that interact with end-users, that is the applications that have a user interface. This is typically the case of most web applications. Moreover, because a smart card is a nomad device, an application may have to account for running in different regional contexts throughout its lifetime. A monetic application,

for example, may have to deal with different currencies along the international itinerary of its holder. Conversely, transit system passes valid within a very limited geographical area may be issued to foreign tourists and, therefore, may have to support different languages.

6.1 Designing A Locale-Independent Application

To design a locale-independent application, that is, to internationalize your application, you, as the application developer, must proceed as follows:

1. Identify your application's locale-sensitive resources and components, such as resources specific to certain cultures or regions.
2. Determine the way your application will retrieve the end-user's locale or preferred locale.
3. Determine the way your application's locale-specific resources and components will be provisioned and retrieved.

6.1.1 Identifying Locale-sensitive Applications Resources and Components

Your application's locale-sensitive resources and components may include the following:

1. The locale-sensitive data that your application will process, input and output and that can be isolated from your application's code. This data may include messages, labels, images, dates, numbers and currencies among many others.
2. The locale-sensitive behavior that your application will implement. This may include any code intended to process or generate locale-specific data that cannot easily be made locale-independent by isolating the locale-sensitive data it processes.

6.1.2 Determining How To Retrieve The Application's Users' Locales

You may choose among several ways of determining your application's users' locales:

- Card issuance-time configuration of the platform locale - the locale is set when the card is issued and your application relies on that locale. This has the same implications as when the locale is configured at application deployment, see next item.
- Application deployment-time configuration of the locale - the locale is statically set when your application is deployed to the card. Deployment-time configuration of the locale may especially apply to your application under the following conditions:
  - if your application is solely facing the card holder whose country and language can be determined at deployment time (or was determined at card issuance-time)
  - and, if your application is not sensitive to external locale-specific environment changes such as time zones, currencies, etc.
- Run-time determination of the user's preferred locale - the locale is dynamically determined when your application is interacting with its user(s). This may apply to your application in any of the following cases:
  - if your application can interact concurrently with users from different regions, such as the card holder and some administrators.
  - if your application is sensitive to external locale-specific environment changes, such as time zones, currencies, etc.
  - or, if your application's users' locale cannot be specifically determined until the user uses it.

6.1.3 Determining How To Provision and Retrieve The Application's Localized Components and Resources

The way your application's localized components and resources will be provisioned, then retrieved by your application depends on the way your application will determine its users' locales conditions. For example:

- If your application uses deployment-time configuration of the user's locale, the set of localized resources and components for that locale may be provisioned at deployment-time as well. The application may retrieve the set of localized resources and components from a generic, locale-independent location within the

application's storage space (file system, class path, etc.). A concrete example of this is the creation by the application provider of distinct localized application bundles, one for each of the locales targeted for deployment. Each application bundle would include the set of localized resources and components that correspond to the targeted locale. This set would overwrite the default resources and components in a locale-independent location within the application's storage space.

- If your application uses run-time determination of the user's locale, the sets of localized resources and components for the supported locales must be provisioned at deployment-time. The application must retrieve the set of localized resources and components that correspond to the user's locale from a locale-specific location within the application's storage space (file system, class path, etc.). A concrete example of this is the creation by the application provider of a single application bundle aiming several locales. The application bundle would include the sets of localized resources and components for each of the targeted locales. Each set would be stored in a locale-specific location within the application's storage space.

Other strategies may be used. For example, instead of provisioning the localized resources along with the application, a web application may use external URLs to designate localized resources located on an external trusted server or store. A specific example would be a web application that generates locale-independent XML content that is laid out in the user's browser using style sheets, images, etc., downloaded from some secure source (the embedding device or some server) and locally cached.

6.2 Implementing A Locale-Independent Application

6.2.1 Isolating Locale-sensitive Resources and Components In Resource Bundles

To allow for your application to be properly localized, you must isolate all the locale-sensitive resources and components of your application.

- You may isolate in resource bundles the resources and components that are referenced from within your application's code. A resource bundle is a container for locale-specific objects. Your application may load as needed locale-specific resources from the resource bundle that is appropriate for the current user's locale. Resource bundles are encapsulated by the `java.util.ResourceBundle` and

`java.util.ListResourceBundle` classes and subclasses thereof. A resource bundle is not limited to contain text but may contain any objects, including references to resources outside the resource bundle, such as URLs.

- You may isolate the locale-specific static resources of your web application in a dedicated directory under your web application's document root (or `WEB-INF` directory for static resource fragments that are dynamically assembled).

Isolating your application's locale-sensitive resources and components in a resource bundle facilitates the job of translator and localization engineers who do not have to go through your code to identify resources and components to localize.

For example, the Transit POS Web application isolates the locale-specific resources and components that are referenced from within its code in subclasses of the `ListResourceBundle` class. The code that is used to format dates in a locale-specific way has been encapsulated in a dedicated component subclassing the `DateFormatter` class.

EXAMPLE 6-1 Default English-Localized Resource Bundle of the Transit POS Web Application (`Resources.java`)

```
public class Resources extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }

  static final Object[][] contents = {
    // LOCALIZE THIS
      {"AUTHORIZE", "AUTHORIZE"},
      {"CANCEL", "CANCEL"},
      {"Time:", "Time:"},
      {"Debit:", "Debit:"},
      {"Credit:", "Credit:"},
      {"DateFormatter", new DateFormatter_en()}
    // END OF MATERIAL TO LOCALIZE
  };

  private static class DateFormatter_en extends DateFormatter {

    public String format(Date time) {
      Calendar calendar = Calendar.getInstance();
      calendar.setTime(time);
      StringBuffer sb = new StringBuffer();

    // LOCALIZE THIS
      sb.append(calendar.get(Calendar.HOUR)).append(':');
      append(sb, calendar.get(Calendar.MINUTE), 2);
      sb.append(calendar.get(Calendar.AM_PM) == Calendar.AM
                ? "AM"
                : "PM").append(' ');
      sb.append(calendar.get(Calendar.MONTH)).append('/');
      sb.append(calendar.get(Calendar.DAY_OF_MONTH)).append('/');
      sb.append(calendar.get(Calendar.YEAR));
    // END OF MATERIAL TO LOCALIZE

      return sb.toString();
    }
  }
}
```

In the Transit POS web application the default English static resources are stored directly under the document root directory. Localized versions of these static resources may be stored in local-specific subdirectories named after their locales. Similarly the default english static resource fragments are stored under the `WEB-INF` directory and additional localized versions of these fragments may be stored in local-specific subdirectories named after their locales. EXAMPLE 6-2 lists the localized content of the Transit POS Web application.

EXAMPLE 6-2 Default English-Localized Content of The Transit POS Web Application (`POSWeb.war`)

```
POSWeb.war/
  + META-INF/
  - WEB-INF/
      + classes/
      + credentials/
      POSCredit-1.html
      POSCredit-footer.html
      POSHistory-1.html
      POSHistory-2.html
      POSMain-1.html
      POSMain-footer.html
      POSHistory-footer.html
      POSMain-footer-2.html
      web.xml
      + <locale>/
  OwnerAuthorization.html
  OwnerAuthorizationCancelled.html
  error.html
  index.html
  l10-error.html
  + <locale>/
```

6.2.2 Determining The Application's User Locale

Your application's users locales may be determined in one of the following ways:

- **At card issuance-time** - If the locale of your application's users is determined by that of the platform, such as when cards are issued to specific locales, you may determine that locale by calling the `getDefault` method of the `java.util.Locale` class.
- **At application deployment-time** - The locale may be a configuration parameter of your application. For a web application, this may be, for example, a context-wide initialization parameter.
- **At runtime, during an interactive user session** - If your application is a web application, you may determine the preferred locale of the user by calling the `ServletRequest.getLocale` method. A call to the

`ServletRequest.getLocales` method returns an enumeration of the locales acceptable to the user, in decreasing order of preference. These locales are retrieved from the `Accept-Language` HTTP request header. The value of this HTTP request header is set by the web client according to its own configuration. Note that when the web client does not set the `Accept-Language` HTTP request header, the value retrieved is the default platform locale (as determined by a call to `Locale.getDefault`).

EXAMPLE 6-3 shows how the Transit POS web application determines the default locale from context and servlet initialization parameters, and how it retrieves the locales acceptable to the user from the HTTP request.

