Smart Cards

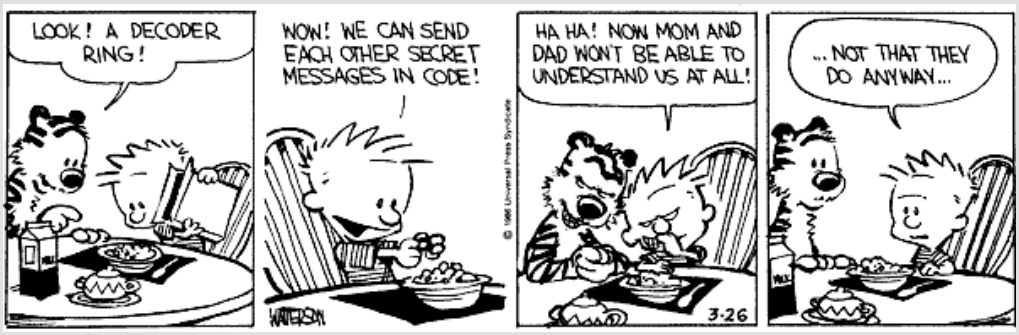
Towards a modern run-time platform

3. Security & Cryptography

Thorsten Kramp & Michael Kuyper
IBM Zurich Research Laboratory

Security vs. Cryptography

- Cryptography
  - *science of information security (greek: kryptos, meaning 'hidden')*
    encryption/decryption, secure hashes, digital signatures, true randomness
  - *cryptology vs. cryptanalysis*
  - *goals: confidentiality, integrity, authentication, non-repudiation*



Security vs. Cryptography

- Computer security
  - *definition*
    - "effort to create a **secure computing platform**, designed so that agents (users and programs) cannot perform actions that they are not allowed to perform, but can perform the actions they are allowed to" [wikipedia]
  - *techniques*
    - cryptography
    - chain of trust

Overview

A. Execution model

language features, sandbox, applet firewall (object isolation and sharing)

B. On-card Cryptography

algorithms and protocols, good cryptographic practice

C. Protecting against attacks

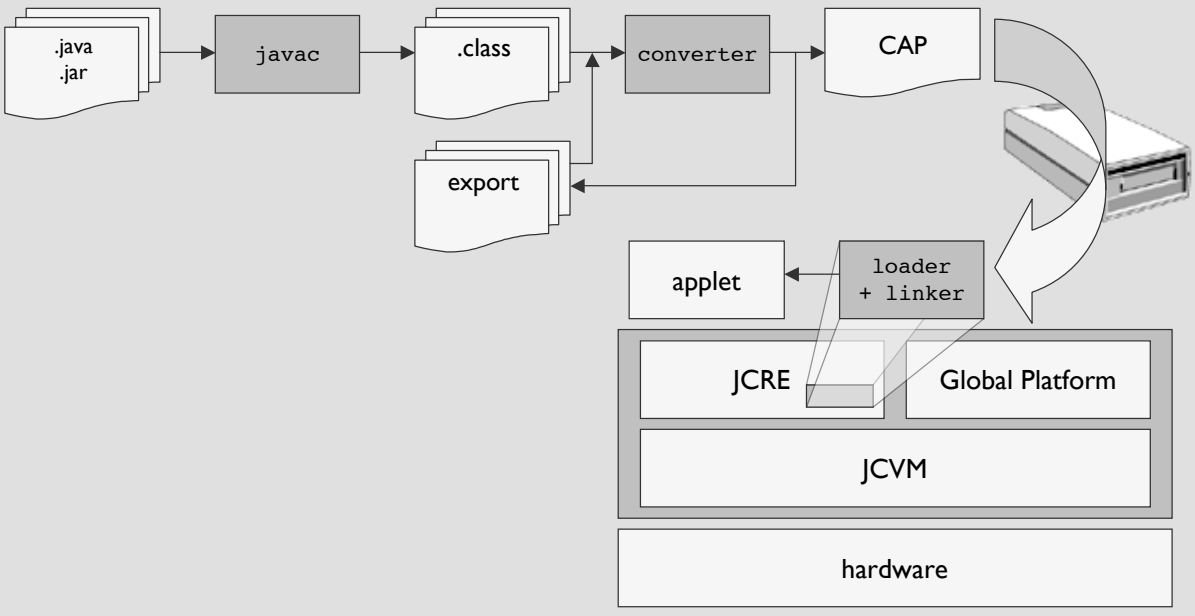
SPA/DPA, timing attacks, fault injection

A. Execution: Language Security

- Java Language Features
  - *strongly typed: no illegal data conversions*
    (static checks at compile time, dynamic checks at run time)
  - *enforced bound checks on array access*
  - *no pointer arithmetic, no way to forge pointers*
  - *no uninitialized variables*
    (default values, compile-time checks)
  - *strictly controlled access levels of fields, methods, and classes*
    (package, `public`, `protected`, `private`)

