
Applet Firewall and Object Sharing

The Java Card platform is a multiapplication environment. Multiple applets from different vendors can coexist in a single card, and additional applets can be downloaded after card manufacture. An applet often stores highly sensitive information, such as electronic money, fingerprints, private cryptographic keys, and so on. Sharing such sensitive data among applets must be carefully limited.

In the Java Card platform, applet isolation is achieved through the *applet firewall* mechanism. The applet firewall confines an applet to its own designated area. An applet is prevented from accessing the contents or behaviors of objects owned by other applets.

To support cooperative applications on a single card—for instance, providing wallet, authentication, loyalty, and phone card functions—Java Card technology provides well-defined and secure object sharing mechanisms.

The applet firewall and the sharing mechanisms affect the way you write applets. This chapter explains the behavior of objects, exceptions, and applets in the presence of the firewall and discusses how applets can safely share data by using the Java Card APIs. JCRE implementation details that are not exposed to applets are intentionally ignored.

9.1 Applet Firewall

With applet isolation, the applet firewall provides protection against the most frequently anticipated security concern: developer mistakes and design oversights that might allow sensitive data to be leaked to another applet. It also provides protection against hacking. An applet might be able to obtain an object reference from a publicly accessible location, but if the object is owned by another applet in a different

package, the firewall prevents access to the object. Thus, a malfunctioning applet, or even a “hostile” applet, cannot affect the operations of other applets or the JCRE.

9.1.1 Contexts

The applet firewall partitions the Java Card object system into separate protected object spaces called *contexts*. The firewall is the boundary between one context and another. When an applet instance is created, the JCRE assigns it a context. This context is essentially a *group context*. All applet instances of a single Java package share the same group context. There is no firewall between two applet instances in a group context. Object access between applets in the same group context is allowed. However, accessing an object in a different group context is denied by the firewall.

In addition, the JCRE maintains its own *JCRE context*. The JCRE context is a dedicated system context that has special privileges: access from the JCRE context to any applet’s context is allowed, but the converse, access from an applet’s context to the JCRE context, is prohibited by the firewall. The Java Card object system partitions are illustrated in Figure 9.1.

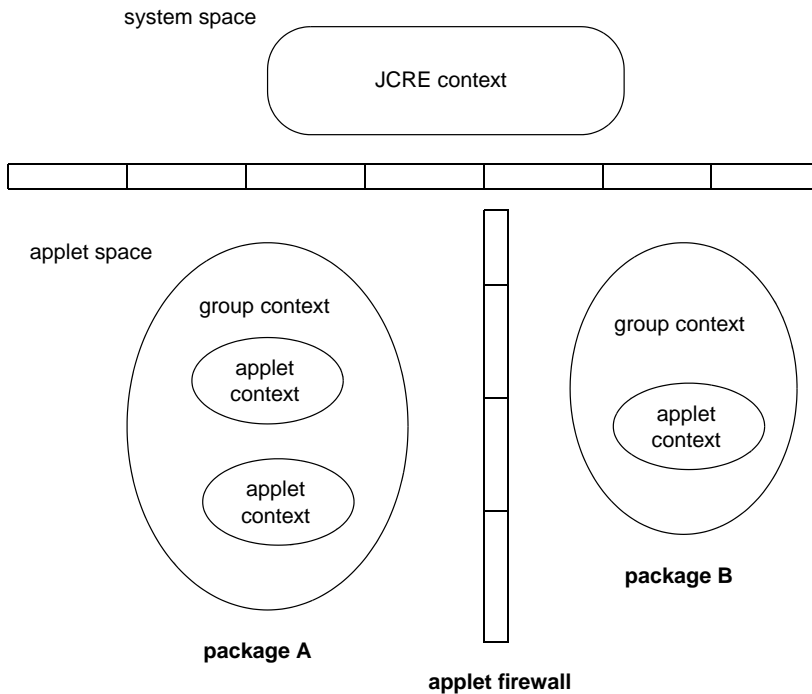


Figure 9.1 The object system partitions on the Java Card platform

9.1.2 Object Ownership

At any time, there is only one *active context* within the virtual machine: either the JCRE context or an applet's group context. When a new object is created, it is assigned an owning context—the currently active context. The object can be accessed from within that context, that is, by all applet instances in its owning context. Also, we say that the object is owned by the active applet in the current context when the object is instantiated. If the JCRE context is the currently active context, the object is owned by the JCRE.

Primitive type static arrays in applets can be initialized when they are declared. Such static arrays are created and initialized by the converter. Because they are statically created before any applet instance is instantiated on the card, the ownership of these arrays can be assigned to any applet instance in their defining package. Any applet within this package can access these arrays. In other words, the owning context of these arrays is the group context of the package.