EXAMPLE 6-3 Retrieving The Transit POS Web User's Accepted Locales From An HTTP Request (`POSServlet.java`)

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {

  ResourceBundle resourceBundle = (ResourceBundle) session
    .getAttribute(RESOURCES_ATTR);
  if (resourceBundle == null) {
    resourceBundle = getResourceBundle(request.getLocales());
    session.setAttribute(RESOURCES_ATTR, resourceBundle);
  }
  ...
}
```

6.2.3 Retrieving The Application's Localized Components and Resources

6.2.3.1 Retrieving Components and Resources From A Resource Bundle

Once it has retrieved the user's locale or acceptable locales, your application must retrieve the resource bundle appropriate for that locale. To do so, you may call the `ResourceBundle.getBundle` method and pass it both the base name for the resource bundle and the locale. If a resource bundle cannot be found for the provided locale or its parent (the one without the country code), the `ResourceBundle.getBundle` method will attempt to return the resource bundle for the platform default locale (or its parent) and will ultimately default to the base resource bundle.

You may want to iterate over the acceptable locales until the `ResourceBundle.getBundle` method finds a resource bundle that is not the one for the platform default locale or the base resource bundle.

If the acceptable locales are `fr_FR` and `pl`, and the default platform locale is `en_US`, then for resource bundle's base name "`com.sun.jcdemo.transit.pos.Resources`", the following resource bundle names will be searched in sequence during this process:

1. `com.sun.jcdemo.transit.pos.Resources_fr_FR`
2. `com.sun.jcdemo.transit.pos.Resources_fr`
3. `com.sun.jcdemo.transit.pos.Resources_pl`
4. `com.sun.jcdemo.transit.pos.Resources_en_US`
5. `com.sun.jcdemo.transit.pos.Resources_en`
6. `com.sun.jcdemo.transit.pos.Resources`

Note that `ResourceBundle.getBundle` does not limit itself to searching for the one resource bundle matching a specified locale, but once that bundle is found it builds a delegation chain from the more specific bundle found to the most generic one for that matched locale. For example, if `getBundle` found the bundle named `com.sun.jcdemo.transit.pos.Resources_fr_FR`, it will attempt to build the following delegation chain: `com.sun.jcdemo.transit.pos.Resources_fr_FR`, `com.sun.jcdemo.transit.pos.Resources_fr`, `com.sun.jcdemo.transit.pos.Resources`.

EXAMPLE 6-3 shows how the Transit POS web application after retrieving the locales acceptable to the user from the request, looks for the resource bundle appropriate for that locale and caches it in the session. Changes of the user's locale during that same session will not be taken into account.

EXAMPLE 6-4 shows how the Transit POS web application uses the user's acceptable locales and starts looking in the application's class path for an appropriate resource bundle, starting from the preferred locale. If no resource bundle can be retrieved for the acceptable locales, the search defaults successively to the default locale configured for the application, then to the platform default locale.

EXAMPLE 6-4 Retrieving The Resource Bundle Appropriate For The Transit POS Web User's Accepted Locales From Its Class Path (`POSServlet.java`)

```
private ResourceBundle getResourceBundle(Enumeration<Locale> locales)
    throws IOException, ServletException {
  ResourceBundle resourceBundle = null;
  while (locales.hasMoreElements()) {
    Locale locale = (Locale)locales.nextElement();
    resourceBundle
      = ResourceBundle.getBundle("com.sun.jcdemo.transit.pos.Resources",
                                 locale);
    // Keep searching unless this is the default locale
    // and it is one of the acceptable locales
    if ((locale.equals(Locale.getDefault())
         && resourceBundle.getLocale().equals(Locale.getDefault()))
        || !resourceBundle.getLocale().toString().equals("")) {
      break;
    }
  }
  return resourceBundle;
}
```

Once you have found a resource bundle appropriate for your application's user's locale, you can retrieve individual resources or components from the bundle using the `getObject`, `getString` and `getStringArray` methods of the `ResourceBundle` class and passing the key identifying the resource or component you are looking for. Individual resources and components are searched along the resource bundle delegation chain from the more specific localized bundle to the more generic one. If a resource cannot be found in one bundle it then delegates to its parent.

EXAMPLE 6-5 shows how the Transit POS web application retrieves from the resource bundle different text labels and a component for formatting dates.

EXAMPLE 6-5 Retrieving Individual Resources and Components From The Resource Bundle Appropriate For The Transit POS Web User's Locale (`POSServlet.java`)

```
DateFormatter dateFormatter = (DateFormatter)
  resourceBundle.getObject("DateFormatter");
String timeLabel = resourceBundle.getString("Time:");
String creditLabel = resourceBundle.getString("Credit:");
String debitLabel = resourceBundle.getString("Debit:");
```

6.2.3.2 Retrieving Localized Web Resources

The Java Card platform does not support any specific means for retrieving localized web resources but you can easily implement such a mechanism using request dispatchers and filters.

For example, the Transit POS web application implements a component (`I18nDispatcher`) that uses the locales acceptable to the web user to search for the appropriate localized version of the requested resource, see EXAMPLE 6-6. It then invokes that resource request dispatcher with a `forward` or `include` method call. The algorithm for searching the appropriate localized resource is similar to that of `ResourceBundle.getBundle`. If the acceptable locales are `fr_FR` and `pl`, and the default platform locale is `en`, then for the requested resource `/OwnerAuthorization.html`, the following resources will be searched in sequence during this process:

1. `/fr_FR_OwnerAuthorization.html`
2. `/pl_OwnerAuthorization.html`
3. `/fr_OwnerAuthorization.html`
4. `/en_OwnerAuthorization.html`
5. `/OwnerAuthorization.html`

It searches first for the exact acceptable locales, then for these locales without their country code, then for the platform default locale, and finally it defaults to the originally requested resource URI. The search stops when one of these localized resources is found. This component can be configured to limit its search to only a list of supported locales. Also, because checking if a web resource actually exists may have an overhead[1] that is not acceptable in some environments, this feature can be switched off. In this case, the delegation chain is not effective and each set of localized resources must, therefore, include all the resources. This, of course, will

1. Unless listing explicitly the localized resources in some data structure other than the file system.

have an impact on footprint as there will be no sharing of resources between children of the same resource bundle; for example, `en_US`, `en_GB` and `en` will not share common resources.

EXAMPLE 6-6 A Component of The Transit POS Web Application That Localizes The Dispatching of Requested URIs (`I18nDispatcher.java`)

```
public class I18nDispatcher {

  public boolean forward(String uri, ServletRequest request,
     ServletResponse response) throws IOException, ServletException {
    uri = (uri != null) ? uri: ((HttpServletRequest) request).getServletPath();
    RequestDispatcher dispatcher=getDispatcher(uri, request);
    if (dispatcher != null) {
      dispatcher.forward(request, response);
      return true;
    }
    return false;
  }

  public boolean include(String uri, ServletRequest request,
     ServletResponse response) throws IOException, ServletException {
    ...
  }

  public RequestDispatcher getDispatcher(String uri, ServletRequest request)
     throws IOException, ServletException {
    boolean[] countryCodes = {true, false};
    for (boolean countryCode : countryCodes) {
      Enumeration<Locale> locales = request.getLocales();
      while (locales.hasMoreElements()) {
        Locale locale = locales.nextElement();
        locale = countryCode ? locale : new Locale(locale.getLanguage());
        RequestDispatcher dispatcher = getLocalizedDispatcher(uri, locale);
        if (dispatcher != null) {
          return dispatcher;
       }
      }
    }

    Locale locale = Locale.getDefault();
    RequestDispatcher dispatcher = getLocalizedDispatcher(uri, locale);
    if (dispatcher != null) {
      return dispatcher;
    }
    return context.getRequestDispatcher(uri);
  }
```

EXAMPLE 6-6 A Component of The Transit POS Web Application That Localizes The Dispatching of Requested URIs (`I18nDispatcher.java`)

```
  private RequestDispatcher getLocalizedDispatcher(String uri, Locale locale) {
    // If the provided locale is one of the supported locales
    // rewrites the provided URI as follows:
    //  /<locale>/<requestedURI> or /WEB-INF/<locale>/<requestedURI>
    // and retrieves the request dispatcher for that localized URI
  }
}
```

The `I18nDispatcher` is invoked by a filter (`I18nFilter`) that is intercepting all the requests for static content, see EXAMPLE 6-7. To apply to static content, this filter has been mapped to the default servlet, see EXAMPLE 6-8. This filter intercepts each request to static resources and invokes the I18nDispatcher to forward the request to the localized version of the requested resource that is appropriate for the locale of the user.

EXAMPLE 6-7 A Filter of The Transit POS Web Application That Forwards Requests to Localized Resources (`I18nFilter.java`)

```
public class I18nFilter implements Filter {
  private I18nDispatcher dispatcher;

  public void init(FilterConfig config) throws ServletException {
    dispatcher = new I18nDispatcher(config.getServletContext());
 }

  public void destroy() {
    dispatcher = null;
  }

  public void doFilter(ServletRequest request, ServletResponse response,
     FilterChain chain) throws IOException, ServletException {
    if (!dispatcher.forward(request, response)) {
      chain.doFilter(request, response);
    }
  }
}
```

EXAMPLE 6-8 Configuration The Transit POS Web Application's Internationalization Filter (`web.xml`)