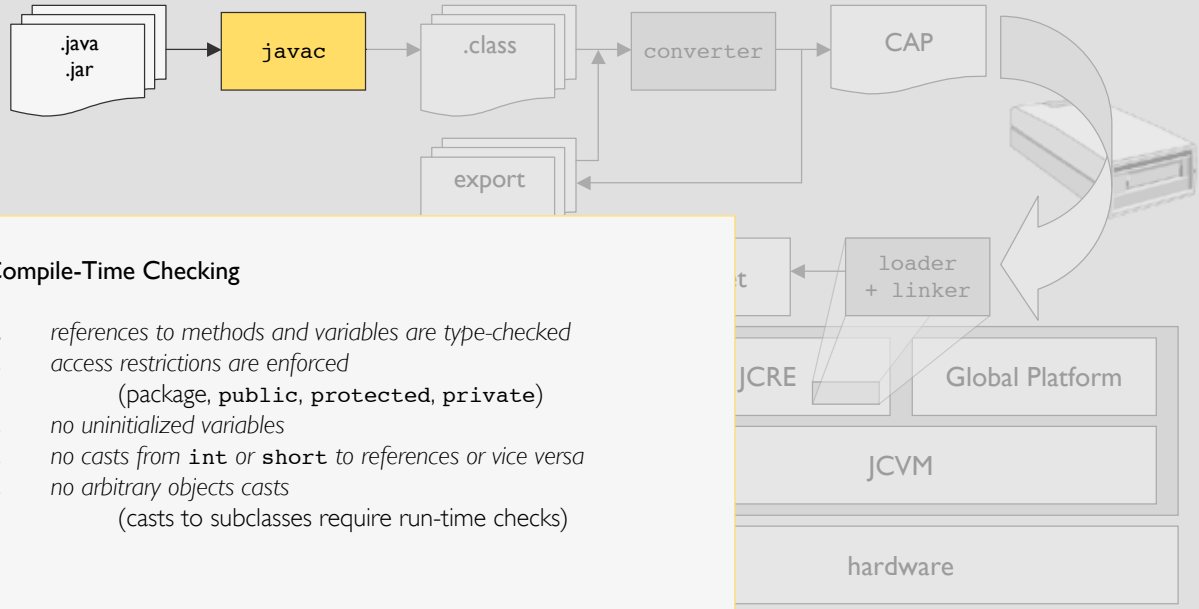
A. Execution: Platform Security

- Security Chain



A. Execution: Platform Security

- Security Chain

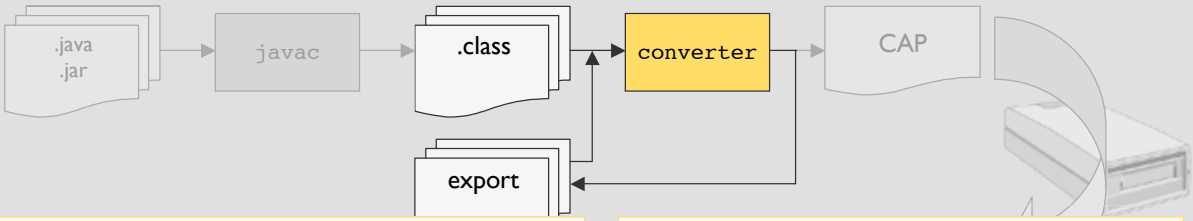


Compile-Time Checking

1. *references to methods and variables are type-checked*
2. *access restrictions are enforced*
   (package, `public`, `protected`, `private`)
3. *no uninitialized variables*
4. *no casts from* `int` *or* `short` *to references or vice versa*
5. *no arbitrary objects casts*
   (casts to subclasses require run-time checks)

A. Execution: Platform Security

- Security Chain



Class File Verification

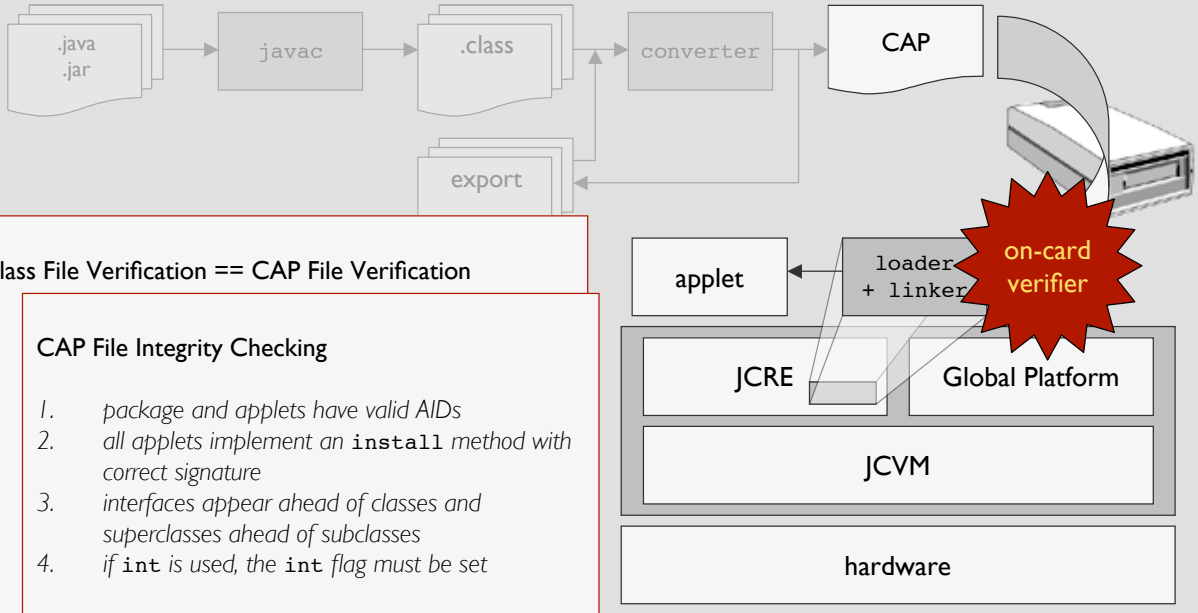
1. *no memory mgmt violations, no stack under-/overflows*
2. *access restrictions are enforced*
3. *methods are called with appropriate arguments*
4. *field are modified with values of appropriate type*
5. *objects are accessed as what they are*
6. *no pointers are forged*
7. *no illegal data conversions*

