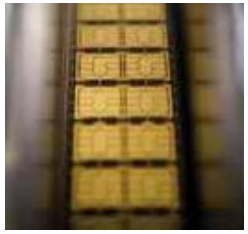
Java Virtual Machine
Advanced features

Carte à puce et Java Card

2010-2011

Jean-Louis Lanet

Jean-louis.lanet@unilim.fr



Université de Limoges

SUFOP Service Universitaire de Formation Permanente

Agenda

- Atomicity
- Sharing mechanism
- Garbage collection
- Logical channel
- Byte code verification
- RMI

Atomicity and transactions

- Smart cards are emerging as a preferred device in such applications as
  - storing personal confidential data and
  - providing authentication services in mobile and distributed environment
- With smart card, there is a risk of failure at any time during applet execution.
  - a computational error,
  - a user may accidentally remove card from the reader, cutting off the power supply to the card CPU.

Atomic Operations:

- JCRE provides a robust mechanism to ensure atomic operations:
  - The java card platform guarantees that any update to a single field in a persistent object or a single class field is atomic.
  - The java card platform supports a transactional model, in which an applet can group a set of updates into a transaction.

Atomicity (I)

- Atomicity means that any update to a single persistent object field (including an array element) or to a class field is guaranteed to either complete successfully or else be restored to its original value if an error occurs during the update.
- The concept of atomicity apply only to the contents of persistent storage.

Atomicity (II)

- Atomicity defines how the JCRE handles a single data element in the case of an error (power loss) occurs during an update to that element
- JCRE atomicity feature does not apply to transient arrays.
- After an error occurs, the transient array is set to default values (zero, false or null).

Block data updates in an array

- The `javacard.framework.Util` class provides methods for block data updates (like in Java) :
  1. `arrayCopy`
  2. `arrayCopyNonAtomic`
  3. `arrayFillNonAtomic`

`Util.arrayCopy` method

- Syntax:
  - `public static short arrayCopy(byte[] src, short srcOff, byte[] dest, short desOff, short length)`
- Guarantees that either all bytes are correctly copied or the destination array is restored to its previous byte values.
- Note that if the destination array is transient, the atomic feature does not hold.
- `arrayCopy` requires extra EEPROM writes to support atomicity (slow)

Util.arrayCopyNonAtomic

- Syntax:
  - `public static short arrayCopyNonAtomic(byte[] src, short srcOff, byte[] dest, short desOff, short length)`
- Does not use transaction facility.

Util.arrayFillNonAtomic

- Syntax:
  - `public static short arrayFillNonAtomic(byte[] bArray, short bOff, short bLen, byte bValue)`
- Non-atomically fills the elements of a byte array with specified value.

Transactions

- Atomicity guarantees atomic modifications of a single data element
- However, an applet may need to atomically update several different fields in several different objects
- For example: credit or debit transaction
  - Increment the transaction number
  - Update the purse balance
  - Write a transaction log

Transactions

- Java card technology supports a similar transactional model, with commit and rollback
- It guarantees that complex operations can be accomplished atomically; either they successfully complete or their partial results are not put into effect.

Commit Transactions

```
    // begin a transaction
JCSystem.beginTransaction();
    //all modifications in a set of updates of
    //persistent data are temporary until
    //the transaction is committed
    ..............
   //commit a transactions
JCSystem.commitTransaction();
```

Abort Transaction

- Transactions can be aborted either by an applet or by the JCRE (by calling `JCSystem.abortTransaction` method)
- Aborting a transaction causes the JCRE to throw away any changes made during the transaction and restore conditionally update fields or array elements to their previous value
- A transaction must be in progress when the `abortTransaction` method is invoked; otherwise, the JCRE throws a `TransactionException`.

Abort Transaction

- If power is lost or an error occurs during a transaction, the JCRE invokes a JRCE internal rollback facility the next time the card is powered on to restore the data involved in the transaction to their pre-transaction value

Nested Transaction

- Transaction in the Java Card platform cannot be nested (except JC3.0 co.-ed).
- There can be only one transaction in progress at a time. (due to the limited computing resources of smart card.)
- If `JCSystem.beginTransaction` is called while a transaction is already in progress, JCRE throws a `TransactionException`

Commit capacity (I)

- To support the rollback of uncommitted transactions, the JCRE maintains a commit buffer where the original contents of the updated fields are stored until the transaction is committed.
- The size of the commit buffer varies, depending on the available card memory.
- Due to the size of commit buffer, putting too many things in a transaction may not be possible

Commit capacity (II)

- Before attempting a transaction, an applet can check the size of the available commit buffer against the size of the data requiring an atomic update
- JCSystem provides two methods
  - `JCSystem.getMaxCommitCapacity()` return the total number of bytes in the commit buffer.
  - `JCSystem.getUnusedCommitCapacity()` returns the number of unused bytes left in the commit buffer.

Commit capacity (III)

- In addition to storing the contents of fields modified during a transaction, the commit buffer holds additional bytes of overhead data (location of the fields).
- The amount of overhead data depends on the number of fields being modified
- If the commit capacity is exceeded during a transaction, the JCRE throws a `TransactionException`. Even so, the transactions is still in progress unless it is explicitly aborted by the applet or by the JCRE.

TransactionException

- JCRE throws a `TransactionException` if
  - nested transaction
  - commit buffer overflow
- `TransactionException` provides a reason code to indicate the cause of the exception:
  - IN_PROGRESS: `beginTransaction` was called while a transaction was already in progress
  - NOT_IN_PROGRESS: `commitTransaction` or `abortTransaction`
  - BUFFER_FULL:
  - INTERNAL_FAILURE

Local variables and Transient Objects during a transaction

- Update to transient objects and **local variables are never undone** regardless of whether or not they were inside a transaction
- See the example 1 in next page
  - key_buffer: transient object
  - a_local: local variable
- See the example 2
  - Ref_1: reference to a transient object
  - Ref_2: reference to a persistent object

EXAMPLE1

```
Byte[] key_buffer = JCSystem.makeTransientByteArray
(KEY_LENGTH, JCSystem.CLEAR_ON_RESET);
JCSystem.beginTransaction();
Util.arrayCopy (src, src_off, key_buffer, 0,
   KEY_LENGTH);
Util.arrayCopyNonAtomic (src, src_off, key_buffer, 0,
   KEY_LENGTH);

For(byte i =0; i<KEY_LENGTH; i++)
    key_buffer[i]=0;
Byte a_local = 1;
JCSystem.abortTransaction();
```

A_local = ?

EXAMPLE2

```
JCSystem.beginTransaction();
// ref_1 refers to a transient object
ref_1 = JCSystem.makeTransientObjectArray(LENGTH,
   JCSystem.CLEAR_ON_DESELECT);
// ref_2 refers to a persistent object
ref_2 = new SomeClass();
if(!condition)
 JCSystem.abortTransaction();
else
 JCSystem.commitTransaction();
return ref_2; // return null if abortTransaction
```

Applet isolation

- The firewall provides a strong isolation mechanism between contexts
- Shareable interfaces are a feature in the API to enable applet interaction through a Shareable Interface Object (SIO),
- From the owning context, the SIO is a normal object whose fields and methods can be accessed,
- To any other context, only the methods defined in the shareable interface are accessible,
- All other fields and methods are protected by the firewall.