```
<filter>
  <filter-name>I18n Filter</filter-name>
  <filter-class>com.sun.jcdemo.transit.pos.I18nFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>I18n Filter</filter-name>
  <servlet-name>default</servlet-name>
</filter-mapping>
...
<servlet>
  <description>default servlet</description>
  <servlet-name>default</servlet-name>
  <servlet-class>javacardx.servlet.http.DefaultServlet</servlet-class>
</servlet>
...
<error-page>
  <exception-type>java.util.MissingResourceException</exception-type>
  <location>/l10-error.html</location>
</error-page>
```

Because on the Java Card platform filters are only applied on requests from clients, the `I18nFilter` is not invoked when the `POSServlet` invokes the `forward` and `include` methods on `RequestDispatcher` objects. The `POSServlet` must therefore invoke itself the `forward` and `include` methods on an `I18nDispatcher` object.

See EXAMPLE 6-9 for an example of such an invocation on a static resource.

EXAMPLE 6-9 Invocation of the `I18nDispatcher` on a static resource (`POSServlet.java`)

```
i18nDispatcher.forward("/OwnerAuthorizationCancelled.html", request,
response);
```

See EXAMPLE 6-10 for an example of invocation of the `I18nDispatcher` on a static resource fragment.

EXAMPLE 6-10 Invocation of the `I18nDispatcher` on a static resource fragment (`POSServlet.java`)

```
  i18nDispatcher.include("/WEB-INF/POSHistory-2.html", request, response);
```

6.2.4 Dealing With Character Encodings

The Java Card specification requires character encoding support for UTF-8 and ISO 8859-1 (Latin-1). Other character encodings may be supported on certain platforms.

Character encoding conversion supports converting text between Unicode and other character encodings when reading incoming text from streams or writing outgoing text to streams.

The platform's default character encoding is ISO 8859-1 (Latin-1). If you do not specify a character encoding when reading from a stream using an `InputStreamReader` or writing to a stream using an `OutputStreamWriter`, the platform's default character encoding is used. If you specify a character encoding that is not supported on the platform, you will get an `UnsupportedEncodingException`.

The default character encoding for HTTP requests and responses is ISO 8859-1 (Latin-1). If the character encoding of a request is not specified by the client you can override the default character encoding used by the container by calling the `HttpServletRequest.setCharacterEncoding` method. The character encoding for a response can be set in multiple ways. It can be set by calling `HttpServletResponse.setContentType`, `HttpServletResponse.setLocale` or `HttpServletResponse.setCharacterEncoding`. Calling `setContentType` with the parameter `"text/html"` and calling `setCharacterEncoding` with the parameter `"UTF-8"` is equivalent to calling `setContentType` with the parameter `"text/html; charset=UTF-8"`. When calling `setLocale`, the web container uses the locale-to-character encoding mapping set in the `locale-encoding-mapping-list` element of a web application's deployment descriptor to determine the corresponding character encoding. The character encoding of a request or response must be set prior to reading from the request or writing to the response. If you use a character encoding that is not supported on the platform, you will get an `UnsupportedEncodingException`.

You should programmatically set the locale of the responses based on the locales acceptable to the user using the `HttpServletResponse.setLocale` method. This will allow for configuring the character encoding in the deployment descriptor of your web application at the time of deployment according to the character encodings supported by the targeted platform.

6.3 Deploying A Localized Application

6.3.1 Localizing An Application

To localize an internationalized application, you, as a localization engineer must translate the resources and components that have been identified by the application developer as locale-sensitive and which have been isolated in the default resource bundle and default static resources locations.

For example, to localize the Transit POS web application for the `fr` locale, a new subclass of `ListResourceBundle` was implemented. The resources and components were adapted to the `fr` locale. The class name was suffixed with `_fr` and was placed in the same Java package as that of the default resource bundle. See EXAMPLE 6-11.

EXAMPLE 6-11 French-Localized Resource Bundle of the Transit POS Web Application (`Resources_fr.java`)

```
public class Resources_fr extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }

  static final Object[][] contents = {
    // LOCALIZED
      {"AUTHORIZE", "AUTORISER"},
      {"CANCEL", "ANNULER"},
      {"Time:", "Date/Heure:"},
      {"Debit:", "D&eacute;bit:"},
      {"Credit:", "Cr&eacute;dit:"},
      {"DateFormatter", new DateFormatter_fr()}
    // END OF LOCALIZED MATERIAL
  };

  private static class DateFormatter_fr extends DateFormatter {

    public String format(Date time) {
      Calendar calendar = Calendar.getInstance();
      calendar.setTime(time);
      StringBuffer sb = new StringBuffer();

    // LOCALIZED
      sb.append(calendar.get(Calendar.HOUR_OF_DAY)).append('h');
      append(sb, calendar.get(Calendar.MINUTE), 2).append(' ');
      sb.append(calendar.get(Calendar.DAY_OF_MONTH)).append('/');
      sb.append(calendar.get(Calendar.MONTH)).append('/');
      sb.append(calendar.get(Calendar.YEAR));
    // END OF LOCALIZED MATERIAL

      return sb.toString();
    }
  }
}
```

The static web resources and web resource fragments were also adapted to the `fr` locale and placed in subdirectories named after that locale under the document root directory and the `WEB-INF` directory, respectively. Only those resources that required translation were localized. See EXAMPLE 6-12.

EXAMPLE 6-12 French-Localized Content of The Transit POS Web Application (`POSWeb.war`)

```
POSWeb.war/
  + META-INF/
  - WEB-INF/
      + classes/
      + credentials/
       - fr/
          POSCredit-1.html
          POSCredit-footer.html
          POSHistory-1.html
          POSHistory-2.html
          POSMain-1.html
          POSMain-footer.html
       POSCredit-1.html
       POSCredit-footer.html
       POSHistory-1.html
       POSHistory-2.html
       POSMain-1.html
       POSMain-footer.html
       POSHistory-footer.html
       POSMain-footer-2.html
       web.xml
  OwnerAuthorization.html
  OwnerAuthorizationCancelled.html
  error.html
  index.html
  l10-error.html
  - fr/
     OwnerAuthorization.html
     OwnerAuthorizationCancelled.html
     error.html
     index.html
```

6.3.2 Provisioning Localized Resources and Components

Depending on the locale or locales of the environment you are targeting for deployment, you, as the application deployer, must package along with your application the required localized resources and components:

- If your application is to be deployed for a single locale, you may package your application to only include the localized resources and components for that locale. This is the most effective solution in terms of footprint but may complicate your product management and certification processes if you are targeting many different locales.
- If your application is to be deployed for a set of geographically or culturally related locales, you may package your application to include several sets of localized resources and components. This is less effective in terms of footprint but simplifies your product management and certification processes.
- Other strategies may be envisioned, such as provisioning the localized resources independently of the application itself. However, this requires specific support from the card management or an application framework that the application could use to retrieve its localized resources from a secure source.

If your application is packaged along with one or more resource bundles, you must declare each of the corresponding classes in the Java Card platform-specific application descriptor of your application. For each such class you must insert a dynamically-loaded-classes element in that descriptor. To alleviate the footprint issue inherent to packaging several resource bundles along with your application, you may only declare the resource bundle classes for the locales that you are targeting for deployment. The other undeclared resource bundle classes will be ignored and discarded when your application is loaded, hence saving space.

For example, the default English-localized resource bundle and the French-localized resource bundle classes are declared in the Java Card platform-specific application descriptor of the Transit POS web application, see EXAMPLE 6-13.

EXAMPLE 6-13 Declaration of the Localized Resource Bundles in the Java Card Platform-specific Application Descriptor of the Transit POS Web Application (`javacard.xml`)