9.1.3 Object Access

When an object is accessed, the Java language access controls are enforced. For example, a private instance method cannot be invoked from outside the object. In addition, the object's owning context is compared to the currently active context. If the contexts do not match, the access is denied, and the comparison results in a `SecurityException`. For instance, in the examples that follow, assume that object `b` owned is by context `A`. The following operations accessing object `b` from context `A'` cause the Java Card virtual machine to throw a `SecurityException`:

- Get and set fields:

```
short a = b.field_a; // get object b's field
b.field_a = 5; // set object b's field
```

- Invoke a public instance method:

```
b.virtual_method_a();
```

- Invoke an interface method:

```
b.interface_method_a(); // if b is an interface type
```

- Throw it as an exception:

```
throw b;
```

- Cast it to a given type:

```
(givenType)b;
```

- Determine whether it is of a given type:

```
if (b instanceof givenType)
```

- Access an array element:

```
short a = b[0]; // if object b is an array type
b[0] = 5;
```

- Obtain the array length

```
int a = b.length; // if b is an array type
```

9.1.4 Transient Array and Context

Like a persistent object, a transient array of `CLEAR_ON_RESET` type can be accessed only when the currently active context is the array's owning context.

Transient arrays of `CLEAR_ON_DESELECT` type are applet-specific resources. They can be created only when the currently active context is the context of the currently selected applet. Because applets from the same package share a group context, a transient array of `CLEAR_ON_DESELECT` type can also be accessed by all applets in its owning context. However, such access is granted only if one of these applets is the currently selected applet.

9.1.5 Static Fields and Methods

Only instances of classes—objects—are owned by contexts; classes themselves are not. No runtime context check is performed when a static field is accessed or when a static method is invoked. In other words, static fields and methods are accessible from any context. For example, any applet can invoke the static method `throwIt` in the class `ISOException`:

```
If (apdu_buffer[ISO7816.OFFSET_CLA] != EXPECTED_VALUE)
    ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
```

Of course, the Java access rules still apply to static fields and methods. For example, static fields and methods with the *private* modifier are visible only to their defining classes.

When a static method is invoked, it executes in the caller's context. This suggests that objects created inside a static method are assigned with the caller's context (the currently active context).

Static fields are accessible from any context. However, objects (including arrays) referenced in static fields are like regular objects. Such objects are owned by the applet (or the JCRE) that created them, and standard firewall access rules apply.

9.2 Object Sharing across Contexts

The applet firewall confines an applet's actions to its designated context. The applet cannot reach beyond its context to access objects owned by the JCRE or by another applet in a different context. But in situations where applets need to execute cooperatively, Java Card technology provides well-defined and secure sharing mechanisms that are accomplished by the following means:

- JCRE privileges
- JCRE entry point objects
- Global arrays
- Shareable interfaces

The sharing mechanisms essentially enable one context to access objects belonging to another context under specific conditions.

9.2.1 Context Switch

Recall that there is only one active context at any time within the execution of the Java Card virtual machine. All object accesses are checked by the virtual machine to determine whether the access is allowed. Normally, such access is denied if the owning context of the object being accessed is not the same as the currently active context. When a sharing mechanism is applied, the Java Card virtual machine enables access by performing a *context switch*.

Members in an object consist of instance methods and fields. Accessing instance fields of an object in a different context does not cause a context switch. Only the JCRE can access instance fields of an object in a different context. Context switches occur only during invocation of and return from instance methods of an object owned by a different context, as well as during exception exits from those methods.

During a context-switching method invocation, the current context is saved, and the new context becomes the currently active context. The invoked method is now executing in the new context and has the access rights of the current context. When the method exits from a normal return or an exception, the original context (the caller's context) is restored as the currently active context. For example, if an applet invokes a method of a JCRE entry point object, a context switch occurs from the applet's context to the JCRE context. Any objects created by the invoked method are associated with the JCRE context.

Because method invocations can be nested, context switches can also be nested in the Java Card virtual machine. When the virtual machine begins running after a card reset, the JCRE context is always the currently active context.

The following sections explore each sharing mechanism and discuss when and how context switch occurs in each access scenario.

9.2.2 JCRE Privileges

In the Java Card platform, the JCRE acts as the card executive. Because it is the “system” context, the JCRE context has special privileges. It can invoke a method on any object or access an instance field of any object on the card. Such system privileges enable the JCRE to control system resources and manage applets. For example, when the JCRE receives an APDU command, it invokes the currently selected applet's `select`, `deselect`, or `process` method.

When the JCRE invokes an applet's method, the JCRE context is switched to the applet's context. The applet now takes control and loses the JCRE privileges. Any objects created after the context switch are owned by the applet and associated with the current applet's context. On return from the applet's method, the JCRE context is restored.

9.2.3 JCRE Entry Point Objects

The JCRE can access any applet contexts, but applets are not allowed to access the JCRE context. A secure computer system must have a way for nonprivileged users (those restricted to a subset of resources) to request system services that are performed by privileged system routines. In the Java Card platform, this requirement is accomplished by using *JCRE entry point objects*.