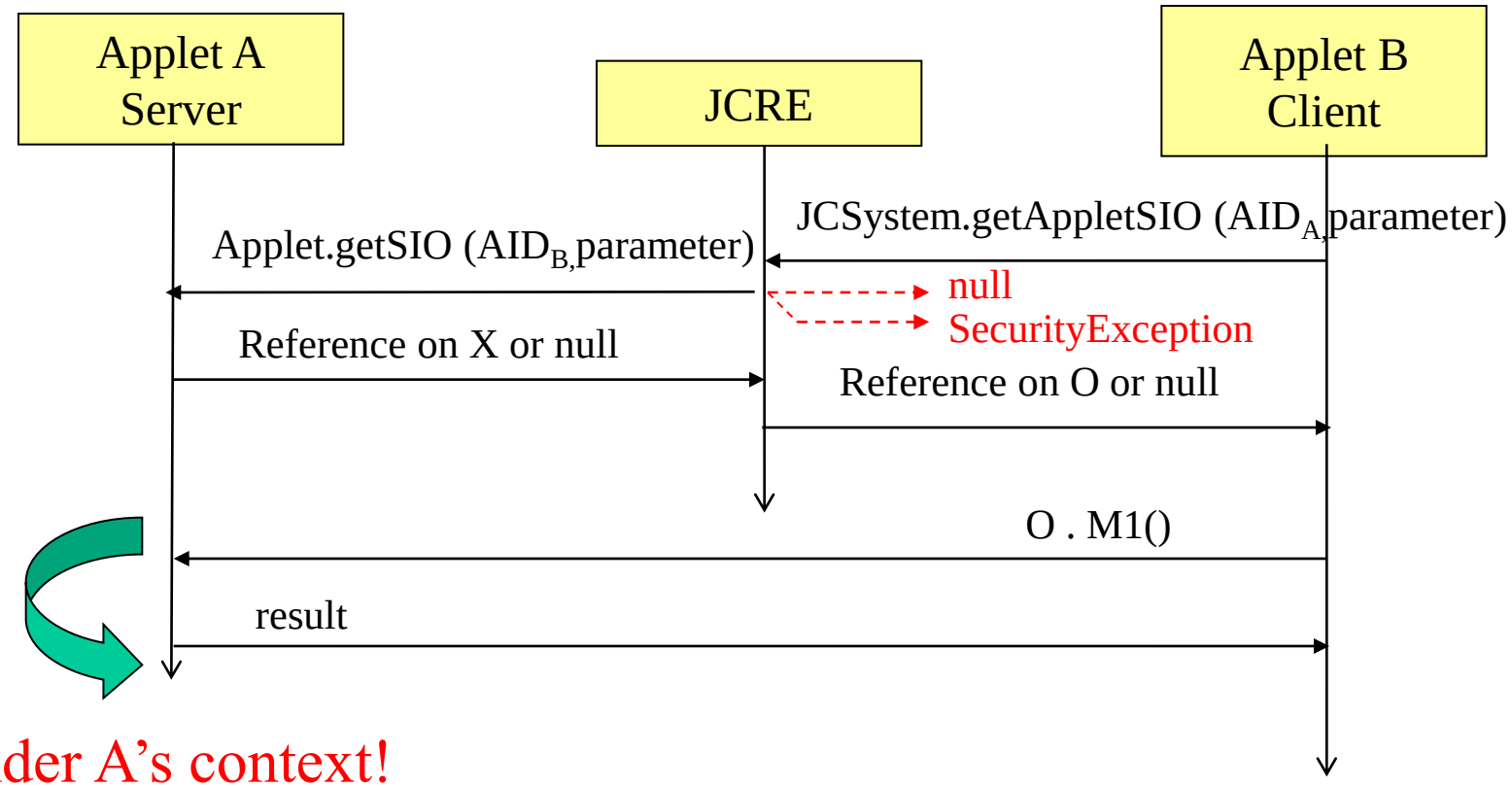
Sharing mechanism

- Methods of an object that implements a shareable interface (SI) can be invoked through the firewall
- Once shared an object (SIO) can't be un-shared
- However the caller needs to obtain a reference to this object
  - Applet *A* agree to share with applet *B*
  - Applet *A* declares to implement a shareable interface and implements the services
  - Applet *B* access the services by obtaining an object reference and invoking the service methods.

Server Applet A

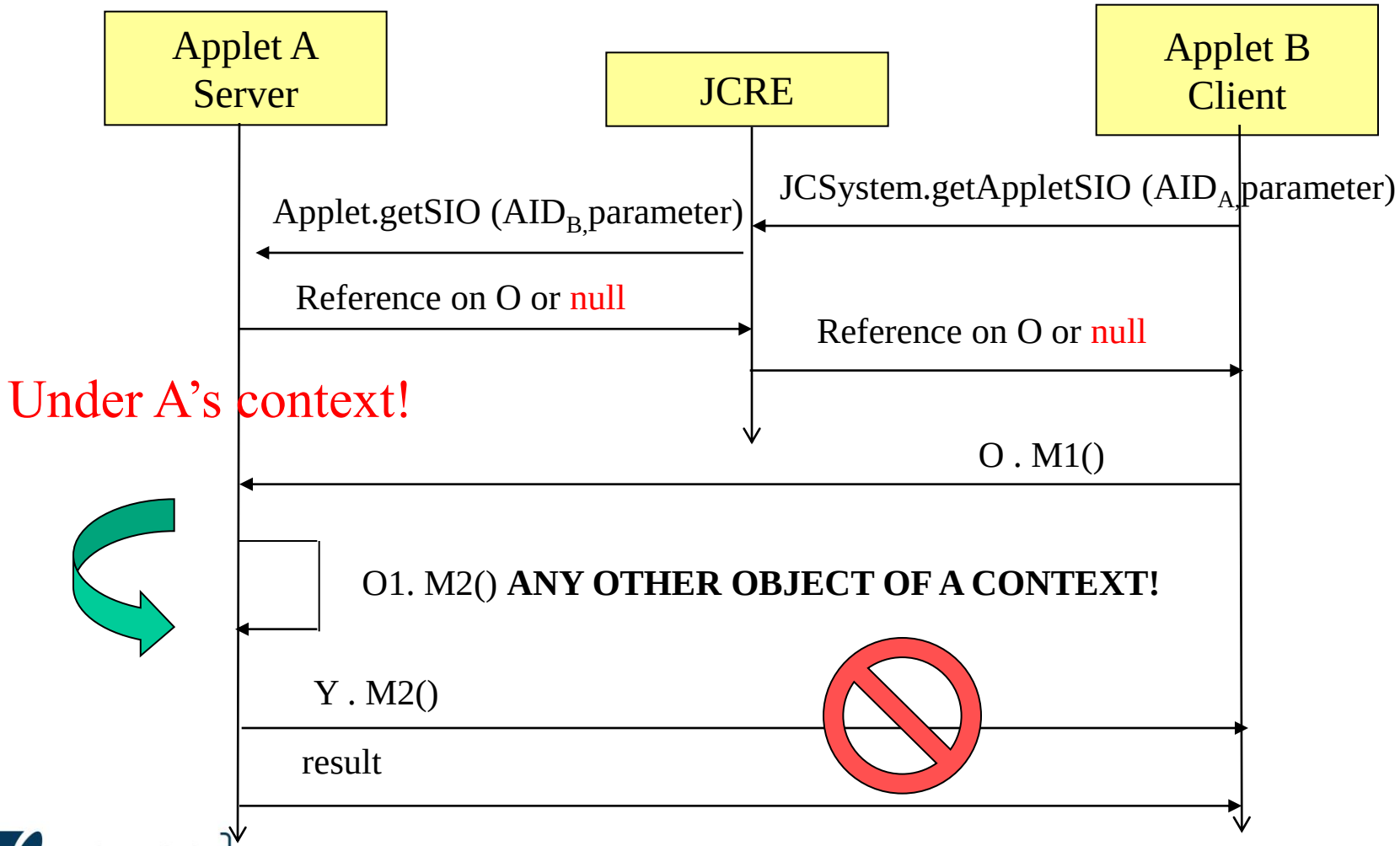
- *A* defines a Shareable Interface (SI) that extends the interface **`javacard.framework.Shareable`**
- Defines a class *C* that implements SI with the methods defined in SI,
- Defines other methods protected by the firewall
- *A* creates an object instance of class *C*

Shareable Interfaces



Under A's context!

Shareable Interfaces



Example server side

The server must create a shareable interface that extends
javacard.framework.Shareable

```
package myPackage
import javacard.framework.Shareable
public interface loyaltyInterface extends Shareable {
  public void grantPoint (short amount)
}
```

- The server implements the interface

```
package myPackage
public class loyaltyApp extends Applet implements loyaltyInterface {
   private short (miles);
   public void grantPoint (short amount) {
        miles= (short) (miles+amount);
   }
 }
```

Example client side

- The client needs to know the server AID,

```
import myPackage;
public class clientApp extends Applet {
// request SIO from the server
loyaltyInterface sio = (loyaltyInterface)
  JCSystem.getAppletShareableInterfaceObject (AIDServer,
  (byte) 0)
if (sio ==null){ISOException.throwIt();}
sio.grantPoint(theAmount);
```

- The JCRE looks up the server applet and invokes the server `getShareableInterfaceObject` with the client AID as parameter !!!