```
<dynamically-loaded-classes>
  <display-name>Resources for Default English localization</display-name>
  <class name="com.sun.jcdemo.transit.pos.Resources"/>
</dynamically-loaded-classes>
<dynamically-loaded-classes>
  <display-name>Resources for French localization</display-name>
  <class name="com.sun.jcdemo.transit.pos.Resources_fr"/>
</dynamically-loaded-classes>
```

Unfortunately, this mechanism for eliminating unused code does not apply to static web resources or even class-path resources. You should, therefore, carefully choose the provisioning model you want to use.

6.3.3 Configuring Character Encodings

As described in Section 6.2.4, "Dealing With Character Encodings" on page 6-14, you can configure in your web application's deployment descriptor the character encoding to be used for responses according to the locale of the user.

In general, you should prefer using UTF-8 character encoding for your application's resources for the following reasons. Using UTF-8 in your application's resources (such as class-path resources that you may read or HTML static resources to you may output) allows you to use a single character encoding for all text regardless of the language of the text. Whereas if you choose to use ISO 8859-1 (Latin-1) encoding then all characters outside that encoding need to be specified in the "`\Uxxxx`" notation or if within an HTML page as character entity references (for example "`&#169;`" or "`&copy;`"). Note though that for some languages such as Japanese, the use of UTF-8 may result in larger encoded content as most of the characters will be encoded on several bytes. In such cases and depending on whether or not it is supported on the platform targeted for deployment you may use specific character encoding, such as Shift_JIS for Japanese. This will reduce the footprint of the localized resources stored on card and will reduce the amounts of data transferred to the clients.

For example, the Transit POS web application could declare that the Shift_JIS character encoding must be used when sending content for the `ja` locale, see EXAMPLE 6-11. Note that this character encoding matches the character encoding used for the French-localized HTML pages, see EXAMPLE 6-12.

EXAMPLE 6-14 Locale-To-Encoding Mapping for the Transit POS Web Application (`web.xml`)

```
<locale-encoding-mapping-list>
  <locale-encoding-mapping>
    <locale>ja</locale>
    <encoding>Shift_JIS</encoding>
  </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

EXAMPLE 6-15 A Japanese-Localized HTML Page of the Transit POS Web Application

```
<html>
    <head>
        <meta content="text/html;charset=Shift_JIS" http-equiv="Content-Type">
        <title>...</title>
    </head>
    <body>
    ...
    </body>
</html>
```

CHAPTER 7

Migrating Classic Applet Applications

This chapter describes how to migrate a classic applet application to an extended applet application for use on the Java Card 3 Platform, Connected Edition.

The Java Card 3 Platform, Connected Edition, supports both extended applet applications as well as classic applet applications. There are many benefits to migrating a classic applet application into an extended applet application. These include:

- The extended applet application executes on separate threads for APDU commands received over different I/O interfaces. This allows the extended applet application to process an incoming APDU command received over a contactless interface immediately, even when another APDU command is concurrently being processed on the contacted interface.
- The extended applet application can use the richer programming environment available for connected applications such as the use of Strings and Collection classes.
- The extended applet application can incorporate new functionality, such as initiate communication to an external server, perform nested transactions and perform background tasks.

Note that extended applet applications do not run on the Java Card 3 Platform, Classic Edition. Therefore, once a classic application is refactored into an extended applet application, it can only run on the Java Card 3 Platform, Connected Edition.

7.1 Designing and Refactoring Into An Extended Applet

To refactor a classic applet application into an extended applet, the application programmer must account for the following:

- The classic applet program must be made thread safe. Since the connected environment is multi-threaded, the applet application code may be executed concurrently by more than one thread.
- The classic transaction API invoked by the program needs to be refactored to use the enhanced transaction model instead. The classic transaction API cannot be invoked by an extended applet application.
- If your applet application provides Shareable services to other applet applications via Shareable Interface Objects (SIOs), you may want to upgrade your SIO provider mechanism to use the SIO-based service facility. This allows a more flexible programming pattern.
- If your applet application acts as a client of Shareable services provided by other applet applications, you may want to upgrade your service object client communication code to use the more advanced SIO registry-based lookup mechanism.
- In addition to the above, you may want to use the richer programming environment available on the connected platform for extended applet applications, such as Strings, Collection classes, and automatic garbage collection to simplify and improve your code.

TABLE 7-1 compares the equivalent features of a classic applet application and an extended applet application. Column 1 lists the classic applet application feature, column 2 shows its equivalent as an extended applet application. Column 3 shows

"Yes" if the corresponding classic feature is not supported on an extended applet application and, therefore, requires refactoring the application code that makes use of it.

TABLE 7-1 Equivalent Coding Features - Classic Applet to Extended Applet

Classic Applet Feature	Extended Applet Feature	Refactoring Required
Single Threaded Execution	Code must account for multithreaded environment. Separate APDU dispatcher thread for each I/O interface.	Yes
Single level transaction API: `JCSystem.beginTransaction, JCSystem.commitTransaction ...`	Multi-level transaction subsystem uses `TransactionType` Annotation on methods and classes.	Yes
Inter-application communication SIO client API: `JCSystem.getAppletShareableInterfaceObject`	Inter-application communication SIO client API: `ServiceRegistry.lookup`	No
Inter-application communication SIO server API: `Applet.getShareableInterfaceObject`	Inter-application communication SIO server API: `ServiceRegistry.register, ServiceFactory`	No
No access to connected functionality	Full access to connected functionality.	Yes

7.1.1 Concurrency Considerations

Classic applet applications run in a single-threaded environment both on the classic platform as well as the connected platform.

Extended applet applications run in a multi-threaded environment on the connected platform. To ensure that your classic applet application executes correctly after it has been migrated into an extended applet application, you need to review the application code and operating environment for the following:

- Is your applet application multiselectable? If not, then it cannot be concurrently dispatched on different threads to process an incoming APDU command.
- Does your applet application expose any Shareable Interface Objects (SIOs) to other applications on the card? If so, it could be accessed concurrently from other extended applet applications on the card.

If your applet application is multiselectable and deployed on a card with multiple I/O interfaces, or exposes SIOs to other applications, it must be refactored to ensure that it can be executed concurrently by more than one thread. Some of the techniques used to ensure that your code is thread safe include:

- Ensure that access to class and instance data is synchronized by using `synchronized` methods and `synchronized` blocks to provide a consistent view to each executing thread.
- Use immutable objects when possible.
- Demote instance data into local variables when possible.

7.1.2 Transaction Requirements

The applet application programmer must locate all uses of the classic transaction system and refactor the code to use the advanced transaction facility on the connected platform. The classic transaction system uses these APIs: `beginTransaction()`, `commitTransaction()`, `abortTransaction()`, `getTransactionDepth()`, `getUnusedCommitCapacity()`, and `getMaxCommitCapacity()` methods of the `javacard.framework.JCSystem` class.

The transaction zone demarcated using the classic transaction system between an unconditional call to the `beginTransaction()` API and a subsequent unconditional call to the `commitTransaction()` API must instead be refactored as a method with a `TransactionType(REQUIRES_NEW)` annotation. This will ensure that the code in the new method starts a new transaction upon entry and commits the updates upon normal exit. Any calls to `abortTransaction()` within the transaction zone need to be refactored to throw an exception instead. Uncaught exceptions thrown from within a method annotated with the `TransactionType(REQUIRES_NEW)` clause will cause the transaction to be aborted.

The Transit Turnstile extended applet application performs the functions in the transaction zone within a method annotated as `TransactionType(REQUIRES_NEW)` as shown in EXAMPLE 7-1.

EXAMPLE 7-1 Transactions Using TransactionType(REQUIRES_NEW) Annotated Method

```
/**
* Debits the ticketBook SIO within a transaction
*
* @param transitFare
*       number of ticket points to debit
*/
@TransactionType(REQUIRES_NEW)
 void transactTicketBookDebit(short transitFare) {
    // Debit transit fee
    ticketBook.debit(transitFare);
    if (!isBlocked()) {
        // Reset entry station ID
        entryStationId = -1;
    }
}
```

A transaction zone that conditionally starts a transaction using the classic transaction subsystem by checking if a transaction is in progress (possibly via the `getTransactionDepth()` method) before starting a new transaction via the `beginTransaction()` method, is probably suited to being refactored as a method with the `TransactionType(REQUIRED)` annotation. This annotation instructs the Java Card runtime environment to start a new transaction if one is not already in progress.

7.1.3 Communicating With Other Applications

A classic applet application uses the `getAppletShareableInterfaceObject()` client method of the `javacard.framework.JCSystem` class and the `getShareableInterfaceObject()` server method of the `javacard.framework.Applet` class as the mechanism for establishing communication between the client and server application.

The connected platform offers an advanced client communication mechanism in the SIO facility that allows a server application to register `ServiceFactory` objects with the `ServiceRegistry` instance to process SIO requests from a client application. The client application initiates communication using the `ServiceRegistry.lookup()` method.

The extended applet programmer should refactor the server-client communication code used by the classic applet to instead use the `ServiceRegistry` mechanisms to use the more flexible programming pattern.

7.1.3.1 Server Mode

If your classic applet application implements the `getShareableInterfaceObject()` method in the `javacard.framework.Applet` subclass, the functionality can easily be refactored into the code within the `create()` method of a `ServiceFactory` instance. Of course, the applet instance needs to register this `ServiceFactory` instance with the `ServiceRegistry` using its `register()` method during applet initialization.