Subset Checking

1. *no unsupported data types*
   (e.g., `int`, `float`, `double`*)*
2. *no unsupported language features*
   (e.g., threads, multi-dimensional arrays)
3. *operations are within limited ranges*
   (e.g., no more than 255 classes/interfaces, arrays have maximum 32767 fields)

A. Execution: Platform Security

- Security Chain



Class File Verification == CAP File Verification

1.
2.
3.
4.
5.
6.
7.

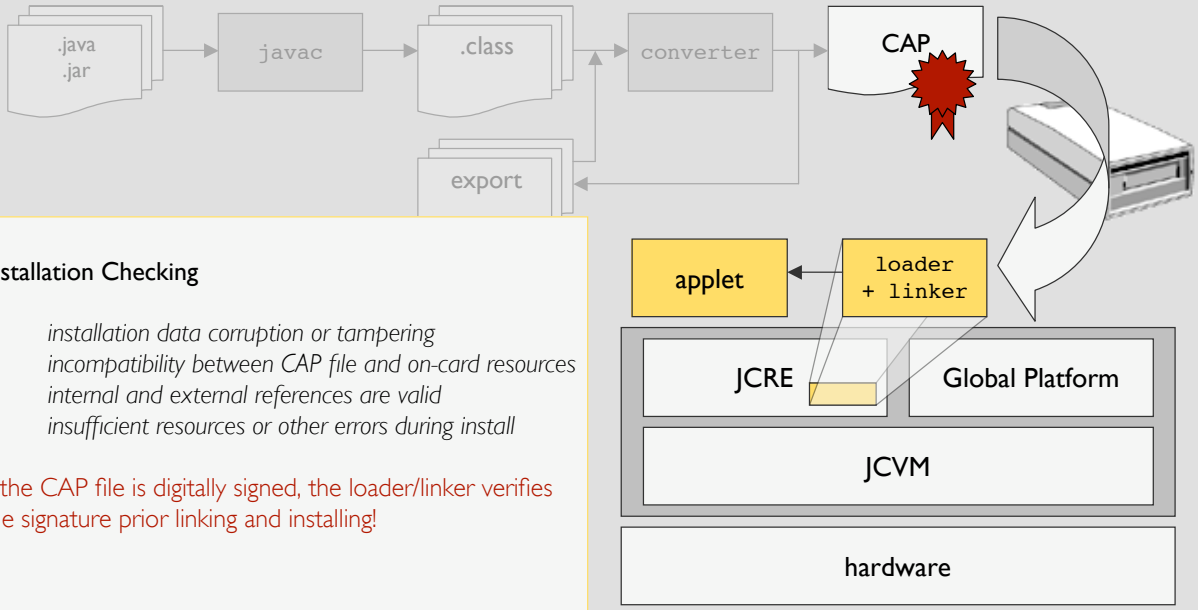
CAP File Integrity Checking

1. *package and applets have valid AIDs*
2. *all applets implement an* `install` *method with correct signature*
3. *interfaces appear ahead of classes and superclasses ahead of subclasses*
4. *if* `int` *is used, the* `int` *flag must be set*

Such an on-card verifier is optional!

A. Execution: Platform Security

- Security Chain



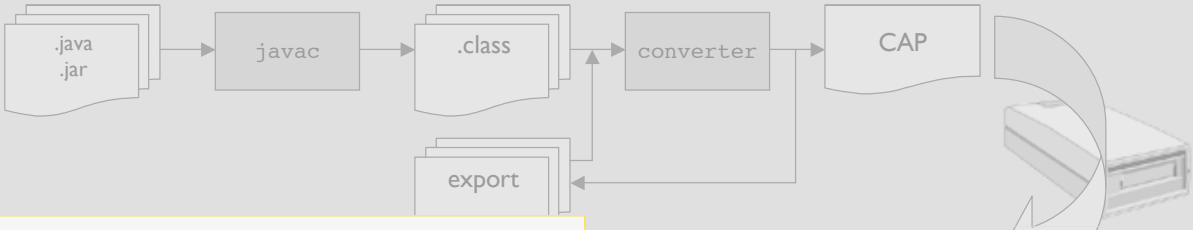
Installation Checking

1. *installation data corruption or tampering*
2. *incompatibility between CAP file and on-card resources*
3. *internal and external references are valid*
4. *insufficient resources or other errors during install*

If the CAP file is digitally signed, the loader/linker verifies the signature prior linking and installing!