JCRE entry point objects are normal objects owned by the JCRE context, but they have been flagged as containing entry point methods. Normally, the firewall would completely protect such objects from access by any applets. The entry point designation allows the public methods of such objects to be invoked from any con-

text. When that occurs, a context switch to the JCRE context is performed. Thus, these methods are the gateways through which applets request privileged JCRE services. Notice that only the public methods of JCRE entry point objects are accessible through the firewall. The fields of these objects are still protected by the firewall.

The APDU object is perhaps the most frequently used JCRE entry point object. Chapter 8 explains how applets instruct the JCRE to read or send APDU data by invoking methods on the APDU object.

There are two categories of JCRE entry point objects:

- *Temporary JCRE entry point objects*—Like all JCRE entry point objects, methods of temporary JCRE entry point objects can be invoked from any context. However, references to these objects cannot be stored in class variables, instance variables, or array fields (including transient arrays) of an applet. The JCRE detects and restricts attempts to store references to these objects as part of the firewall functions of preventing unauthorized reuse. The APDU object and all JCRE-owned exception objects are examples of temporary JCRE entry point objects.
- *Permanent JCRE entry point objects*—Like all JCRE entry point objects, methods of permanent JCRE entry point objects can be invoked from any context. Additionally, references to these objects can be stored and freely reused. The JCRE-owned AID instances are examples of permanent JCRE entry point objects. The JCRE creates an AID instance to encapsulate an applet's AID when the applet instance is created.

Only the JCRE itself can designate entry point objects and whether they are temporary or permanent. JCRE implementors are responsible for implementing the JCRE, including the mechanism by which JCRE entry point objects are designated and how they become temporary or permanent.

9.2.4 Global Arrays

JCRE entry point objects allow applets to access particular JCRE services by invoking their respective entry point methods. The data encapsulated in the JCRE entry point objects are not directly accessible by applets. But in the Java Card platform, the global nature of some data requires that they be accessible from any applet and the JCRE context.

To access global data in a flexible way, the firewall allows the JCRE to designate primitive arrays as *global*. Global arrays essentially provide a shared memory

buffer whose data can be accessed by any applets and by the JCRE. Because there is only one context executing at any time, access synchronization is not an issue.

Global arrays are a special type of JCRE entry point object. The applet firewall enables public fields (array components and array length) of such arrays to be accessed from any context. Public methods of global arrays are treated the same way as methods of other JCRE entry point objects. The only method in the array class is the `equals` method, which is inherited from the root class `Object`. As does invoking any method of a JCRE entry point object, invoking the `equals` method of a global array causes the current context to be switched to the JCRE context.

Only primitive arrays can be designated as global, and only the JCRE itself can designate global arrays. All global arrays must be temporary JCRE entry point objects. Therefore, references to these arrays cannot be stored in class variables, instance variables, or array components (including transient arrays).

The only global arrays required in the Java Card APIs are the APDU buffer and the byte array parameter in an applet's `install` method. Typically, a JCRE implementation passes the APDU buffer as the byte array parameter to the `install` method. Because global arrays can be viewed and accessed by anyone, the JCRE clears the APDU buffer whenever an applet is selected or before the JCRE accepts a new APDU command, to prevent an applet's sensitive data from being potentially "leaked" to another applet via the global APDU buffer.

9.2.5 Object Shareable Interface Mechanism

To reiterate the sharing mechanisms between the JCRE and applets:

- The JCRE can access any object due to its privileged nature.
- An applet gains access to system services via JCRE entry point objects.
- The JCRE and applets share primitive data by using designated global arrays.

Java Card technology also enables object sharing between applets through the *shareable interface* mechanism.

9.2.5.1 Shareable Interface

A shareable interface is simply an interface that extends, either directly or indirectly, the tagging interface `javacard.framework.Shareable`.

```
public interface Shareable {}
```


This interface is similar in concept to the `Remote` interface used by the RMI facility. Neither interface defines any methods or fields. Their sole purpose is to be extended by other interfaces and to tag those interfaces as having special properties.

A shareable interface defines a set of methods that are available to other applets. A class can implement any number of shareable interfaces and can extend other classes that implement shareable interfaces.

9.2.5.2 *Shareable Interface Object*

An object of a class that implements a shareable interface is called a *shareable interface object* (SIO). To the owning context, an SIO is a normal object whose fields and methods can be accessed. To any other context, the SIO is an instance of the shareable interface type, and only the methods defined in the shareable interface are accessible. All fields and other methods of the SIO are protected by the firewall.

9.2.5.3 *Thoughts behind the Shareable Interface Mechanism*

Applets store data in objects. Data sharing between applets means that an applet makes an object it owns available to other applets, thus sharing the data encapsulated in the object.