Note that if the client application is a web application, the classic API `getShareableInterfaceObject()` API cannot be used to service SIO requests.

7.1.3.2 Client Mode

If your classic applet application uses the `getAppletShareableInterface()` method of the `javacard.framework.JCSystem` class to obtain a server applet application's SIO object, it can easily be refactored to use the `lookup()` method of the `ServiceRegistry` class instead. The APIs are very similar.

Note that the classic `getAppletShareableInterface()` API cannot be used to communicate with a server web application.

The Transit Turnstile extended applet application obtains the SIO object from the POS Web server application as shown in EXAMPLE 7-2.

EXAMPLE 7-2 Turnstile Extended Applet Uses the ServiceRegistry Lookup Mechanism

```
ServiceRegistry serviceRegistry =
        ServiceRegistry.getServiceRegistry();
SharedTicketBook ticketBook = (SharedTicketBook)
        serviceRegistry.lookup(transitPOSAppURI, "ticketbook");
```

7.1.4 Communicating With Classic Applet Applications

If you are programming an extended applet application or a web application that communicates with a classic applet application, you need to understand the use of synchronization proxy classes. In most cases, off-card tools seamlessly process classic applet applications to produce the synchronization proxy classes to account for all runtime requirements. However, if the classic applet application uses the short cut programming pattern described below in "Problem Programming Pattern For

Automatic Proxy Generation" on page 8, you may need to address the proxy limitations as described in "Using New Features Of The Connected Platform" on page 10.

Tip – The converter tool links a classic application's CAP components to another classic application's Shareable Interface class via the other application's export component. Since an extended applet application does not publish an export component, Shareable interface classes visible to a classic application should be declared in the classic application itself or in another classic application or in a classic library. In the `Transit` application suite, the `WalletClassicLib` library declares the `SharedWalletAccess` interface used for communication between the `ClassicWalletApplet` application and the `POSWeb` servlet application.

Understanding Synchronization Proxy Classes

Synchronization proxy classes make sure that when an extended applet communicates with a classic applet via an SIO, the single threaded execution guarantee of the classic applet code is preserved.

Synchronization proxy classes are automatically produced by the tools that process classic applications, namely the Converter and Normalizer. These proxy classes are placed in a subpackage called `proxy` of the classic application's Java package.

The Converter converts the classes of a classic applet application into CAP file components, and packages these with the application classes, descriptors and the proxy classes to produce the classic applet application distribution unit.

Similarly, the Normalizer reverses the conversion by converting CAP file components of a classic applet application into Java class files, and packages these with the CAP components, descriptors and the proxy classes to produce the classic applet application distribution unit.

Every `Shareable` interface class referenced by the classic applet application is implemented by at least one proxy class. A proxy class either implements exactly the same set of Shareable interface classes that are implemented by an SIO class in the package, or as many referenced `Shareable` interface classes as possible. A referenced `Shareable` interface class may belong to the application itself or another application or a *classic library*.

Proxy classes are loaded dynamically as needed, by the Java Card Runtime Environment. They must, therefore, be declared in the `dynamically-loaded-classes` element of the Java Card Platform-specific Application Descriptor. The converter and normalizer tools automatically insert these elements into the descriptor file.

Problem Programming Pattern For Automatic Proxy Generation

Since the Converter tool generates the proxy classes based on visible references to `Shareable` interface classes, it needs visibility into all the interfaces implemented by the SIO objects handled by the application.

In some circumstances, when an extended applet application communicates with a classic applet application, via an intermediary classic applet application that uses some coding short cuts, the proxy classes generated by the Converter may not be working correctly. The specific scenario involved is shown in EXAMPLE 7-3.

EXAMPLE 7-3 Automatic Proxy Generation Failure Case

```
package com.sun.testIntermediaryClassic;
/*shareable interface in intermediary classic server application*/
public interface WalletSI extends Shareable {
  BalanceSI getBalance(Shareable acc);
}

/*shareable interface Object class in intermediary classic server
application*/
public class WalletSIImpl implements WalletSI {
  RealWalletSI realWalletSIO = (RealWalletSIO)
      System.getAppletShareableInterfaceObject
                                    ( TestServerClassicAID, 0);

  BalanceSI getBalance(Shareable acc) {
      return ( realWalletSIO(acc) );
  }
}
package com.sun.testServerClassic;
/*shareable interface in classic server application*/
public interface RealWalletSI extends Shareable {
  BalanceSI getBalance(AccSI acc);
}
package com.sun.testClientExtended;
/* shareable interface in client extended application */

public static final byte WALLET_SI_PARAM = (byte) 1;
public interface AccSI extends Shareable {
  short getAccNum();
  byte getVersion();
}

/* run time exception when testClientExtended application invokes
the testIntermediaryClassic SIO */
...
WalletSI walletSIO =
            lookup( testIntermediaryClassicURL,WALLET_SI_PARRAM);
AccSI myAccount = new AccSIImpl(..);
walletSIO.getBalance( myAccount ); /* causes ClassCastException */
...
```

In the scenario shown in EXAMPLE 7-3, the extended applet client application (`testClientExtended`) calls the classic applet server application (`testServerClassic`) indirectly via the intermediary classic application (`testIntermediaryClassic`). Since the intermediary classic application's shareable interface method `WalletSI.getBalance` simply declares the incoming

parameter as a `Shareable` interface and not the actual `AccSI` interface, the proxy generation tool does not detect any references to the `AccSI` interface and therefore does not generate a synchronization proxy class for the `AccSI` interface. Then, a runtime invocation of the `getBalance` method by the extended applet client application (shown at the bottom of EXAMPLE 7-3) will result in a `ClassCastException` being thrown.

Avoiding The Problem Programming Pattern

The classic applet application programmer can avoid the problem programming pattern described in EXAMPLE 8-7 by simply declaring the actual interfaces in all the SIO method signatures. In the example shown, the signature of the interface method `WalletSI.getBalance` should be made identical to the `RealWalletSI.getBalance` method to avoid the problem.

7.1.5 Using New Features Of The Connected Platform

Your extended applet application can use the richer programming environment and utilities available on the connected platform. For example, text data processing code can use the String class for easier text manipulation. Collection classes may provide better encapsulation of program data, as well as storage and retrieval programming patterns.

Other advanced features of the connected platform, such as background communication sessions with external off-card end-points, are also available to the extended applet programmer. The extended applet application may use the Generic Connection Framework (GCF) classes (`javax.io.microedition`) to establish connections as either a server or a client end-point. Note that the GCF API should not be invoked from the main extended applet thread invoked by the Applet Container to process incoming APDU messages. This ensures that the sequential APDU communication with the CAD terminal is not locked up.

The advanced security features including client role-based security may be used by the extended applet programmer to authenticate users, off-card clients as well as on-card clients. The techniques involved are discussed in Chapter 5 and Chapter 4 respectively.

The extended applet may use the event facility of the connected platform to fire events as well as listen for events from other applications. The Transit Turnstile extended applet application listens to `ticketbook` events fired by POS Web server application as shown in EXAMPLE 7-4.

EXAMPLE 7-4 Turnstile Extended Applet Listens to `ticketbook` Events

```
// TicketBook service event listener registration
BlockedEventListener blockedListener = new
                                    BlockedEventListener(this);
eventRegistry.register(transitPOSAppURI, "ticketbook/overdraft",
                                          blockedListener);
UnblockedEventListener unblockedListener = new
                                  UnblockedEventListener(this);
eventRegistry.register(transitPOSAppURI, "ticketbook/credited",
                                             unblockedListener);
eventRegistry.register(transitPOSAppURI, "ticketbook/unblocked",
                                             unblockedListener);
```

7.2 Configuring The Extended Applet

An extended applet application has three application descriptors, `applet.xml`, `javacard.xml` and the runtime descriptor information in the `MANIFEST.MF` file. These descriptors need to be correctly configured for the deployment environment. The contents of these descriptors are discussed in more detail in Section 8.2.1, "Configuring Application Descriptors For Applet Applications" on page 8-8.

CHAPTER 8

Packaging, Distributing and Deploying Applications

The Java Card application distribution unit is a JAR file. The distribution unit typically comprises the classes, the configuration descriptors and the resources of the application.

Each Java Card application distribution unit, when loaded on a Java Card platform, is accorded code space isolation from other applications on the card and from the system itself. An application communicates with other applications via shareable interface classes or library classes. The unloading of an application distribution unit may be constrained by dependencies from other applications introduced by inter-application communication.

This chapter provides information for the application provider or deployer for assembling the components of the application distribution unit for a web application and an applet application. It also describes how application units and library units become dependent on one another and techniques for minimizing and managing them.