Example server side revisited

The server must create a shareable interface that extends `javacard.framework.Shareable`

```
package myPackage
import interface loyaltyInterface extends Shareable {... }
```

- The server implements the interface

```
package myPackage
public class loyaltyApp implements loyaltyInterface{ private
   short (miles);
  public void grantPoint (short amount) {
   miles= (short) (miles+amount) ;
  }
 public Shareable getShareableInterfaceObject
   (ClientAID, byte param){
    if (clientAID.equals( myFriend)){
    return ((Shareable)this)}
   else {return null}
  }
```

Example client side revisited

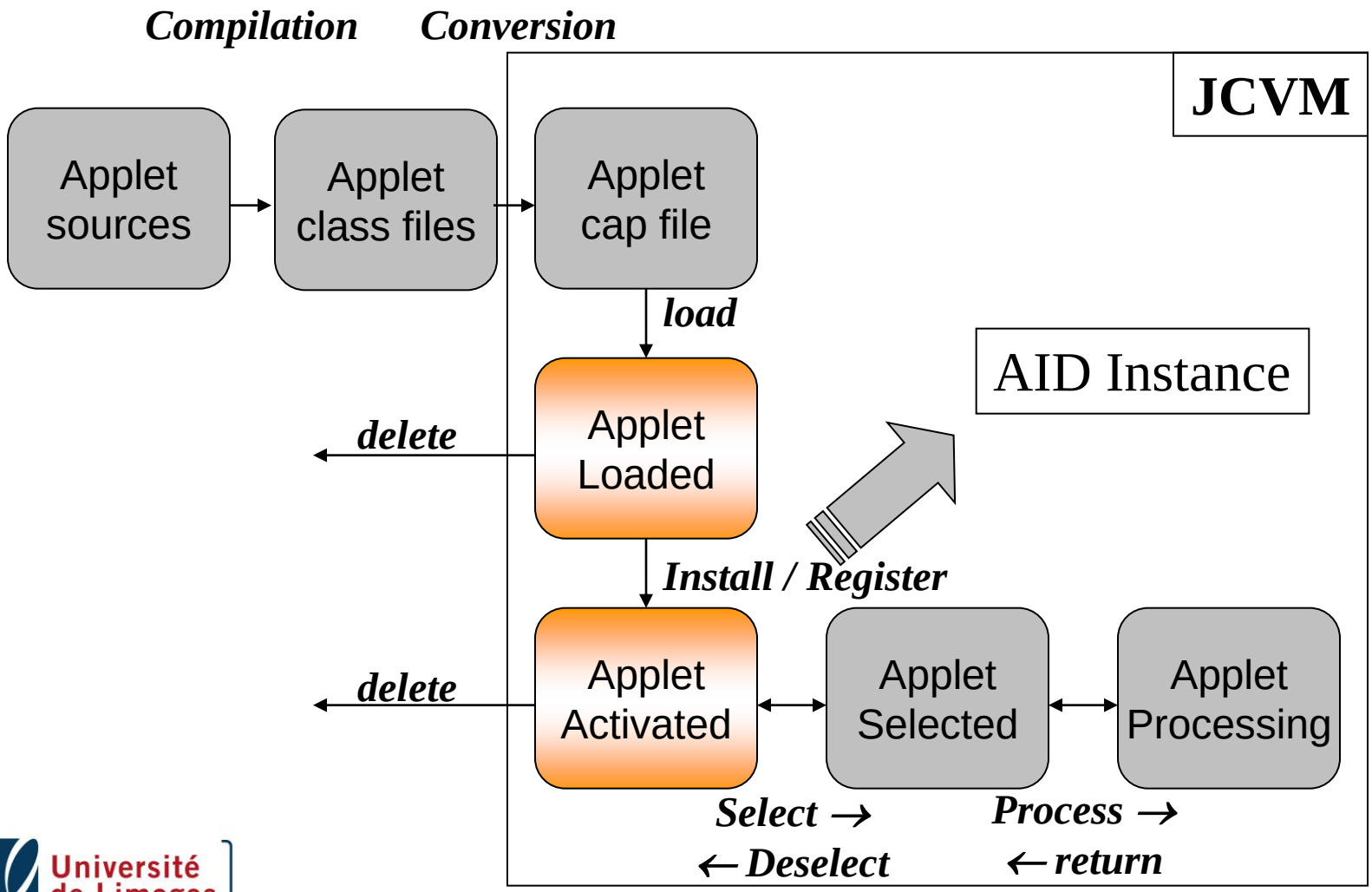
- The client needs to know the server AID,

```
import myPackage;
public class clientApp extends Applet {
// request SIO from the server
loyaltyInterface sio = (loyaltyInterface)
  JCSystem.getAppletShareableInterfaceObject
  (AIDServer, (byte) 0)
if (sio == null){ISOException.throwIt();}
sio.grantPoint(theAmount);
```

Agenda

- Sharing mechanism
- Garbage collection
- Logical channel
- Byte code verification
- RMI

Java Card applet life cycle



Applet Installation

- Applet installer is an optional part of the JCRE (if no post issuance)
- If the installer is included, then the deletion manager is mandatory,
- To the terminal the installer appears to be an applet (AID, select,…) but doesn't need to be installed as an applet,
- Installer applet privileges are the same as the RTE,
  - Read and write directly in memory,
  - Access object owned by other applets,
  - Invoke non-entry point methods of the RTE,
  - Able to invoke the install method of a new applet.

Deletion manager

- To the terminal the DM appears to be an applet,
- Same as the installer it doesn’t need to be implemented as an applet,
- Same privileges as the installer,
- An applet instance deletion may be unsuccessful :
  - An object owned by the applet instance is referenced from an object owned by another applet instance,
  - An object owned by the applet instance is referenced from a static field,
  - An applet instance belonging to the context is active on the card
- Applet instance deletion must be atomic,
- The resource used by the deleted instance may be recovered for future use.

Applet deletion

- Request deletion sent in the form of an APDU,
- To be eligible for deletion, no object on the card should have dependencies on the applet to be deleted,
- Warning static objects belong to the package not the applet…
- Call the uninstall () of the `javacard.framework.AppletEvent` interface
  - Must implement `AppletEvent` interface
  - Before applet deletion,
  - Allow to release resources such as static objects and shared keys,
  - Backup data into another applet space.
  - Notify all other dependant applets,

Guidelines

- Calling uninstall method does not guarantee that the applet will be deleted,
  - Other applet are still dependent on this applet,
  - A tear occurs before the deletion element are processed
- Implement the uninstall method defensively:
  - The applet continues to function consistently and securely if deletion fails,
  - Applet can withstand a possible tear during the execution,
  - The uninstall method can be called again if deletion is reattempted.

Requesting the uninstall

```
private class test extends Applet implements AppletEvent{
  private boolean disableApp = false;
  ...
  public void uninstall () {
    if (!disableApp){
   // to protect against tear
        JCSystem.beginTransaction ();
        disableApp = true; // marked uninstalled
        ... Remove dependency
        JCSystem.commitTransaction();
    }
   }
  public boolean select (){
    if (disableApp) return false ;
  return true;
  }
```

Object deletion

- Allows to recover unreachable objects,
- Such an object can neither be pointed to by a static field nor by an object field,
- Objects in persistent memory and potentially volatile memory,
- Needs eeprom write operations, slow and security marker.
- GC “on request”, optional,
- Initiated by the applet but activated by JCRE at the return of process command
- The API in `javacard.framework.JCSystem` contains two static methods:
  - `isObjectDeletionSupported`, inform applet if object deletion is supported by the platform,
  - `requestObjectDeletion` (), invoke the mechanism.

Guidelines

- When throwing exception avoid creating exception objects and rely on the GC to perform clean up,
- Do not create objects in method or block scope, the Object deletion mechanism should not be considered as a GC !
- Use the deletion mechanism when a large object such as a certificate or key must be replaced by a new one
- Use the object deletion mechanism when object resizing is required.

Requesting the deletion

```
Void updateBuffer (byte requiredSize)
try {
       if (buffer != null && buffer.length == requiredSize)
       {return}
       JCSystem.beginTransaction ();
       byte[] oldBuffer = buffer;
       if (oldBuffer != null)
          JCSystem.requestObejctDeletion();
       JCSystem.commitTransaction ()
    } catch (Exception e) {
   JCSystem.abortTransaction(); }
}
```

Agenda

- Sharing mechanism
- Garbage collection
- Logical channel
- Byte code verification
- RMI

Logical Channels

- In Java Card the session-specific security data of an application is available only when selected,
- When deselected, part of RAM is cleared (keep in mind the transient object) and session keys for example are lost,
- In a multi application card, need to switch from an application to another,…
- Problems :
  - Java Card is currently mono threaded
  - Not possible to perform a new action while the previous one is not completed
  - An application shall be selected to process an APDU command

Handling channel information on APDU commands

A terminal can start up to 20 sessions (JC 2.2.2, 4 only with JC 2.2.1)

- Each session uses a logical channel,
- Session are associated with IO interface, one for contact and one for contact less communication (ISO 14443),
- An applet instance can receive APDU from both IO interface, so an applet can be connected up to 40 logical channels,
- If two APDU arrives concurrently, the contact less session has the higher priority,
- Power loss on the contacted interface implies a card reset even a contact less session is in progress.