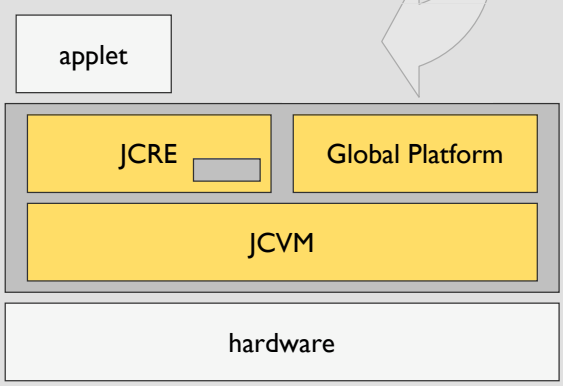
A. Execution: Platform Security

- Security Chain



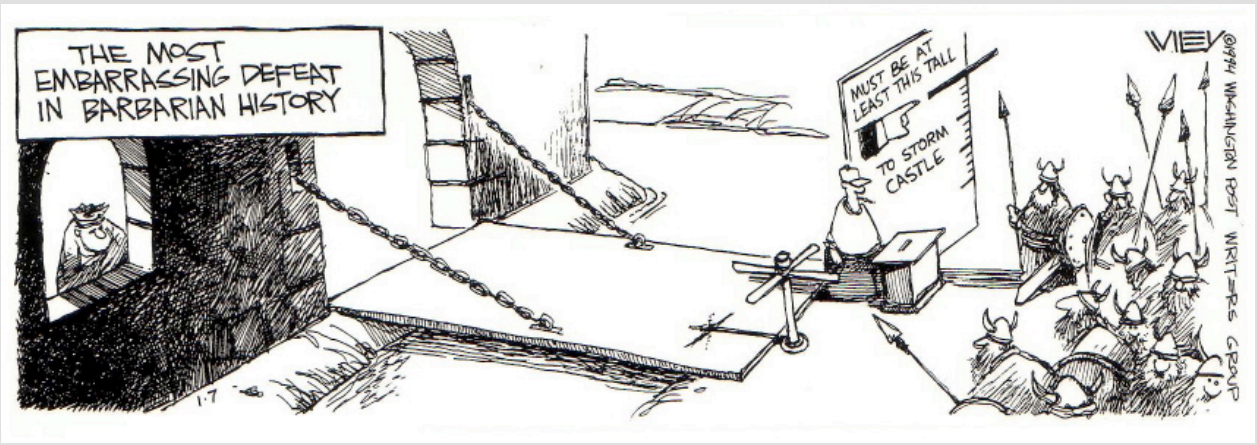
Run-Time Checking

1. *ensuring language type safety*
   (e.g., casts, boundary checks)
2. *applet isolation through an applet firewall*
   (instead of class loaders and security managers as in the Java sandbox model)



A. Execution: Applet Firewall & Object Sharing

- Applet Firewall
  - ***applet isolation:*** *confines an applet to its own designated area and prevents access to the contents or behaviours of objects owned by other applets*
- Object Sharing
  - ***applet cooperation:*** *allows cooperative applets on a single card through a well-defined and secure object sharing mechanism*