8.1 Packaging A Web Application

A web application is typically composed of:

- **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 for secure communication, user authentication and authorization.

- **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 lifecycles.
- **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.

These components need to be packaged as a *Java Card web application distribution unit* before deployment. The format of the distribution unit is similar to that of the Java EE technology WAR (Web Archive) format. As the Java Card web application distribution format is compatible with the Java EE WAR file format, off-the-shelf tools can be used to process them.

8.1.1 Configuring Application Descriptors For Web Applications

A web application has three application descriptors, `web.xml`, `javacard.xml` and the runtime descriptor information in the `MANIFEST.MF` file. These descriptors need to be correctly configured for the deployment environment.

8.1.1.1 `web.xml`

The web application deployment descriptor (`web.xml`) contains configuration information about the servlets, filters and event listeners and external accessibility characteristics such as URL mapping and security requirements. These elements are configured exactly as in the Java EE technology environment. Some elements, such as the ones related to JSP programming, are not applicable for the Java Card platform. Security constraint elements and filter elements may be configured against servlet URI and static resources with some limitations.

8.1.1.2 `javacard.xml`

The Java Card platform-specific application descriptor (`javacard.xml`) contains Java Card platform-specific elements that are independent of any particular application model (web or applet). It includes elements used for declaring the roles used for programmatic security checks and elements for declaring the service interface classes exposed to other applications.

Declaring Security Roles

The security role related elements have been described in other chapters, see Section 3.2.3.5, "Declaring User Roles" on page 3-35, and Section 4.2.2.2, "Declaring Client Application Roles" on page 4-17.

The client application roles, namely `TURNSTILE_CLIENT` and `ADMIN_CLIENT`, which are used for programmatic client role-based security checks in the Transit POS application are declared at the end of that web application's Java Card platform-specific application descriptor. See EXAMPLE 8-1.

EXAMPLE 8-1 All Client Application Roles (On-card Client Roles) of the Transit POS Web Application (`META-INF/javacard.xml`)

```
<security-role>
    <role-name category=
"ON-CARD-CLIENT">TURNSTILE_CLIENT</role-name>
</security-role>
<security-role>
    <role-name category="ON-CARD-CLIENT">ADMIN_CLIENT</role-name>
</security-role>
```

Declaring Shareable Interfaces

Applications communicate with other applications on the card via `Shareable` interfaces. A `Shareable` interface is an Interface class that directly or indirectly implements the `javacard.framework.Shareable` interface.

If your application acts as a server application and exposes a Shareable interface, it needs to declare these interfaces in the `shareable-interface-classes` element to ensure that these classes are loaded by the `Shareable` interface class loader and can be accessed by other client applications on the card. The Transit POS server application declares the `SharedTicketBook` and `SharedTicketBookControl` shareable interface classes as shown in EXAMPLE 8-2.

You must also identify any `Shareable` interfaces used as parameters or return types in the methods of a `Shareable` interface and list them in the `shareable-interface-classes` element.

EXAMPLE 8-2 Declaring Shareable Interface Classes

```
<shareable-interface-classes>
    <description>Ticketbook service exposes methods to credit,
                 debit and get the balance of the transit user's
                 ticket book
    </description>
    <display-name>Ticket book service</display-name>
    <class name="com.sun.jcdemo.transit.pos.SharedTicketBook"/>
</shareable-interface-classes>

<shareable-interface-classes>
    <description>Shared Ticketbook Control service allows the
                 transit user's ticket book to be
                 unblocked
    </description>
    <display-name>Ticket book service</display-name>
    <class name=
        "com.sun.jcdemo.transit.pos.SharedTicketBookControl"/>
</shareable-interface-classes>
```

Declaring Dynamically Loaded Classes

A dynamically loaded class is one which is dynamically retrieved by your application by calling the `java.lang.Class.forName` method with the name of the class as a `String` parameter. The dynamic class loading feature in the Java programming language allows you, the application programmer, to determine the name of the class for the given functionality at runtime using external information without hard coding the name directly into the program.

If your program dynamically loads classes at runtime on the card, the names of these dynamically loaded classes need to be declared in the `dynamically-loaded-classes` element of the Java Card platform-specific application descriptor. This is required for security purposes to ensure that the dynamically loaded class has been safely programmed for it. Declaring the names of the dynamically loaded classes also allows all the reachable classes to be pre-loaded and verified when the application is loaded on the card.

As an example, the application provider or deployer may insert the class name information as an application property attribute in the application's runtime descriptor prior to deployment when the deployment environment is known. At runtime, your program reads this information using the `JCSystem.getAppProperty` API to obtain the class name information.

Dynamically loaded classes are also used when the name of the class to be loaded is derived from a base class name combined with other runtime information. For example, the `ResourceBundle.getBundle` method dynamically loads an application-defined subclass of the `ResourceBundle` class appropriate for the current user's locale using standardized class naming extensions on behalf of the application. The `ResourceBundle` classes required for the current user's locale must be declared in the `dynamically-loaded-classes` element of the Java Card platform-specific application descriptor. Techniques for internationalizing and localizing your application are described in Chapter 6.

The `TransitExtLib` extension library includes the localized resource bundles classes for the Transit application suite. Extension library distribution units do not need to declare dynamically loaded classes. However, if for example, the Transit POS server application were to include the localized resource bundles classes in the application distribution unit instead, or if the application calls the `ResourceBundle.getResourceBundle` method on classes that are included in an extension library, it would need to declare the English-localized resource bundle class, `Resources`, and the French-localized resource bundle classes chain rooted at `Resources_fr` as shown in EXAMPLE 8-3. Again, the Transit POS server application

EXAMPLE 8-3 Declaring Dynamically Loaded Classes

```
<dynamically-loaded-classes>
  <display-name>Resources for Default English
localization</display-name>
  <class name="com.sun.jcdemo.transit.pos.Resources"/>
</dynamically-loaded-classes>
<dynamically-loaded-classes>
  <display-name>Resources for French localization</display-name>
  <class name="com.sun.jcdemo.transit.pos.Resources_fr"/>
</dynamically-loaded-classes>
```

relies on the `java.util.ResourceBundle.getBundle` method to dynamically load the appropriate `Resources` class. However, the POS server application could instantiate a specific `Resources` class, such as `com.sun.jcdemo.transit.pos.Resources_fr` on its own using code as shown in EXAMPLE 8-4.

EXAMPLE 8-4 Dynamic Loading Using `Class.forName`

```
...
  class<ResourceBundle> myResourceClass = (Class<ResourceBundle>)
      Class.forName ( "com.sun.jcdemo.transit.pos.Resources_fr"
);
  ResourceBundle myResource = myResourceClass.newInstance();
...
```

8.1.1.3 Runtime Descriptor Information in MANIFEST.MF File

The application provider or deployer typically configures the elements in the runtime descriptor based on information about the deployment environment. The deployment environment information includes:

- the user authenticators included on the card
- on-card client application information, such as its URI, protection domain name or credential alias name

The user authenticator information is used by the application deployer for configuring the elements of user authentication in the runtime descriptor as described in Section 3.3, "Configuring An Application's User Security" on page 3-37.

The on-card client information is used by the application deployer for configuring the elements of on-card authentication in the runtime descriptor as described in Section 4.3, "Configuring Inter-application Communication Security" on page 4-18.

The application provider or deployer must also configure the basic identification information, such as application type `web`, and include the web context path.

EXAMPLE 8-5 below shows an edited excerpt from the Runtime Descriptor Information of the Transit POS server web application. It shows the web application type and web context path declarations. The sample also shows how the OWNER-POS user role and the TURNSTILE-CLIENT client role is declared and mapped. In addition, the `WALLET-ASSIST-SIO-URI` user property declaration is shown.

EXAMPLE 8-5 Runtime Descriptor Information

```
Runtime-Descriptor-Version: 3.0
Application-Type: web
Web-Context-Path: /transit/pos
WALLET-ASSIST-SIO-URI: sio://aid/A000000062/03010C0602/Wallet
User-Role-List: OWNER-POS
OWNER-POS-Mapped-To-Auth-URI:
sio:///standard/auth/holder/session/transit/owner-pos/pin
On-Card-Client-Role-List: TURNSTILE_CLIENT
TURNSTILE_CLIENT-Mapped-To-Client-URI:
/transit/turnstile,//aid/a000000062/03010c0d02
```

8.1.2 Configuring Resources

A web application typically includes static resources such as HTML or multi-media content. Some of these resources may be suitable for access from an external client.

In addition, class-path resources such as property files, may be required by your application code for example for purposes of customizing the application to the external off-card environment.

8.1.2.1 Static Resources

Static resources reside in files inside the web application distribution unit.