Channel 0 is the default channel for both IO interface,

- Opened at card reset and cannot be closed,
- All other channels are closed at reset.

Applet selection

Multiplex command into the APDU channel, if no sharing.

If sharing is needed, use multi selectable applets…

Only specific APDU commands can contain encoded logical information

An applet can be selected on one logical channel, but also through several channels (contact less or not),

Use of the CLA byte,

- If CLA = “0000 00cc” then the two least bits (cc) are used for the logical channels which range from 0 to 3, i.e. :0x0X and 0x1X
- If CLA = “0100 cccc” then the four least bits (cccc) are used for the logical channels which range from 4 to 19, i.e. 0x4Y, 0x5Y, 0x6Y and 0x7Y.
- Card compliant with 2.2.2 spec must also support proprietary class value 0x8X, 0x9X, 0xAx and 0xBX (channel 0 to 3) and 0xCY, 0xDY, 0xEY and 0xFY (Channel 4 to 19).

Default applet

Currently selected Applet : the applet which is responding to an APDU

Normally an applet can only be selected if a command SELECT FILE is successfully treated,

Some applet need a default applet to be selected after reset, specially for opening a new logical channel,

Logical channels may share the same instance applet as the default applet instance.

Card reset

After a reset the JCRE performs initialization and checks the presence of a default applet,

- It provides the channel 0 to this applet,
- Select this applet and execute the select() method of this applet
- If an exception is thrown or returns false, the JCRE must indicate that no active applet are available on channel 0,
- The JCRE must send an ATR.

Applet selection

Two possibilities

- Using a command APDU MANAGE CHANNEL OPEN
  - Open a channel from an already opened channel and selection of a default applet on that channel,
  - If P2 equals 0, let the JRTE chose the channel in other case P2 must be in the range 0..19.
  - Call the `select` method or `MultiSelectable.select` if required,
  - In case of an exception or a return false, the channel is closed (status code 0x6999),
- With a command APDU SELECT FILE
  - Select an applet on a new logical channel

In both cases, the CLA byte must specify either the channel to be open or the open channel

- INS=0x70 for the command APDU MANAGE CHANNEL OPEN
- INS=0xA4 for the command APDU SELECT FILE

Deselecting an applet

While receiving a command APDU MANAGE CHANNEL CLOSE (INS=0x70)

While selecting another applet using a SELECT FILE,

Or selecting the same applet on another channel specified in the CLA byte,

The RTE call the `MultiSelectable.deselect` (`appInstStillActive`) method, where the `appInstStillActive` parameter is set to true if the same applet instance is still active on another logical channel,

Multi selectable applet

- Multi-selectable applets shall implement the interface `javacard.framework.MultiSelectable`
- In such a case, the following methods are invoked during selection and de-selection respectively,
  - `MultiSelectable.select ()`
  - `MultiSelectable.deselect()`
- When a multi-selected applet instance is deselected from of the logical channels the method `MultiSelectable.deselect` is called
- When this instance is the last active, the `Applet.deselect` is called

Agenda

- Sharing mechanism
- Garbage collection
- Logical channel
- Byte code verification
- Software based attacks

JVM offensive vs. defensive

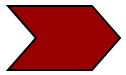
- Sun defines only a byte code format and a verification algorithm,
- It is possible to code a defensive virtual machine,
- Until now a few cards implement BC verification or partially
- If post issuance is allowed without any cryptographic means to authenticate the sender, it becomes unavoidable to implement it
- Often some of the test are implemented in a hybrid way

Exemple

Consider the following code fragment:

```
private int monIncrement ;
public int increment(int base) {
        int resultat ;
        resultat = base + monIncrement ;
        return resultat + 1 ;
   }
```

Execution of byte code



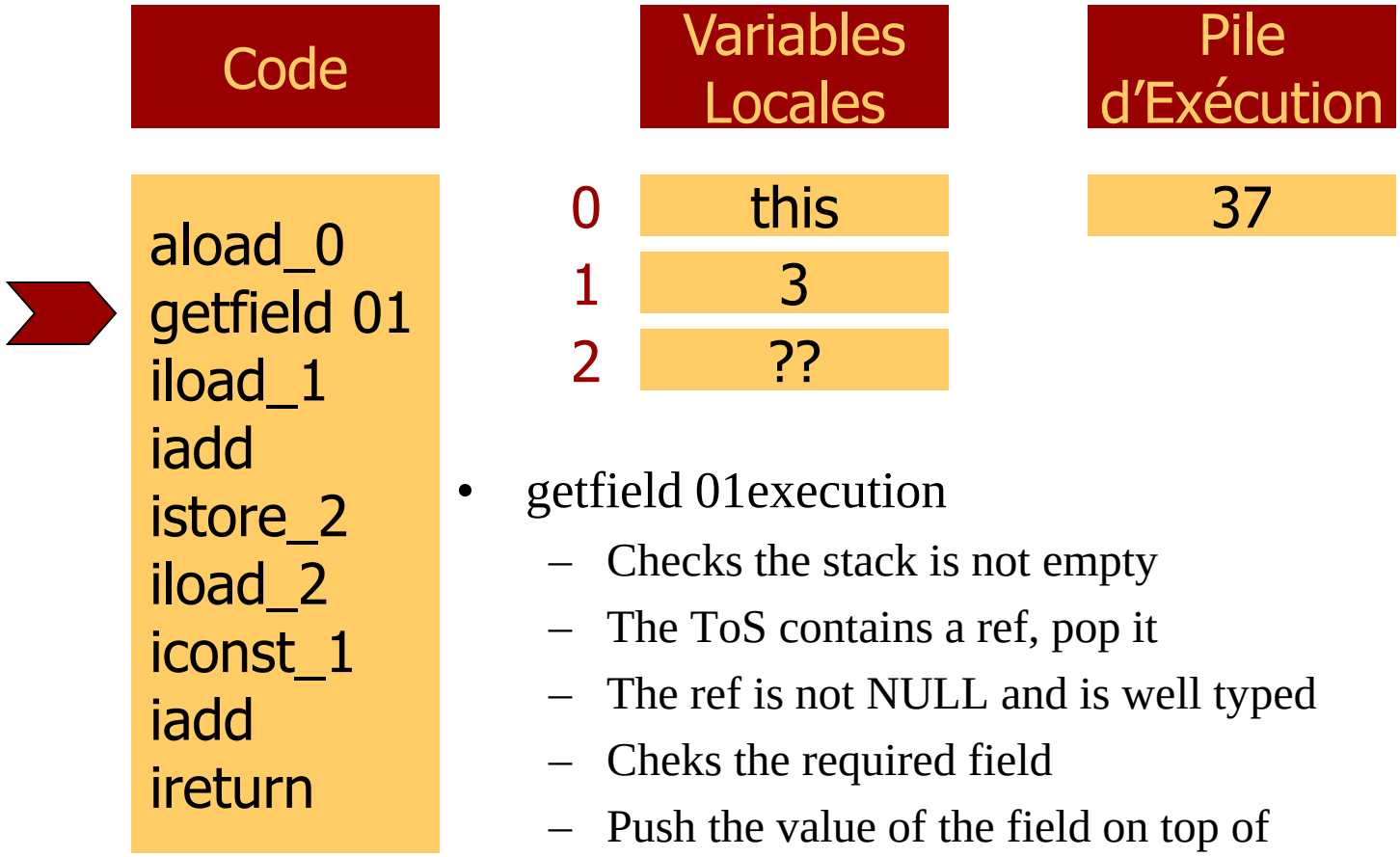
Code
aload_0 getfield 01 iload_1 iadd istore_2 iload_2 iconst_1 iadd ireturn

	Variables Locales
0	this
1	3
2	??