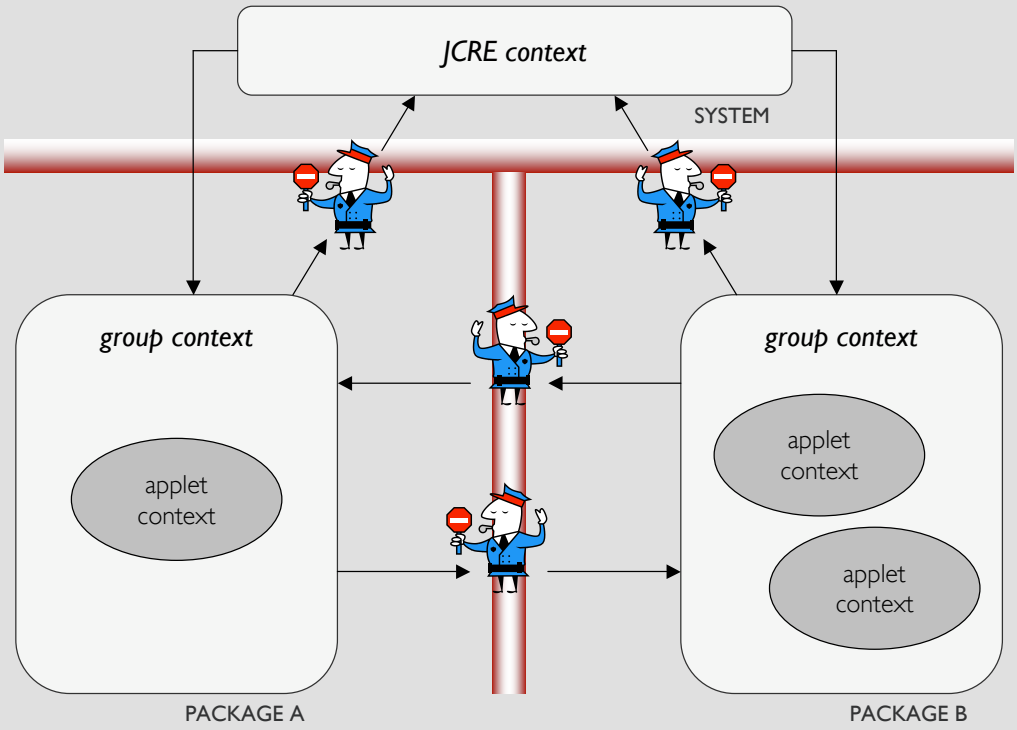
A. Execution: Applet Firewall

- Protects against malfunctioning or "hostile" applets
  developer mistakes and design oversights, hacking attacks
- Partitions the JavaCard object system into separate **contexts**
  - *firewall is the boundary between different contexts*
  - *JCRE assigns applets their context during install*
    - all applet instances of the same package share the same **(group) context** (i.e., object access between applets in the same group context is allowed)
  - *JCRE maintains its own **JCRE context** w/ special privileges*
    - access from the JCRE context to any applet's context is universally allowed
    - access from an applet context to the JCRE context only via **(Temporary) Entry-Point Objects** or **global arrays**

A. Execution: Applet Firewall

- Object Ownership
  - *new objects are owned by the currently active context (exactly one active context at any time)*
  - *primitive static type arrays are owned by the group context of the package (created before any applet instance and initialized by the converter)*
- Transient Arrays and Context
  - *transient arrays are accessible only if the array's owning context is active*
- Static Fields and Methods
  - *NO run-time check when a static fields is accessed or a static method invoked (i.e., static fields and methods are accessible from any context)*
  - *BUT for accessing objects referenced by static fields, the firewall rules apply*
  - *static methods execute in the caller's context (i.e., objects created inside a static method are assigned the caller's context)*

A. Execution: Applet Firewall



DEMO

A. Execution: Object Sharing

- Crossing Context Boundaries

  1. JCRE privileges *2. JCRE entry-point objects*

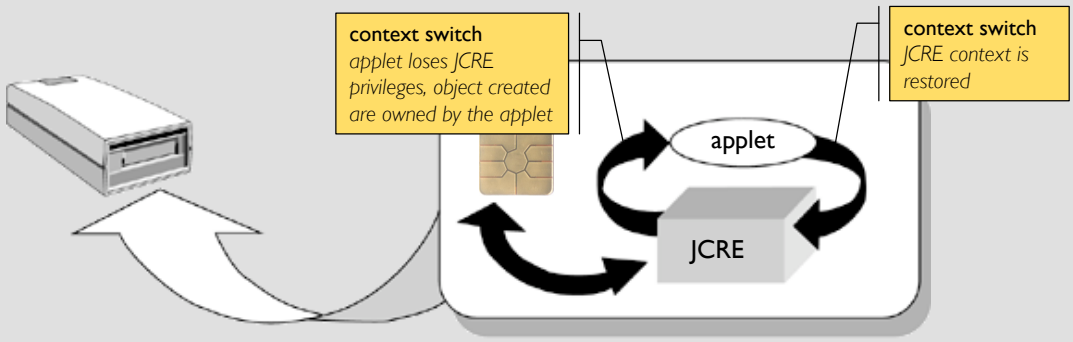
  3. global arrays *4. shareable interfaces*

- Underlying Mechanism: Context Switch
  - *context switches occur during invocation of and return from instance methods of an object owned by a different context (incl. exception exits)*
    - **invocation:** the current context is saved and the new context becomes the currently active context
      - ☛ the invoked method executes with the access right of the new context and all objects created are owned by the new context
    - **return/exit:** the original context is restored and becomes active again
  - *context switches can be nested*
  - *NOTE: Accessing instance fields in a different context is never allowed*

A. Execution: Object Sharing

1. JCRE Privileges
   - *JCRE as "card executive" runs in the system context w/ special privileges*
   - *JCRE context is active after reset*
   - *JCRE may invoke any method on any object (causing a context switch)*
     access any instance field of any object

     e.g., invokes `Applet.process()`, `Applet.install()`, `Applet.select()`, …



A. Execution: Object Sharing

2. JCRE Entry-Point Objects

- *allow applets to request system services to perform privileged system routines*
- *JCRE Entry-Point Objects are objects that…*
  - …are owned by the JCRE context
  - …contain public entry-point methods to be invoked from any context (no fields are accessible, though)
- *invoking an entry-point method causes a context switch to the JCRE context*
- ***Temporary JCRE Entry-Level Objects***
  - references to these object cannot be stored in class or instance variables
  - examples: the `APDU` object, all JCRE-owned exception objects
- ***Permanent JCRE Entry-Level Objects***
  - references to these object can be stored and freely re-used
  - example: the JCRE-owned `AID` objects