In the object-oriented world, an object's behavior (aside from direct variable access) is expressed through its methods. Message passing, or method invocation, supports interactions and communications between objects. The shareable interface mechanism enables applets sending messages to bypass the surveillance of the firewall. An owning applet creates a shareable interface object and implements methods defined in the shareable interface. These methods represent the public interface of the owning applet, through which another applet can send messages and consequently access services provided by this applet.

The sharing scenario illustrated in Figure 9.2 can be described as a client/server relationship. Applet A (providing SIOs) is a server, and applets B and C (using the SIOs of applet A) are clients. An applet may be a server to some applets and yet a client of other applets.

In the Java programming language, an interface defines a reference type that contains a collection of method signatures and constants. A client applet views an SIO as having a shareable interface type. The class type of the SIO that implements the shareable interface is not exposed. In other words, only methods defined in the shareable interface are presented to the client applet; instance fields and other methods are not disclosed. In this way, a server applet can provide controlled access to data it wants to share.

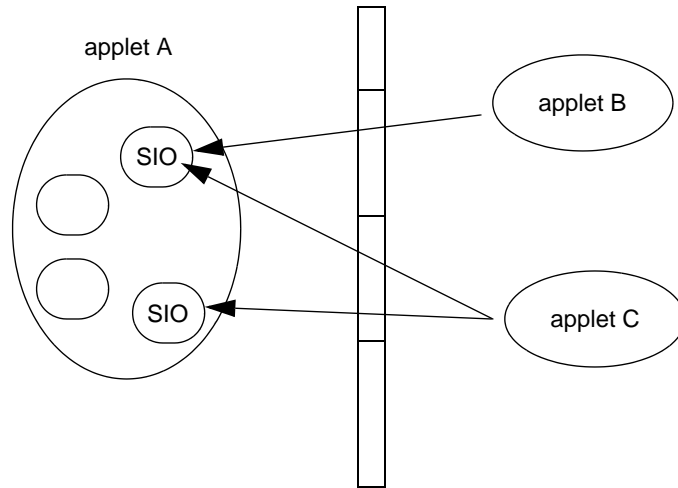


Figure 9.2 Shareable interface object mechanism

When interacting with a different client applet, a server applet could wear a different hat. This would require the server applet to customize its services with respect to the client applet without having the door wide open. A server applet can do so by defining multiple interfaces, each interface declaring methods that are suitable for a group of client applets. If methods in interfaces are distinct, a server applet may choose to create classes, each of which implements an interface. But often services overlap; a server applet can define a class that implements multiple interfaces. Therefore, an SIO of that class can play multiple roles.

9.2.5.4 An Example of Object Sharing between Applets

This section uses a wallet applet and an air-miles applet to provide an example of object sharing between applets. The wallet applet stores electronic cash. The money can be spent to purchase goods.

The air-miles applet provides travel incentives. Similar to the wallet applet, the air-miles applet also stores values—the miles the card holder has traveled. Under a comarketing deal, for every dollar spent using the wallet applet, one air mile is credited to the air-miles applet.

Suppose that the wallet applet and the air-miles applet are in different contexts (they are defined in separate packages). Following are the steps of how they interact in the presence of the firewall.

1. The air-miles applet creates a shareable interface object (SIO).
2. The wallet applet requests the SIO from the air-miles applet.
3. The wallet applet requests miles to be credited by invoking a service method of the SIO.

In this case, the air-miles applet is a server that grants miles on request from the wallet applet (a client), as shown in Figure 9.3.

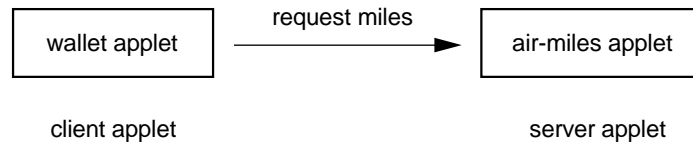


Figure 9.3 Object sharing between the wallet applet and the air-miles applet

Next, how to implement the wallet applet and the air-miles applet is discussed. The code examples are edited for brevity—methods and fields not pertinent to the discussion are omitted, and condition and boundary checks during a transaction are also ignored.

9.2.5.5 Create a Shareable Interface Object

To create an SIO, the server applet (the air-miles applet) must first define a shareable interface that extends `javacard.framework.Shareable`.

```
package com.fasttravel.airmiles;
import javacard.framework.Shareable;

public interface AirMilesInterface extends Shareable {

    public void grantMiles (short amount);
}
```

Next, the server applet creates a service provider class (a service provider class can be the applet class itself) that implements the shareable interface. The server applet can then create one or more objects of the service provider class and can share such objects (SIOs) with other applets in a different context.

```

package com.fasttravel.airmiles;
import javacard.framework.*;

public class AirMilesApp extends Applet
    implements AirMilesInterface {

    private short miles;

    public void grantMiles(short amount){

        miles = (short)(miles + amount);
    }
}

```

Before a client can request an SIO, it must find a way to identify the server. In the Java Card platform, each applet instance is uniquely identified by an AID.