Pile d'Exécution
this

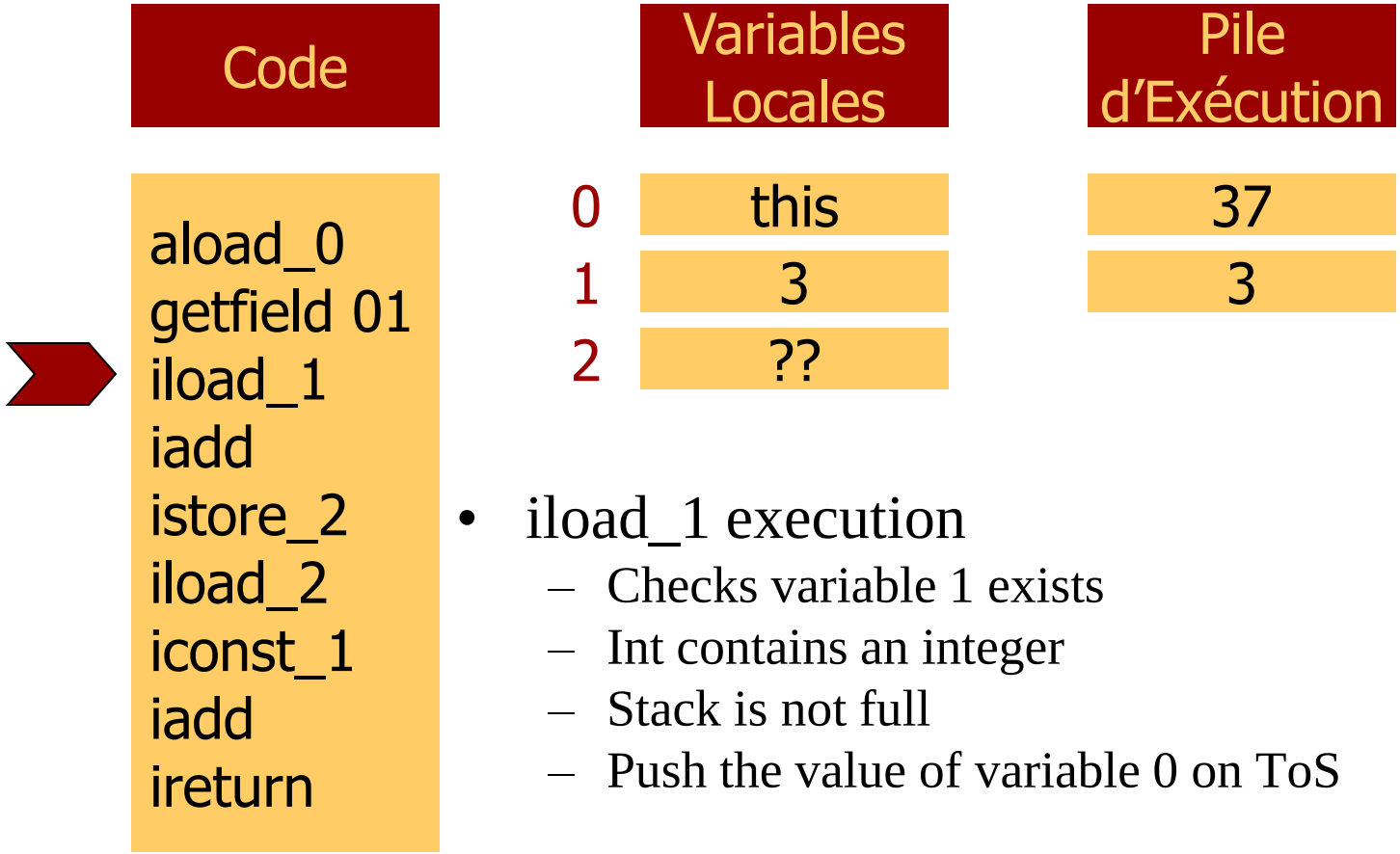
- aload_0 execution
  - Checks variable 0 exist
  - It contains a ref
  - Stack is not full
  - Push the value of variable 0 on top of stack

Execution of byte code



- getfield 01execution
  - Checks the stack is not empty
  - The ToS contains a ref, pop it
  - The ref is not NULL and is well typed
  - Cheks the required field
  - Push the value of the field on top of stack

Execution of byte code



- iload_1 execution
  - Checks variable 1 exists
  - Int contains an integer
  - Stack is not full
  - Push the value of variable 0 on ToS

Such an execution is costly

- Java is a typed language
  - Instruction are typed
  - Execution stack must be typed
  - Local variables must be typed
- Java guarantee the correct frame allocation
  - Local variables must have been allocated
  - Stack under and overflow must be check
- Such verifications have been a huge problem for other stack based languages
  - Java introduced type verification

Useful the verification ?

- Some properties are statically verifiable.
  - No need to check them dynamically
  - Check once during upload
- Verification can be very costly
  - Unfortunately Sun designed the byte code in order to simplify the verification,
  - Rules on byte code are clearly stated

How to ?

- Different kind of properties,
  - The simplest properties are just tests (size of a component etc.),
  - Type verification is more complex and need a specific mechanism
- Verification based on Abstract Interpretation
  - Abstract values replace real value
  - Fix point algorithm replace loop evaluation

Static constraints

- Major constraints are:
  - Branch only to method code;
  - Branch only to an instruction not operand
  - An instruction cannot access a variable for which the index is higher than the declared one
  - All the reference to the constant pool must be well typed
  - End of method must finish with a return
  - Exception handler are bounded by instruction and their end is after the start.

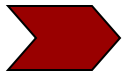
Structural constraints

- Major constraints are:
  - All instruction are executed with the expected arguments on top of stack and in the variables
  - A local variable is not accessed before being assigned a value
  - Size of stack cannot exceed max_stack
  - Stack underflow is not allwed
  - All method invocation are well typed
  - Fields and private methods are correctly accessed
  - All assignments are type-verified
- Plus several minor verifications…

Method verification

- Information are include in the CAP file
  - Signature (parameter type and return parameter)
  - Size max of the stack
  - Number of local variables
- Other MUST be inferred during verification
  - Local variable types
  - Stack type

Abstract execution example



Code
aload_0 getfield 01 iload_1 iadd istore_2 iload_2 iconst_1 iadd ireturn

	Variables Locales
0	ref: C
1	int
2	unknown

Pile d'Exécution
ref: C

- aload_0 execution
  - Variable 0 exists
  - It contains a ref
  - Stack is not full
  - Push the ***TYPE*** of the variable on ToS

Abstract execution example

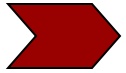
Code
aload_0 getfield 01 iload_1 iadd istore_2 iload_2 iconst_1 iadd ireturn

	Variables Locales
0	ref: C
1	int
2	unknown

Pile d'Exécution
int

- getfield 01execution
  - The stack is not empty
  - ToS contains a ref, pop it
  - Checks the type of the ref
  - Chek the required field
  - Push the ***TYPE*** of the field on ToS

Abstract execution example



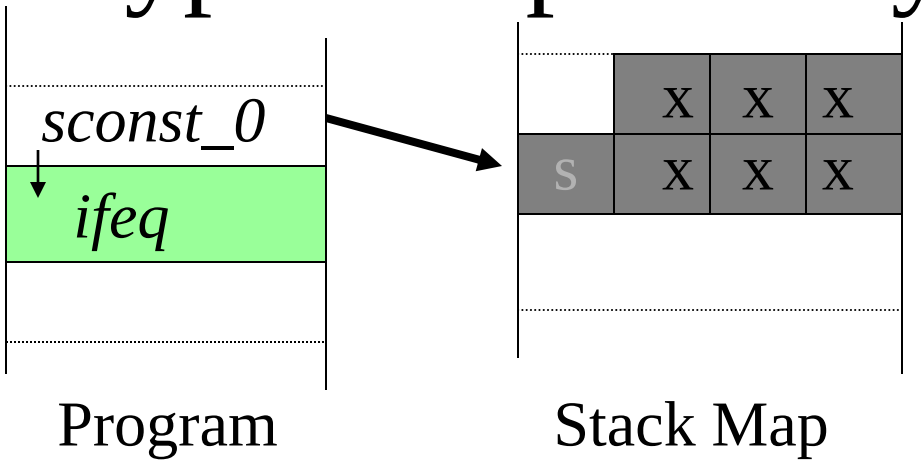
Code
aload_0 getfield 01 iload_1 iadd istore_2 iload_2 iconst_1 iadd ireturn