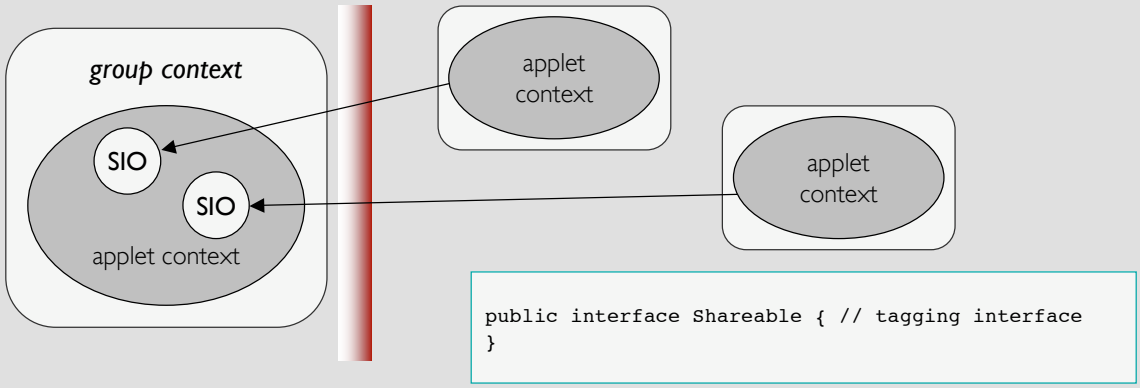
A. Execution: Object Sharing

3. Global Arrays

- *memory buffer shared by the JCRE and all applets (data encapsulated in JCRE Entry-Point Objects is not directly accessible)*
- *arrays of primitive type (can only be designated by the JCRE)*
- *special type of Temporary JCRE Entry-Point Objects: public fields (i.e., array components and array length) can be accessed from any context*
- *public methods are treated as for any other JCRE Entry-Point Object (only method is* `Object.equals()`*, invocation causes context switch)*
- *automatically cleared whenever an applet is selected or before the JCRE accepts a new APDU command*
- *examples:*
  `APDU` *buffer byte array;*
  `Applet.install()` *byte array parameter (=* `APDU` *buffer?!)*

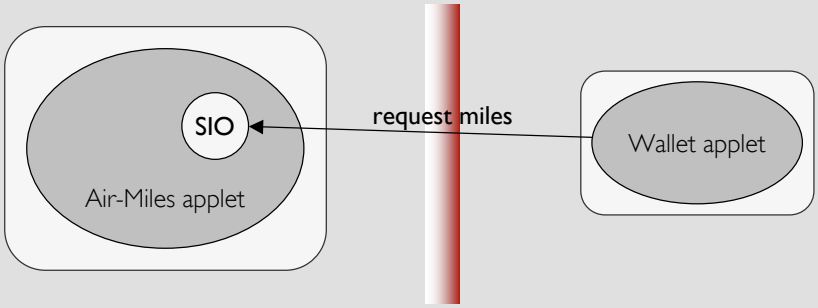
A. Execution: Object Sharing

4. Shareable Interfaces
   - *extends directly or indirectly* `javacard.framework.Shareable`
   - *defines a set of methods available to other applets*
   - *different interfaces allow "wearing a different hat" for different applets*
   - *object implementing shareable interfaces:* ***Shareable Interface Object*** *(SIO)*
   - *class type, instance fields, or other other methods of the SIO are not exposed*



A. Execution: Shareable Interfaces Example

- Combination of a Wallet applet and an Air-Miles applet
  - *Wallet: stores electronic cash*
  - *Air-Miles: provides travel incentives in exchange for miles*
  - *Cooperation: For every $ spent, one air mile is credited*
    1. Air-Miles applet creates an SIO (acts as server)
    2. Wallet applet requests the SIO from the Air-Miles applet (acts as client)
    3. Wallet applet requests miles to be credited by invoking a method of the SIO



A. Execution: Shareable Interfaces Example

```
package com.nevercomebackairlines.airmiles;

import javacard.framework.*;

public interface AirMilesInterface extends Shareable {
   public void grantMiles(short amount);
};

public class AirMilesApplet extends Applet implements AirMilesInterface {
   private short miles;

   public void grantMiles(short amount) {
      miles = (short)(miles + amount);
   }

   ...
}
```


A. Execution: `javacard.framework`

- `javacard.framework.JCSystem`
  - *collection of methods to control applet execution, memory management, atomic transaction management, inter-applet object sharing*
- `javacard.framework.Applet`
  - *abstract base class that defines a JavaCard applet*