Recall from Chapter 7 that when an applet instance is created, it is registered with the JCRE using one of the two register methods. The method with no parameter registers the applet with the JCRE using the default AID defined in the CAP file. The other register method allows the applet to specify an AID other than the default one. The JCRE encapsulates the AID bytes in an AID object (owned by the JCRE) and associates this AID object with the applet. During object sharing, this AID object is used by a client applet to specify the server.

9.2.5.6 *Request a Shareable Interface Object*

Before requesting an SIO from a server applet, a client applet must first obtain the AID object associated with the server applet. To do that, the client applet calls the lookupAID method in the class JCSysm:

```

public static AID lookupAID
    (byte[] buffer, short offset, byte length)

```

The client applet must know ahead of time the server applet's AID bytes, and it supplies the AID bytes in the parameter buffer. The lookupAID method returns the JCRE-owned AID object of the server applet or returns null if the server applet is not installed on the card. Because the AID object is a permanent JCRE entry point object, the client applet can request it once and cache it in a permanent location for later use.

Next, the client applet calls the method `JCSystem.getAppletShareableInterfaceObject`, using the `AID` object to identify the server:

```
public static Shareable  
getAppletShareableInterfaceObject(AID server_aid, byte parameter)
```

The second parameter in the method `getAppletShareableInterfaceObject` is interpreted by the server applet. It can be used to select an SIO if the server has more than one available. Alternatively, the parameter can be used as a security token, which carries a secret shared by the server and the client.

In the `getAppletShareableInterfaceObject` method, the JCRE looks up the server applet by comparing the `server_aid` with the AIDs of applets that are registered with the JCRE. If the server applet is not found, the JCRE returns null. Otherwise, the JCRE invokes the server applet's `getShareableInterfaceObject` method.

```
public Shareable  
getShareableInterfaceObject(AID client_aid, byte parameter)
```

Notice that, in the `getShareableInterfaceObject` method, the JCRE replaces the first argument with the `client_aid` object and passes along the same parameter byte supplied by the client applet. The server applet uses both parameters to determine whether to provide services to the requesting applet and if so, which SIO to export.

The method `getShareableInterfaceObject` is defined in the base applet class `javacard.framework.Applet`. The default implementation returns null. An applet's class must override this method if it intends to share any SIOs. Here is how the air-miles applet implements the `getShareableInterfaceObject` method. (The process to authenticate the client is described in 9.2.5.10.)

```
public class AirMilesApp extends Applet  
    implements AirMilesInterface {  
  
    short miles;  
    public Shareable getShareableInterfaceObject(AID client_aid,  
                                                byte parameter) {  
  
        // authenticate the client -- explained later  
        // ...  
  
        // return the shareable interface object
```

```

        return this;
    }

    public void grantMiles(short amount){

        miles = (short)(miles + amount);
    }
}

```

When the server applet returns the SIO, the JCRE forwards it to the requester—the client applet. The process of requesting a shareable interface object is summarized in Figure 9.4.

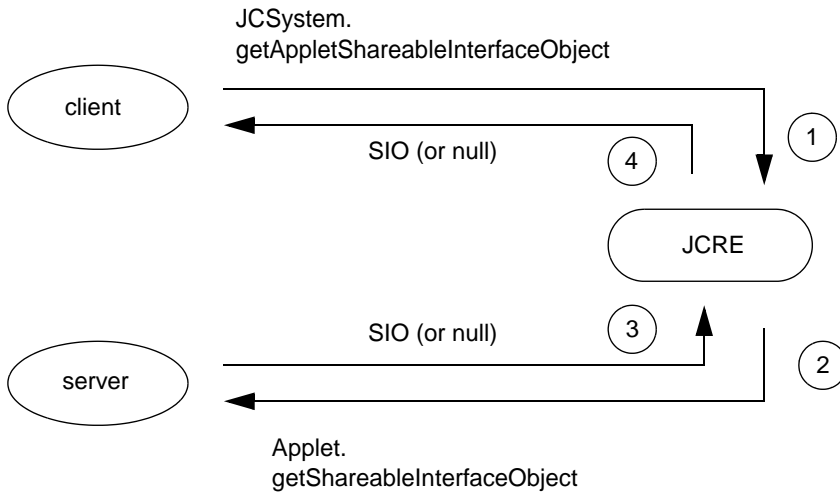


Figure 9.4 Requesting an SIO

9.2.5.7 Use a Shareable Interface Object

To enable a server to return any shareable interface type using a single interface, both methods `JCSys.getAppletShareableInterfaceObject` and `Applet.getShareableInterfaceObject` have the return type `Shareable`—the base type of all shareable interface objects. A client applet must cast the SIO returned to the appropriate subinterface type and store it in an object reference of that type. For example, the wallet applet casts the SIO to `AirMilesInterface`:

```

AirMilesInterface sio = (AirMilesInterface)
JCSys.getAppletShareableInterfaceObject(server_aid, parameter);

```

After the client applet receives the SIO, it invokes the shareable interface methods to access services from the server. However, only the methods defined in the shareable interface are visible to the client applet.