	Variables Locales
0	ref: C
1	int
2	unknown

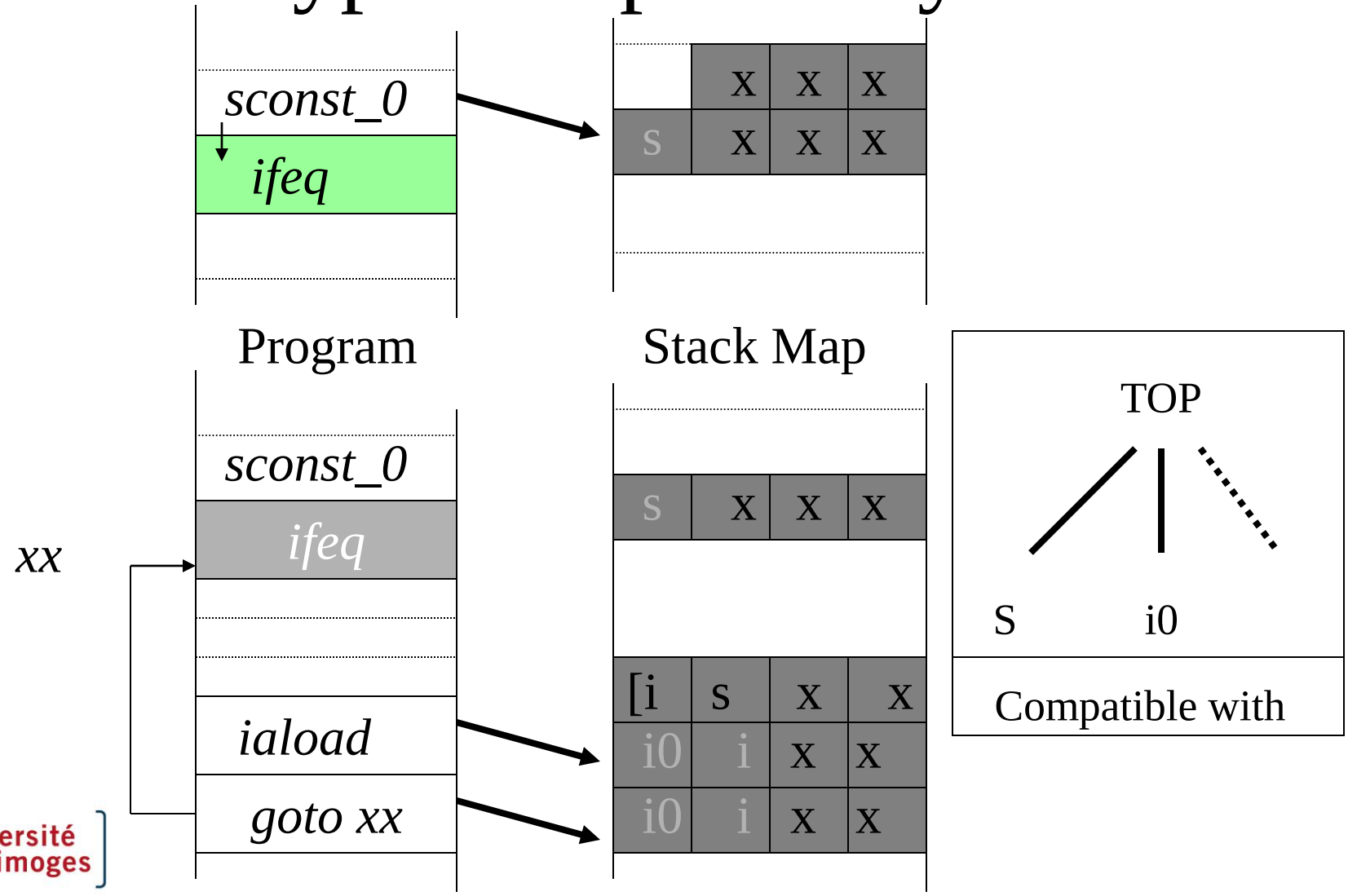
Pile d'Exécution
int
int

- iload_1 execution
  - Check variable 1 exists
  - Its ***TYPE*** is an int
  - Stack is not full
  - Push an ***INT*** on ToS

Type compatibility



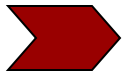
Type compatibility



About type inference and verification

- The method is simple
  - But the example was trivial,
  - Branching add complexity
  - Exception handler add also complexity
  - Subroutines cannot be treated by this scheme (the function is not monotonic)
- Hard to test (really !)
  - Building test cases is very complex
  - One way is to prove the correctness of the code
- Speed up the interpreter execution time
  - Less verification during execution

Execution without verification



Code
aload_0 getfield 01 iload_1 iadd istore_2 iload_2 iconst_1 iadd ireturn

	Variables Locales
0	this
1	3
2	??

Pile d'Exécution
this

- aload_0 execution
  - Push the value of variable 0 on ToS

Execution without verification

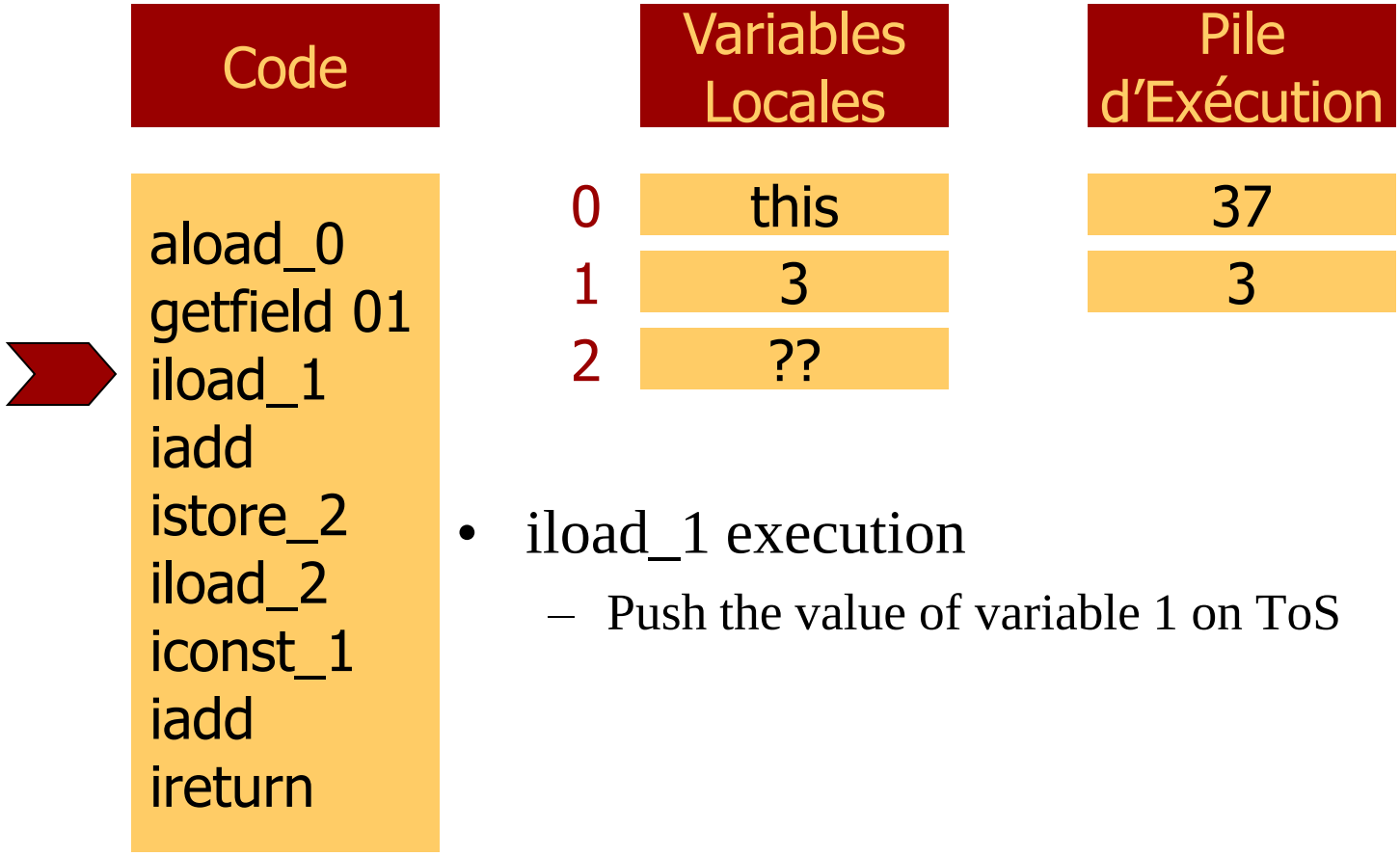
Code
aload_0 getfield 01 iload_1 iadd istore_2 iload_2 iconst_1 iadd ireturn

	Variables Locales
0	this
1	3
2	??