A. Execution: javacard.framework

- javacard.framework.JCSystem
  - *collection of methods to control applet execution, memory management, atomic transaction*
- javacard.frame
  - *abstract base class*

```
public final class JCSystem {
   public static AID lookupAID
         (byte[] buffer, short offset, byte length);
   public static AID getPreviousContextAID();

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

   ...
};
```

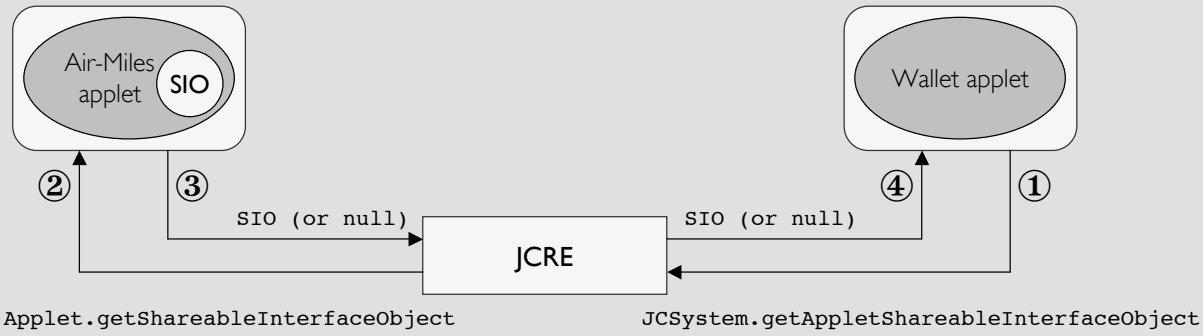

A. Execution: javacard.framework

- javacard.framework.JCSystem
  - *collection of methods to control applet execution, memory management, atomic transaction*
- javacard.frame
  - *abstract base class*

```
public class Applet {
   public Shareable getShareableInterfaceObject
         (AID client_aid, byte parameter);

   ...
};
```

A. Execution: Shareable Interfaces Example

```
package com.nevercomebackairlines.airmiles;

import javacard.framework.*;

public class AirMiles extends Applet implements AirMilesInterface {
   public Shareable getShareableInterfaceObject(AID client_aid, byte parameter) {
      // return the shareable interface object
      return this;
   }

   ...
}
```

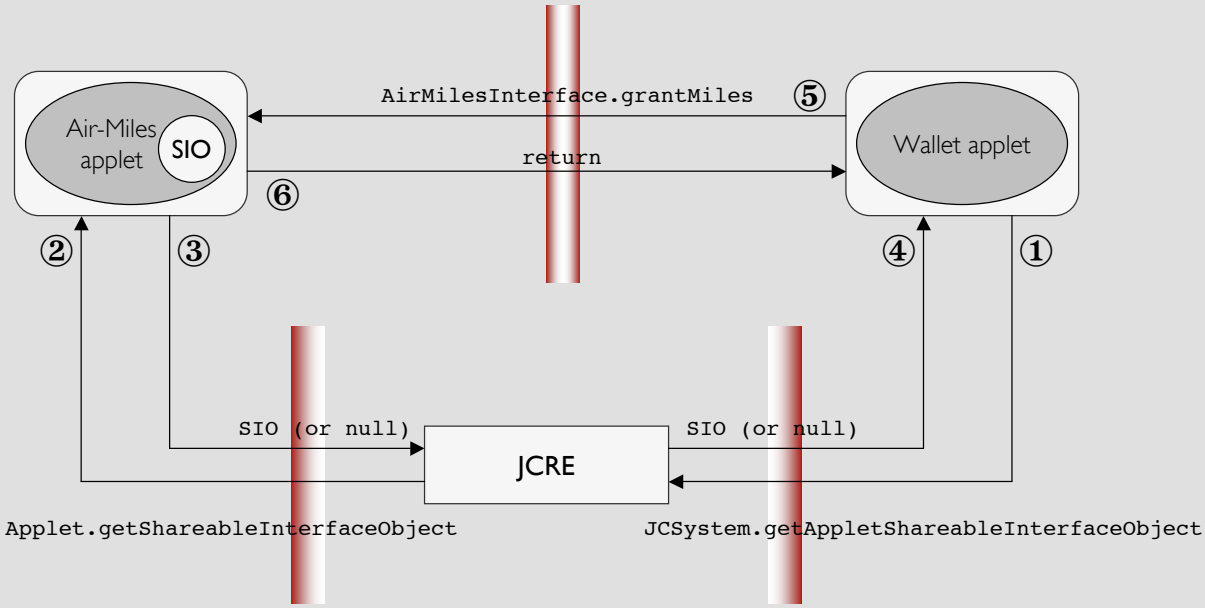


A. Execution: Shareable Interfaces Example

```
package com.wheredidallmymoneygobank.wallet;

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

public class Wallet extends Applet {
   private short balance;

   public void debit(short amount) {
      if (balance < amount)
        ISOException.throwIt(SW_EXCEED_BALANCE);

      balance = (short)(balance - amount);

      AID aid;
      AirMilesInterface sio;

      if ((aid = JCSystem.lookupAID(AIR_MILES_AID,(short)0,AIR_MILES_AID.length)) == null)
        ISOException.throwIt(SW_NO_AIRMILES_APPLET);

      sio = (AirMilesInterface)JCSystem.getAppletShareableInterfaceObject(aid,SECRET);
      if (sio == null)
        ISOException.throwIt(SW_NO_AIRMILES_SIO);

      sio.grantMiles(amount)
   }
   ...
}
```