For instance, in the preceding code snippet, even though the `sio` actually points to the `air-miles` applet (its applet class implements the interface `AirMilesInterface`), all instance fields and nonshareable interface methods (such as `methods process`, `select`, and `deselect`) are protected by the firewall.

Following is an example of how the `wallet` applet requests air miles in a debit transaction.

```
package com.smartbank.wallet;

import javacard.framework.*;
import com.fasttravel.airmiles.AirMilesInterface;

public class WalletApp extends Applet {

    private short balance;
    // hardcoded in the applet or assigned at
    // applet personalization
    private byte[] air_miles_aid_bytes = SERVER_AID_BYTES;

    // called by the process method on receiving
    // a DEBIT APDU command
    private void debit(short amount) {

        if ( balance < amount)
            ISOException.throwIt(SW_EXCEED_BALANCE);

        // update the balance
        balance = (short)(balance - amount);

        // ask the server to grant miles
        requestMiles(amount);
    }

    private void requestMiles(short amount) {

        // obtain the server AID object
```

```

        AID air_miles_aid =
            JCSystem.lookupAID(air_miles_aid_bytes,
                               (short)0,
                               (byte)air_miles_aid_bytes.length);

        if (air_miles_aid == null)
            ISOException.throwIt(SW_AIR_MILES_APP_NOT_EXIST);

        // request the sio from the server
        AirMilesInterface sio = (AirMilesInterface)
            (JCSystem.getAppletShareableInterfaceObject(air_miles_aid,
                                                         SECRET));

        if ( sio == null )
            ISOException.throwIt(SW_FAILED_TO_OBTAIN_SIO);

        // ask the server to grant miles
        sio.grantMiles(amount);
    }
}

```

When an error occurs, an applet can throw an `ISOException` by invoking the static method `throwIt` (see the example in the `requestMiles` method). The `throwIt` method throws the JCRE-owned `ISOException` object. Such an object is a JCRE entry point object and can be accessed from any applet's context.

9.2.5.8 Context Switches during Object Sharing

The JCRE, the client applet, and the server applet reside in separate contexts. Context switches must occur to enable object sharing. The client applet calls the `JCSystem.getAppletShareableInterfaceObject` method to request an SIO. An internal mechanism in the method switches the client applet's context to the JCRE context. Then, the JCRE invokes the server applet's `getShareableInterfaceObject` method. Such invocation results in another context switch so that the server applet's context becomes current. On return from both methods, the client applet's context is restored.

Next, the client applet can request service from the server applet by invoking one of the shareable interface methods of the received SIO. During the invocation, the Java Card virtual machine performs a context switch. The server applet's context becomes the currently active context.

Because execution is now in the server applet's context, the code gains access to the protected resources of the server applet—instance fields, other methods,

and even objects owned by the server applet's context. When the service method completes, the client applet's context is restored. Context switching during object sharing is illustrated in Figure 9.5.

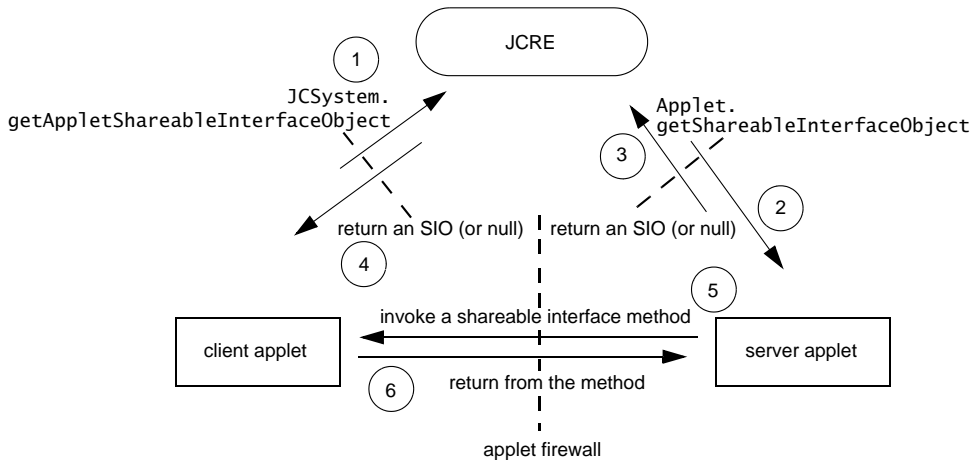


Figure 9.5 Context switch during object sharing

9.2.5.9 Parameter Types and Return Types in Shareable Interface Methods

In the Java programming language, method parameters and return values can be any primitive or reference types. But in the Java Card platform, passing objects (including arrays) as parameters or return values in a shareable interface method is allowed in a limited fashion. For example, if the wallet applet passes one of its own objects as a parameter in the `grantMiles` method, the firewall prevents the air-miles applet from accessing this object. Likewise, if the air-miles applet returns one of its own objects as a return value, the firewall prevents the wallet applet from accessing this object. Although this may seem annoying, it is actually the applet firewall doing its job.