To make static resource files addressable by an off-card client, you should place them outside the `META-INF` and `WEB-INF` directories. For example, the `index.html` file in the Transit POS application is accessible via the URL `http://<host name>:<default port>/transit/pos/index.html`. These files are also subject to security constraints prescribed in the `web.xml` descriptor.

Static resource files that are used by your application as building blocks for dynamically generated responses are typically placed in the `WEB-INF` directory. This allows your application program to access the static resource information and yet it is not directly accessible to an off-card client. For example, the `WEB-INF/POSCredit-1.html` file is accessed internally by the code in `POSServlet` servlet to compose an HTML response with dynamic content. The code in the Transit POS application uses the `RequestDispatcher.include` API method to directly write the contents of `WEB-INF/POSCredit-1.html` to the HTTP Response object.

Files placed in the `META-INF` directory are also not addressable by an off-card-client. This space is typically used for inserting meta-data files, which are intended for the implementation-specific card manager. For example, certificate data could be placed here to customize the application's credentials. Note that the usage and naming conventions for files in this space are not standardized at this time.

Note – Files placed under the `META-INF` directory are visible neither to the off-client nor to the application code.

8.1.2.2 Class-path Resources

Resources (other than the class files themselves) under the `WEB-INF/classes` directory are also referred to as *class-path resources*. These resources can also be accessed via the `Class.getResourceAsStream` API method. Your program would typically use a class-path resource when it is bound closely to the code rather than to the web application functionality.

8.2 Packaging An Applet Application

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 lifecycles.
- **An applet application deployment descriptor:** The deployment descriptor contains information to uniquely identify each concrete applet class in the application.

These components need to be packaged as a *Java Card applet application distribution unit* before deployment. The structure of the distribution unit is similar to the Java Card Web application distribution unit with applet-specific descriptors and directory names replacing those of the web.

8.2.1 Configuring Application Descriptors For Applet Applications

An applet application has three application descriptors, `applet.xml`, `javacard.xml` and the runtime descriptor information in the `MANIFEST.MF` file. These descriptors need to be correctly configured for the deployment environment.

8.2.1.1 `applet.xml`

The applet application deployment descriptor (`applet.xml`) contains configuration information about the applet classes in the *applet application distribution unit*.

The names of all the applet classes whose instances need to be selectable by off-card clients must be declared in `applet.xml` along with their respective Application Identifiers (AID).

8.2.1.2 `javacard.xml`

The Java Card platform-specific application descriptor (`javacard.xml`) contains Java Card platform-specific elements that are independent of any particular application model (web or applet). It includes elements used for declaring the roles used for programmatic security checks, and elements for declaring the service interface classes exposed to other applications.

The considerations described in Section 8.1.1.2, "javacard.xml" on page 8-2 for web applications are also applicable to applet applications.

Note that only the shareable interface class information elements are applicable to a classic applet application provider.

8.2.1.3 Runtime Descriptor Information in `MANIFEST.MF` File

If the extended applet application uses role-based authentication facilities, the application provider or deployer needs to configure the elements in the runtime descriptor based on information about the deployment environment. The deployment environment information includes:

- the user authenticators included on the card
- on-card client application information such as its URI, protection domain name or credential alias name

The user authenticator information is used by the application deployer for configuring the elements of user authentication in the runtime descriptor as described in Section 3.3, "Configuring An Application's User Security" on page 3-37.

The on-card client information is used by the application deployer for configuring the elements of on-card authentication in the runtime descriptor as described in Section 4.3, "Configuring Inter-application Communication Security" on page 4-18.

The application provider or deployer must also configure the basic identification information such as application type `extended-applet` or `classic-applet`, and include the package AID information for classic applet applications.

Note that only the basic identification information attributes are applicable to the classic applet application provider.

8.2.2 Configuring Resources

An extended applet application may also include static resources for implementation-specific use.

In addition, class-path resources such as property files, may be required by your application code for example for purposes of customizing the application to the external off-card environment.

8.2.2.1 Static Resources

Static resources reside in files inside the extended applet application distribution unit.

Files placed in the `META-INF` directory are typically used for inserting meta-data files, which are intended for the implementation-specific card manager. For example, certificate data could be placed here to customize the application's credentials. Note that the usage and naming conventions for files in this space are not standardized at this time.

8.2.2.2 Class-path Resources

Resources (other than the class files themselves) under the `APPLET-INF/classes` directory are also referred to as *class-path resources*. These resources can also be accessed via the `Class.getResourceAsStream` API method. Your program would typically use a class-path resource when it is closely bound to the code.

8.2.3 Packaging a Classic Applet Application

A classic applet application is packaged as a *classic applet application distribution unit* and has two additional items:

- CAP (Card Applet) components - These files are named with the `.cap` extension and represent the converted representation of the classic application package suitable for loading on the *Java Card 3 Platform, Classic Edition*.
- Synchronization Proxy classes - These proxy classes, which are typically inserted by the Converter and Normalizer tools, intervene to ensure that the classic applet application executes in a single thread even when called from a web application or an extended applet application.

The application provider or deployer does not need to perform any manual steps to include the additional CAP components and synchronization proxy classes. These items are automatically inserted when the classic application package is pre-processed by the Converter tool. Similarly, the Normalizer tool inserts these items into the classic applet application distribution unit when processing a classic applet application binary CAP file.

The only element applicable to a classic applet application's Java Card platform-specific application descriptor (`javacard.xml`) is the declaration of the exposed shareable interface classes.

The only attributes applicable to a classic applet application's runtime descriptor are the application-type (`classic-applet`) and the package AID information.

Note that the Normalizer tool automatically generates all the descriptors required in the classic applet application distribution unit when processing a classic applet application binary CAP file.

8.3 Managing Dependencies Between Applications

Shareable Interface Objects (SIOs) are used for inter-application communication:

- Event objects are SIOs
- Services, even Authentication Services, use SIOs
- A client application may lookup an SIO from a server application. SIO objects can be passed as parameters or return types between applications

Inter-application communication programming patterns can cause code and data dependencies between the client and server applications.

When a client application distribution unit is deployed and linked to the shareable interface class loaded from a server application deployment unit, a code dependency is created.

When a client application references a server application's SIO object, the client application is deemed to have a data dependency on the server application.

8.3.1 Managing Data Dependencies

To minimize data dependencies, the client application code should lookup the service object (Shareable Interface Object) to obtain a temporary reference to the object each time it is required.

If that is not possible, and the client needs to maintain a reference for performance reasons, the client should listen for the `event:///standard/app/deleting` (`Event.EVENT_STANDARD_APP_DELETING_URI`) event to be notified about any attempt to delete the server application. Upon catching the deletion event, the client should `null` out the reference to the service object.

The Transit Admin application obtains a temporary reference to the `SharedTicketBook` SIO object of the Transit POS application each time it is needed within the `doPost` method as shown in EXAMPLE 8-6.

EXAMPLE 8-6 Minimizing SIO Data Dependency Via Temporary References

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
            throws ServletException, IOException {
    String transitPOSAppURI = JCSystem.getAppProperty(
                                            "transit-pos-app");
    SharedTicketBook ticketBook = lookupTicketBookService()
    ...
```

In contrast, EXAMPLE 8-7 shows how the Transit Turnstile application keeps a reference to the `SharedTicketBook` SIO object of the Transit POS application in the `ticketBook` field, but makes it `null` upon being notified of the `Event.EVENT_STANDARD_APP_DELETING_URI` event on the Transit POS application.

EXAMPLE 8-7 Minimizing SIO Data Dependency Using Deletion Events

```
ticketBook = lookupTicketBookService();

EventRegistry eventRegistry = EventRegistry.getEventRegistry();