Pile d'Exécution
37

- getfield 01 execution
  - Checks ToS is not NULL
  - Pop the ref on ToS
  - Push the epected field on ToS

Execution without verification



- iload_1 execution
  - Push the value of variable 1 on ToS

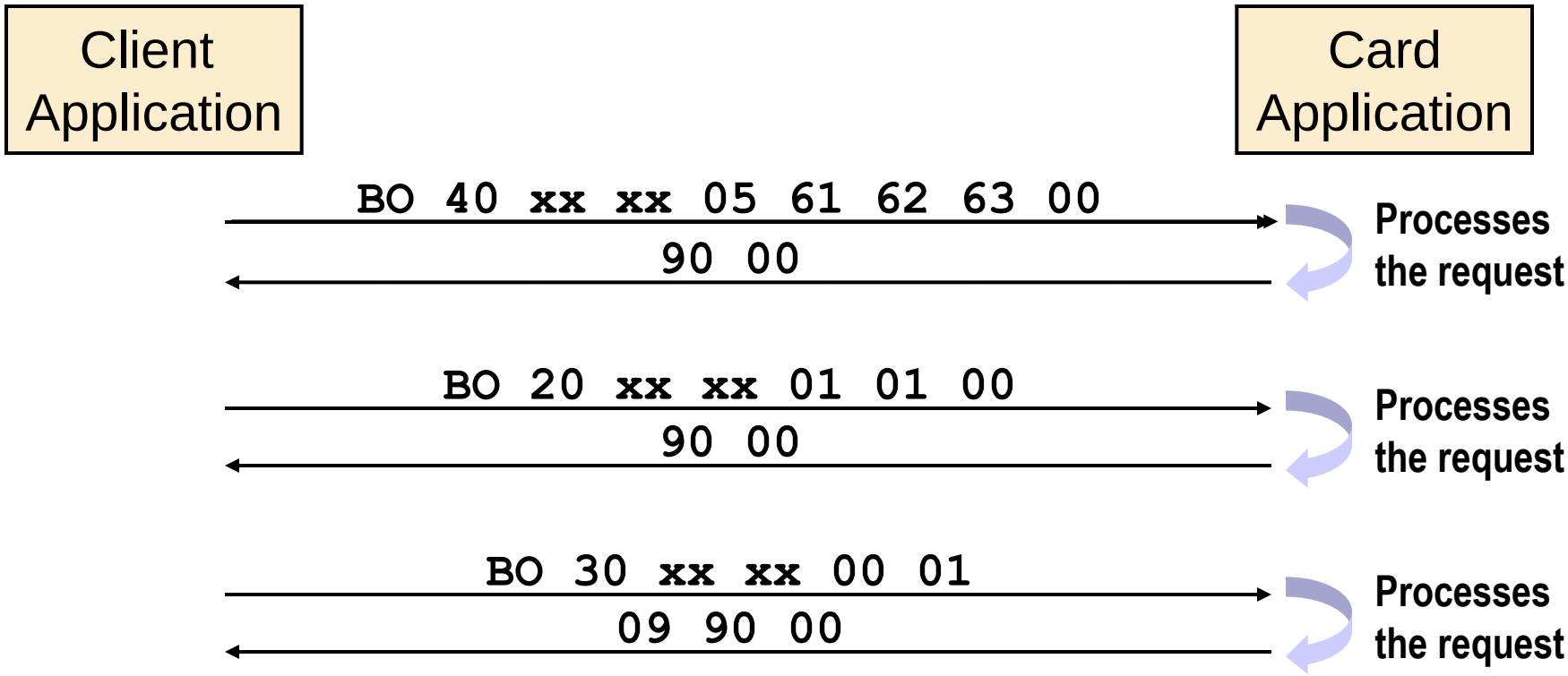
Conclusion on byte code verification

- Byte code verification is a key mechanism of Java
  - The interpreter is designed to run with BC verification
  - BC verification is very efficient
    - Most of the cases linear complexity
    - Worst case quadratic
  - Then interpreter is simple and efficient
    - JIT compiler is also simplified
- BC verification is a key element to Java success
  - It allows to download safely executable code
  - Applet isolation.

Agenda

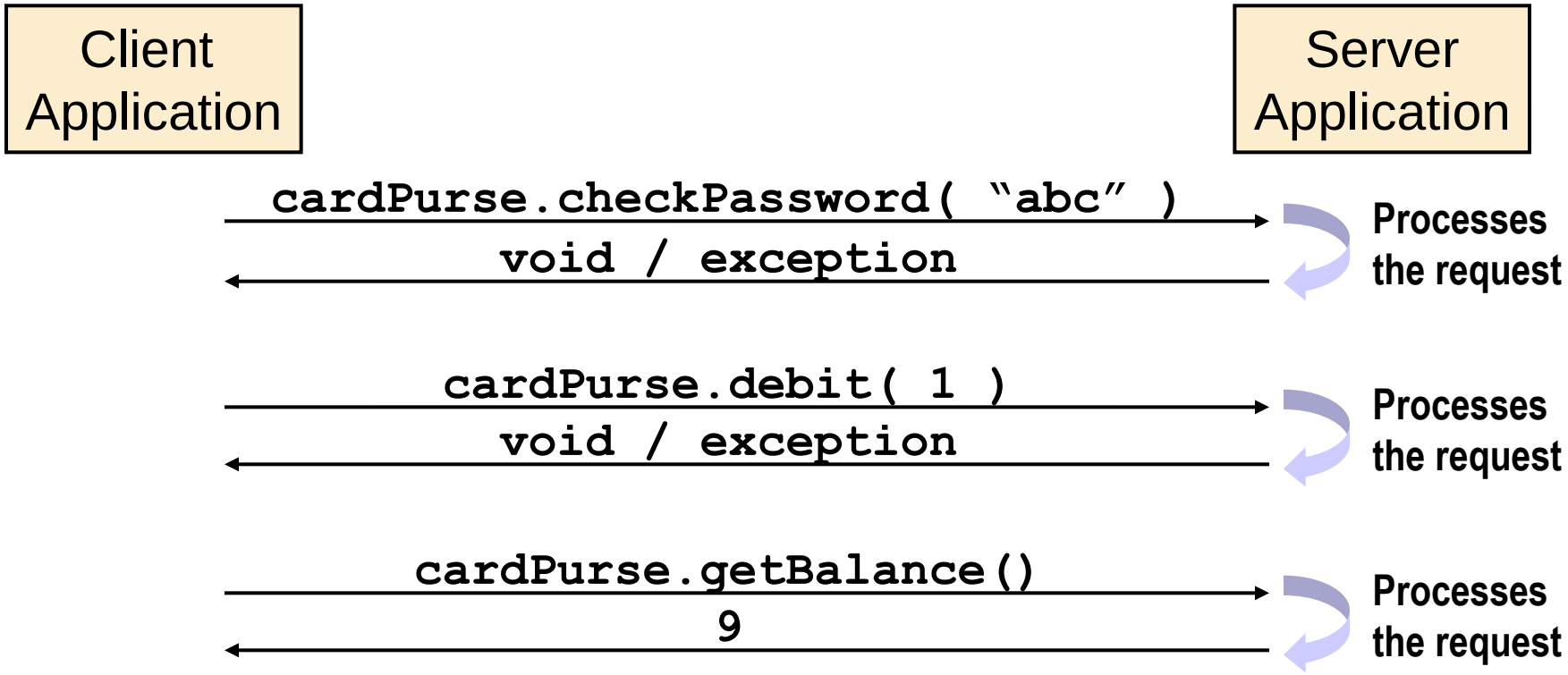
- Sharing mechanism
- Garbage collection
- Logical channel
- Byte code verification
- RMI

Java Card-based applications



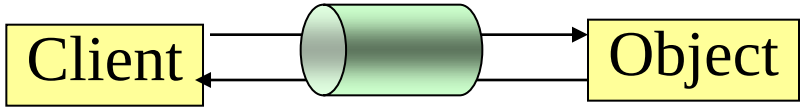
Exchanges are done through APDU messages defined by the legacy smart card protocol