To avoid this problem, the following types of values can be passed in shareable interface methods as parameters and return values:

- *Primitive values*—These are easily passed on the stack. The primitive types supported in the Java Card platform are `boolean`, `byte`, `short`, and (optionally) `int`.
- *Static fields*—Public static fields are accessible from any context. However, objects referenced by such static fields are protected by the firewall.
- *JCRE entry point objects*—Public methods of these objects can be accessed from any context.

- *Global arrays*—These can be accessed from any context. For example, the APDU buffer can be used for this purpose.
- *SIOs*—Sharable interface methods of these objects can be accessed from any context. An SIO returned from a client allows the server context to call back into the client context to obtain some service or data. However, the developer should be careful to avoid excessive context switching (which might reduce performance) and deep nesting of context switches (which might use up extra stack space).

The code in the next section provides an example of passing objects and arrays across the firewall.

9.2.5.10 *Authenticate a Client Applet*

To prevent an unauthorized client from gaining access to protected data, the server applet should authenticate the client applet before granting an SIO and before executing a service method on the SIO.

To determine whether to grant an SIO, a server can check the AID of the requester (the client applet). For instance, the air-miles applet can enforce the following checks in the `getShareableInterfaceObject` method.

```
public class AirMilesApp extends Applet
    implements AirMilesInterface {

    public Shareable getShareableInterfaceObject(AID client_aid,
                                                byte parameter) {

        // assume the wallet AID bytes are preknown
        if (client_aid.equals(wallet_app_aid_bytes, (short)0,
                               (byte)(wallet_app_aid_bytes.length)) == false)
            return null;

        // examine the secret to further authenticate the
        // wallet applet
        if (parameter != SECRET)
            return null;

        // grant the SIO
        return (this);
    }
}
```

When a shareable interface method is next invoked, the server should verify the client applet again. This precaution is necessary because the client applet that originally requests the SIO could break the contract and share the SIO with a third party without getting the proper permission. The server applet must exclude this scenario to protect sensitive data from an untrusted client. To find out the AID of the actual caller, the server applet can invoke the `JCSystem.getPreviousContext-AID` method (explained in 9.2.5.11). The code to detect the caller's identity is added to the `grantMiles` method as follows:

```
public void grantMiles (short amount) {

    // get the caller's AID
    AID client_aid = JCSystem.getPreviousContextAID();

    // check if this method is indeed invoked
    // by the wallet applet
    if (client_aid.equals(wallet_app_aid_bytes, (short)0,
        (byte)(wallet_app_aid_bytes.length)) == false)
        ISOException.throwIt(SW_UNAUTHORIZED_CLIENT);

    // grant miles
    miles = (short)(miles + amount);
}
```

The security of the authentication scheme in the preceding code requires that applet loading be controlled under restricted security measures to avoid applet AID impersonation. Such a scheme may not be sufficient for an applet that requires a higher degree of security. In this case, additional authentication methods must be defined, such as cryptographic exchange.

The following code implements an authentication scheme called challenge-response in the wallet and air-miles applet example. When the service method `grantMiles` is invoked, the air-miles applet generates a random challenge phrase and sends the challenge to the wallet applet. The wallet applet encrypts the challenge and returns a response to the air-miles applet. By verifying the response, the air-miles applet authenticates the wallet applet and adds requested miles to its balance.

To support this scheme, first, the `grantMiles` method is updated to take two additional parameters—an authentication object and a buffer.

```

public interface AirMilesInterface extends Shareable {

    public void grantMiles(AuthenticationInterface authObject,
                           byte[] buffer, short amount);
}

```

The air-miles applet authenticates the wallet applet by invoking the challenge method of the authentication object. The buffer is used to pass challenge and response data.

```

public class AirMilesApp extends Applet
    implements AirMilesInterface {

    public void grantMiles(AuthenticationInterface authObject,
                           byte[] buffer, short amount) {

        // generate a random challenge phrase in the buffer
        generateChallenge(buffer);

        // challenge the client applet
        // the response is returned in the buffer
        authObject.challenge(buffer);

        // check the response
        if (checkResponse(buffer) == false)
            ISOException.throwIt(SW_UNAUTHORIZED_CLIENT);

        miles = (short)(miles + amount);
    }
}

```

Notice that the authentication object is created and owned by the caller—the wallet applet. The applet firewall requires such an object to be an SIO:

```

public interface AuthenticationInterface extends Shareable {

    public void challenge(byte[] buffer);
}

public class WalletApp extends Applet
    implements AuthenticationInterface {