// Transit POS Application lifecycle event listener registration
EventNotificationListener listener = new
    EventNotificationListener() {
        public void notify(SharedEvent e) {
            if (e.getURI().equals(
                        Event.EVENT_STANDARD_APP_CREATED_URI) {
                ticketBook = lookupTicketBookService();
                setBlocked(true);
            } else if (e.getURI().equals(
                        Event.EVENT_STANDARD_APP_DELETING_URI) ||
                       e.getURI().equals(
                        Event.EVENT_STANDARD_APP_DELETED_URI)) {
                ticketBook = null;
                setBlocked(true);
            }
    }
eventRegistry.register(transitPOSAppURI,
                Event.EVENT_STANDARD_APP_CREATED_URI, listener);
eventRegistry.register(transitPOSAppURI,
                Event.EVENT_STANDARD_APP_DELETING_URI, listener);
eventRegistry.register(transitPOSAppURI,
                Event.EVENT_STANDARD_APP_DELETED_URI, listener);
```

8.3.2 Managing Code Dependencies

Class pre-loading optimization is used by the Java Card platform during the loading of the application distribution units. Classes in other distribution units that the application or library depend on must already be present on the card.

The `Shareable` interface class is loaded by the `Shareable` interface class loader. Both the client application distribution unit, as well as the server application distribution unit, are linked to the `Shareable` interface class. The application distribution unit, either server or client, from which the `Shareable` interface class was loaded is the *owner* of the class. The other application unit is, therefore, dependent on the owner application distribution unit. The owner application distribution unit cannot be unloaded until all the dependent application units on the card have been unloaded.

You can make the loading order of these applications independent by duplicating these `Shareable` Interface classes in both applications or by placing them in a Classic or Extension library.

8.3.2.1 Duplicating `Shareable` Interface Classes

A `Shareable` interface is typically accessed by both the client application and the server application. The server provides the concrete class, which implements the `Shareable` interface to provide the service. Instances of this class are called Shareable Interface Objects (SIO). The client invokes the service via the methods of the `Shareable` Interface. To enable the client application and the server application to be loaded in any order onto the card, you could package the `Shareable` Interface class in both the applications. These interfaces must also be declared in the `shareable-interface-classes` element of the `javacard.xml` files of both applications.

The Transit POS application and the Transit Admin application share the `SharedTicketBook` Shareable Interface declared in the `TransitExtLib` extension library. However, both applications could have included the `SharedTicketBook` Shareable Interface and declared it in their respective `javacard.xml` files as shown in EXAMPLE 8-8.

EXAMPLE 8-8 Declaring Shareable Interface Classes

```
<shareable-interface-classes>
    <description>Ticketbook service exposes methods to credit,
                 debit and get the balance of the transit
                 user's ticket book</description>
    <display-name>Ticket book service</display-name>
    <class name="com.sun.jcdemo.transit.pos.SharedTicketBook"/>
</shareable-interface-classes>
```

Note – If `Shareable` interface classes are duplicated in both the client and server, the applications can be loaded in any order, but the owner application will depend on which application unit is loaded first.

Note – This technique of including the `Shareable` interface class in both the client and the server does not work if one of the applications is a classic application. The classic application package is always sealed and must therefore be loaded first.

8.3.2.2 Placing Shareable Interfaces in Libraries

Placing the `Shareable` interface class within an extension library (or classic library for classic applications) allows the server and client applications that depend on these classes to be loaded in any order. Of course, the extension library would need to be loaded first.

8.4 Working With Extension Libraries

An extension library typically comprises framework and utility classes and resources that extend the core library functionality. Extension libraries are shared by all the web and extended applet applications on the card.

The following sections address some design and packaging issues unique to the extension library distribution unit.

8.4.1 Managing class-path Resources

Your *extension library distribution unit* may include resource files, such as property or configuration files as class-path resources. These resources are accessed via the `Class.getResourceAsStream` method. HTML or GIF files may also be included but are not static web resources and cannot be directly served to off-card clients of a web application.

Resources in an extension library can be accessed from web applications and extended applet applications, as well as extension libraries themselves. When searching for a requested resource in the extension libraries, the extension library

class loader uses an implicit search path based on the order in which the extension libraries were loaded. If the same resource is present in more than one extension library, the first occurrence of the resource is returned.

8.4.2 Declaring `Shareable` Interface Classes

You need not declare `Shareable` Interface classes. All `Shareable` Interface classes in the extension library distribution unit are automatically loaded by the `Shareable` Interface class loader and made accessible to all applications and extension libraries on the card.

8.4.3 Managing Data In Static Fields

Classes in an extension library distribution unit are accessible to all web and extended applet applications on the card, and, therefore, are also data which are stored in static fields of these classes. Since data belonging to one application's group context are not accessible from another application's group context, the only type of data which are inherently suitable for storing in static fields are those that are accessible to all group contexts, namely primitive data, implicitly transferable objects, and JCRE entry point objects.

The extension library programmer should design the code to run in any application's group context. Objects created by the library code will be owned by the executing group context. In particular, the extension library programmer should be aware that the static initializer methods (`<clinit>`) may execute in any application's group context and objects created during that time may belong to that application's group context.

8.4.4 Managing Dependencies Between Extension Libraries

All extension library distribution units are loaded by the extension library class loader and, therefore, share the same namespace. As a result, dependencies may develop among them.

Extension library distribution units may develop dependencies on classes in other extension library distribution units at link time during deployment. This can happen in various ways:

- code in an extension library directly references a class which is present only in another extension library.

- code in an extension library references a class that is present in its own distribution unit, but the same class is also present in an extension library that was deployed earlier. This aspect is a result of the implicit search path during the loading of extension library classes.

In a similar manner, extension library distribution units may develop dependencies on class-path resources in other extension library distribution units during run time. This can happen in various ways:

- code in an extension library loads a class-path resource that is present only in another extension library using the `Class.getResourceAsStream` method.
- code in an extension library loads a class-path resource that is present in its own distribution unit, but the same class-path resource is present in an extension library that was deployed earlier. This aspect is a result of the implicit search path based on the order of loading of extension library classes.

Before an extension library distribution unit is unloaded, all the application distribution units and extension library distribution units that depend on it must be unloaded first.

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) — 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.

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.

Applet instance URIs are constructed from their applet instance AID using the "aid" registry-based namespace authority as follows:

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

where `<RID>` (resource identifier) and `<PIX>` (proprietary identifier extension) are components of the AID.

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 3 Platform. The Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and is backward compatible with it, targeting resource-constrained devices that solely support applet-based applications.

classic library — a Java programming language package that does not contain any non-abstract classes that extend the class `javacard.framework.Applet`. A classic applet application comprises a Java programming language package that contains one or more non-abstract classes that extend the `javacard.framework.Applet` 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 classic SIO synchronization proxy.

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 role-based security.

Connected Edition one of the two editions in the Java Card 3 Platform. The Connected Edition has a significantly enhanced runtime environment and a new virtual machine. It includes new network-oriented features, such as support for web applications, including the Java™ Servlet APIs, and also support for applets with extended and advanced capabilities. An application written for or an implementation of the Connected Edition may use features found in the Classic Edition.

connection endpoint (client, server) see *application-managed connection endpoint*, *container-managed connection endpoint*.

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 `Applet`, `Servlet` and `Filter`.

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 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.

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.

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 event listener.

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 `String` 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 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 *Java Language Specification*, and code isolation restrictions (see the "Code Isolation" section in *Runtime Environment Specification, Java Card Platform, v3.0, Connected Edition*).

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 *Java Language Specification* are listed in the export file.

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 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).

Finalization is not supported by the Java Card virtual machine. The method `finalize()` is not called automatically by the Java Card virtual machine.

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 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.

The Java Card virtual machine's volatile heap is typically garbage collected on demand and on card tear.

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.

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 *externally visible*.

inter-application communication facility see service facility, event notification facility.

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 — 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:

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.

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.

JDK™ software — an acronym for Java Development Kit. The JDK software is a 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 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 *ISO/IEC 7816 Specification, Part 4*. (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 `javacard.framework.MultiSelectable` 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.

multiselected 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 3 Platform, Connected Edition and the Java Card 3 Platform, Classic Edition.

normalization (URI) the process of removing unnecessary "." and ".." segments from the path component of a hierarchical URI.

object-oriented a programming methodology based on the concept of an *object*, which is a data structure encapsulated with a set of routines, called *methods*, 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 `Permission` 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 *principal name* and authenticated by using *authentication data*. 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 `TransientReference` 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 an interface of an applet application, which extends, directly or indirectly, the interface `java.rmi.Remote`.

Each method declaration in the remote interface or its super-interfaces includes the exception `java.rmi.RemoteException` (or one of its superclasses) in its `throws` clause.

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.

In addition, Java Card RMI imposes additional constraints on the definition of remote methods of an applet application. See *Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition.*

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 2396, 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 *realm*.

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 `servlet` interface. A set of initialization parameters can be associated with a servlet definition. See *Java Servlet Specification, Connected Edition*.

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 *Java Servlet Specification, Connected Edition*.

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 service.

smartcard a card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Unlike magnetic stripe cards, smartcards 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 lifecyle events such as application creation and deletion events, and standard resource lifecyle 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 `Runnable` 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.

tear when a card holder user moves his card out of a reader's range.

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 *currently active context*.

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 classes whose instances can have their ownership transferred to a context different from their currently owning context. Transferable classes are of two types:

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.

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.

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 2396 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 role-based security.

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 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.

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 *Java Servlet Specification, Connected Edition*.

web application archive 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.

A web application archive file is identified by the `.war` extension and is often referred to as a WAR file. A new extension is used instead of `.jar` 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 *Java Servlet Specification, Connected Edition*.

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

Symbols

A

B

C

D

E

N

O

P

R

S

T

U

V

W

X