A. Execution: Shareable Interfaces Example

- Context Switches



A. Execution: Object Sharing

4. Shareable Interfaces: Parameter and Return Types
   - *passing objects (incl. arrays) as parameters or return values does not work because of the object firewall*

     e.g., objects created by the Wallet applet are not accessible by the AirMiles applet
   - *the following types can be passes in shareable interface methods*
     - **primitive values:** passed on the stack
     - **static fields:** public static fields are accessible from any context (but objects referenced by such static fields are protected by the firewall)
     - **JCRE entry-point objects:** public methods are accessible from any context
     - **global arrays:** accessible from any context
     - **SIOs:** shareable interface methods are accessible from any context (allows call backs from the server to the client)

A. Execution: Shareable Interfaces Example

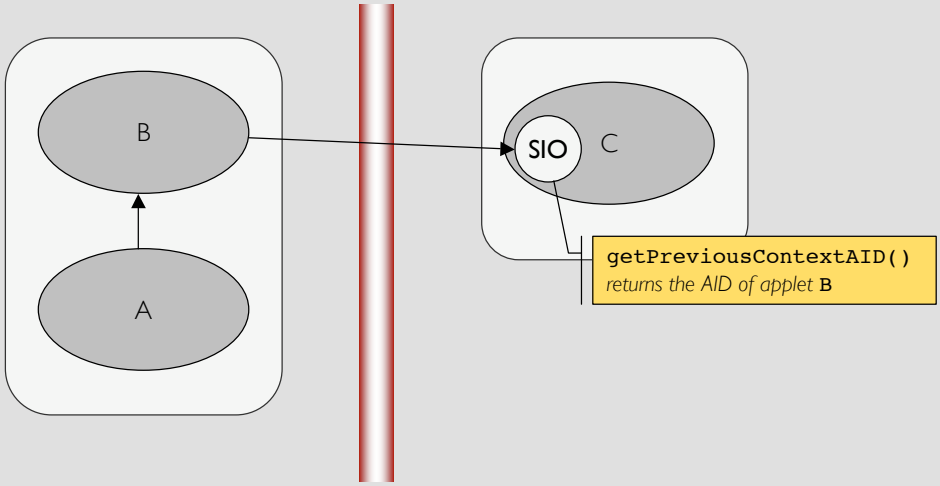
- Authentication of the Client Applet (simple)

```
package com.nevercomebackairlines.airmiles;

import javacard.framework.*;

public class AirMiles extends Applet implements AirMilesInterface {
   public Shareable getShareableInterfaceObject(AID clientAID, byte param) {
      // assume that the Wallet AID is known
      if ((!clientAID.equals(WALLET_AID,(short)0,WALLET_AID.length) || (param != SECRET))
        return null;

      // return the shareable interface object
      return this;
   }

   public void grantMiles(short amount) {
      AID clientAID = JCSystem.getPreviousContextAID();
      if (!clientAID.equals(WALLET_AID,(short)0,WALLET_AID.length)
        ISOException.throwIt(SW_UNAUTHORIZED_CLIENT);

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

   ...
}
```


A. Execution: Object Sharing

4. Shareable Interfaces: `JCSystem.getPreviousContextAID()`

   returns the JCRE-owned AID object associated with the applet that was active at the time of the last context switch



A. Execution: Shareable Interfaces Example

- Authentication of the Client Applet (advanced)

```
package com.nevercomebackairlines.airmiles;

import javacard.framework.*;
import com.wheredidallmymoneygobank.wallet.AuthenticationInterface;

public interface AirMilesInterfaces extends Shareable {
   public void grantMiles(AuthenticationInterface auth, byte[] buffer, short amount);
}

public class AirMiles extends Applet implements AirMilesInterface {
   public void grantMiles(AuthenticationInterface auth, byte[] buffer, short amount) {
      generateChallenge(buffer);

      auth.generateResponse(buffer);

      if (!checkResponse(buffer))
        ISOException.throwIt(SW_UNAUTHORIZED_CLIENT);

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

   ...
}
```


A. Execution: Shareable Interfaces Example

- Authentication of the Client Applet (advanced, cont'd)

```
package com.wheredidallmymoneygobank.wallet;

public interface AuthenticationInterface extends Shareable {
   public void generateResponse(byte[] buffer);
}

public class Wallet extends Applet implements AuthenticationInterface {
   public void generateResponse(byte[] buffer) {
      ...
   }

   public void debit(short amount) {
      ...

      sio.grantMiles(this,APDU.getCurrentAPDUBuffer(),amount)
   }

   ...
}
```

DEMO