```

```
public void challenge(byte[] buffer) {

    // get response.
    // both challenge and response data are carried
    // in the buffer
    getResponse(buffer);
}

public void process(APDU apdu) {

    if (getCommand(apdu) == DEBIT)
        debit(apdu);
}

private void debit(APDU apdu) {

    short amount = getDebitAmount(apdu);

    // update the balance
    balance = (short)(balance - amount);

    // ask the air-miles applet to grant miles
    requestMiles(apdu.getBuffer(), amount);
}

private void requestMiles(byte[] buffer, short amount) {

    // obtain the AID object
    AID air_miles_aid =
        JCSystem.lookupAID(air_miles_aid_bytes,
                           (short)0,
                           (byte)air_miles_aid_bytes.length);

    // request the SIO from the air-miles applet
    AirMilesInterface sio = (AirMilesInterface)
        (JCSystem.getAppletShareableInterfaceObject(air_miles_aid,
                                                       SECRET));

    // ask the air-miles applet to grant miles
    sio.grantMiles(this, buffer, amount);
}
}
```

In this code example, the buffer used for passing the challenge and response data is the APDU buffer. The APDU buffer is a global array that can be accessed from any context.¹

9.2.5.11 *getPreviousContextAID Method*

During object sharing, a server can find out the AID of the caller applet by invoking the `JCSystem.getPreviousContextAID` method:

```
public AID getPreviousContextAID()
```

This method returns the JCRE-owned AID object associated with the applet instance that was active at the time of the last context switch. In the code example on page 123, when the wallet applet calls the shareable interface method `grantMiles`, a context switch occurs. The `getPreviousContextAID` method returns the AID of the wallet applet that was active before the context switch.

Now consider a more complex scenario, as shown in Figure 9.6. Suppose that two applet instances A and B share a group context. No group context switch occurs if applet A calls a method of object `b` (owned by applet B). This action is allowed regardless of whether object `b` is an SIO.

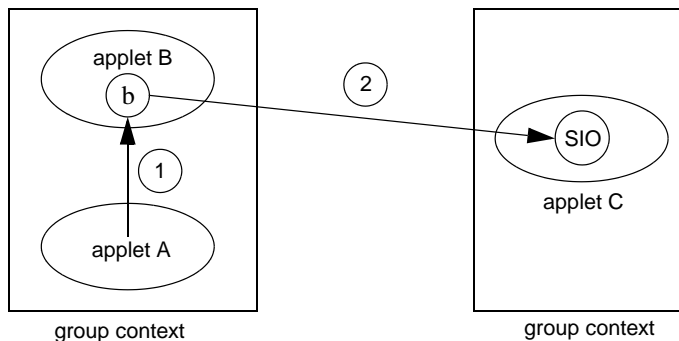


Figure 9.6 Object sharing between applets A, B, and C

Now when object `b` accesses an SIO of applet C, execution moves into a new context. The active applet at the last context switch was applet B, and thus, the `getPreviousContextAID` method returns the AID of applet B.

¹ Even so, there are still limitations in using the APDU buffer to pass structured data. For example, data being passed may not be of the appropriate length or type for the APDU buffer. Both the server and the client may need to put considerable effort to manipulate the APDU buffer.

9.2.5.12 Summary

To conclude, the process of sharing objects between a server applet and a client applet is summarized.

1. If a server applet A wants to share an object with another applet, it first defines a shareable interface SI. A shareable interface extends the interface `javacard.framework.Shareable`. The methods defined in the shareable interface SI represent services that applet A wishes to make accessible to other applets.
2. Applet A then defines a service provider class C that implements the shareable interface SI. C provides actual implementations for the methods defined in SI. C may also define other methods and fields, but these are protected by the applet firewall. Only the methods defined in SI are accessible to other applets.
3. Applet A creates an object instance O of class C. O belongs to applet A, and the firewall allows A to access the fields and methods of O.
4. If a client applet B wants to access applet A's object O, it invokes the `JCSys-tem.getAppletShareableInterface` method to request a shareable interface object from applet A.
5. The JCRE searches its internal applet table for applet A. When found, it invokes applet A's `getShareableInterfaceObject` method.
6. Applet A receives the request and determines whether it wants to share object O with applet B. If applet A agrees to share with applet B, A responds to the request with a reference to O.
7. Applet B receives the object reference from applet A, casts it to type SI, and stores it in object reference SI0. Even though SI0 actually refers to A's object O, SI0 is of type SI. Only the shareable interface methods defined in SI are visible to B. The firewall prevents the other fields and methods of O from being accessed by B. Applet B can request service from applet A by invoking one of the shareable interface methods of SI0.
8. Before performing the service, the shareable interface method can authenticate the client (B) to determine whether to grant the service.