Client/server application

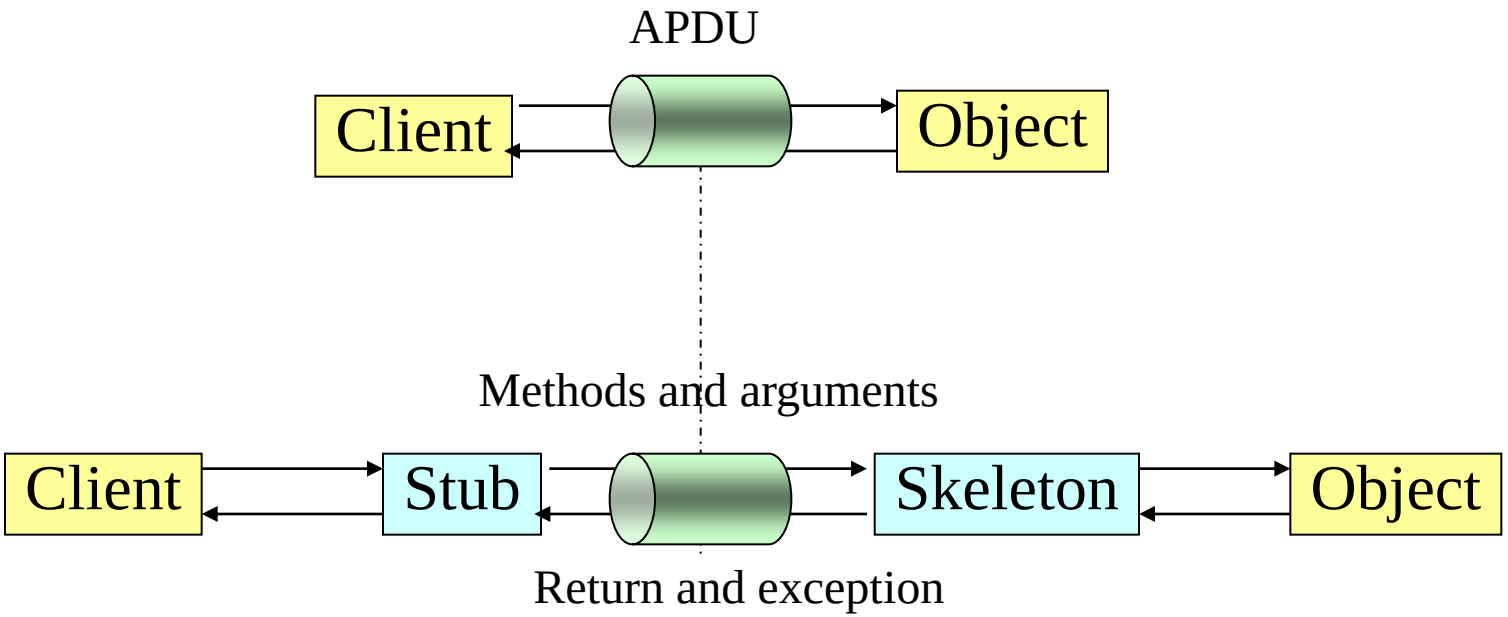


Exchanges are done through method invocations supported by the object abstraction

RMI paradigm



RMI paradigm



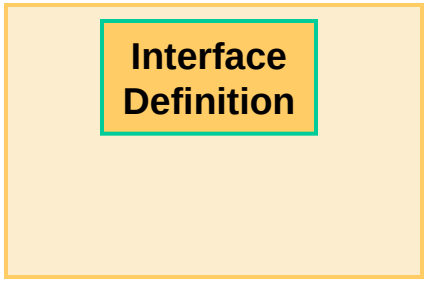
RMI in the Java platform

- Characteristics
  - Client-server application protocol : defines a format for method invocation
  - Uses Java platform interfaces as contracts
  - Supports mobile code
- Advantages
  - Communicates through invocation
  - No message encoding and decoding: independent of low-level protocol
  - Reduces communication errors: reports problems with `RemoteException`

RMI subset

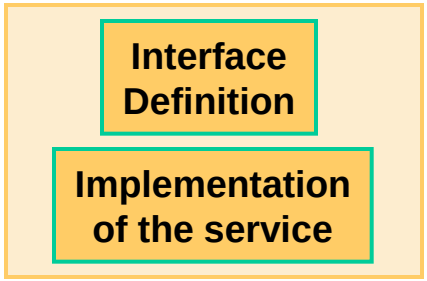
- Remote object features
  - Designed with Remote and `RemoteException`
  - One object reference exported
  - Generated skeleton is a Java Card API-based applet instance
- Remote method features
  - Parameters and return types can be primitive types (boolean, byte, short) and single-dimensional arrays.
  - Java Card API exceptions are reported like in RMI

Five-step development



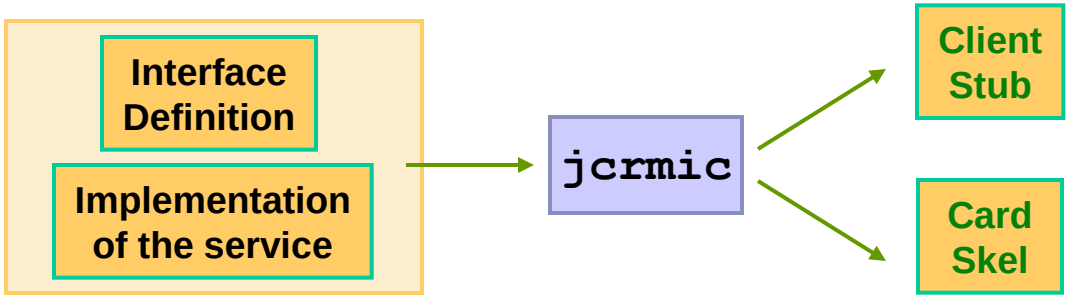
☆Define the remote interface to your card service

Five-step development



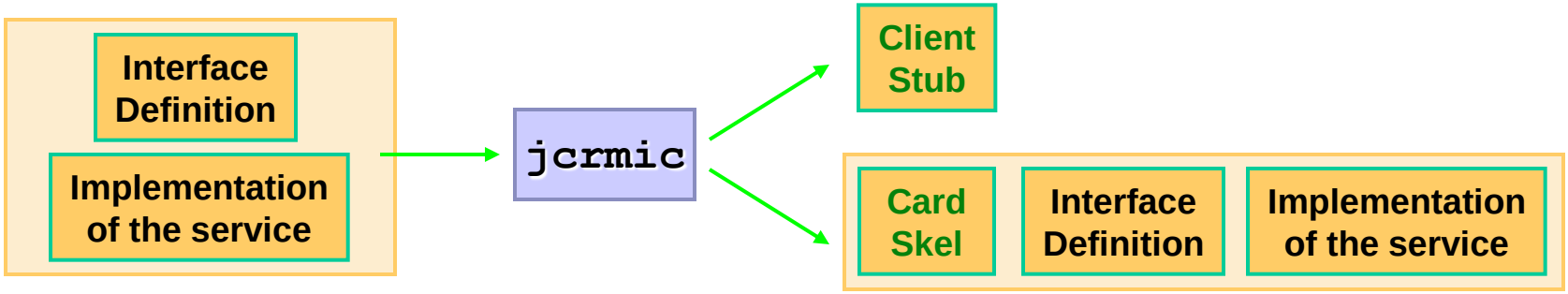
- Implement your card service

Five-step development



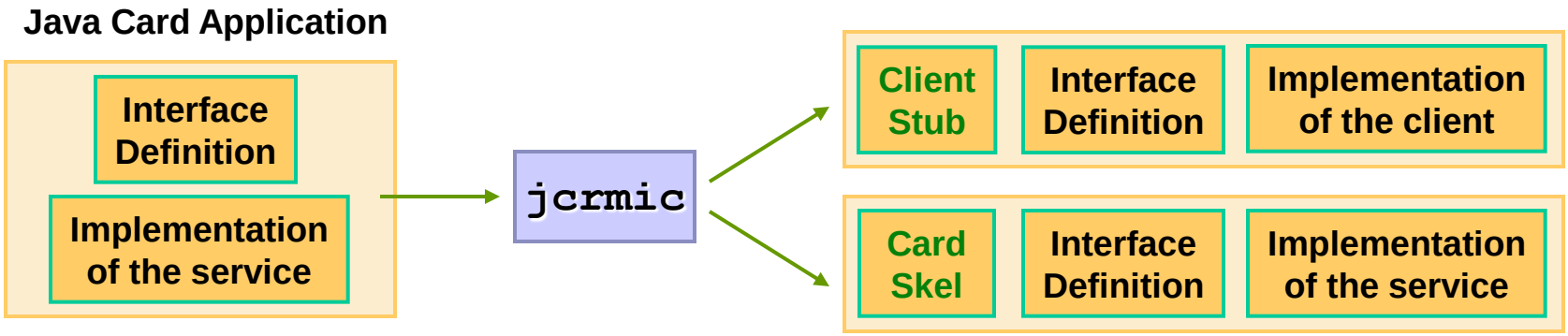
- Run "`jcrmic`" on the implementation classes

Five-step development



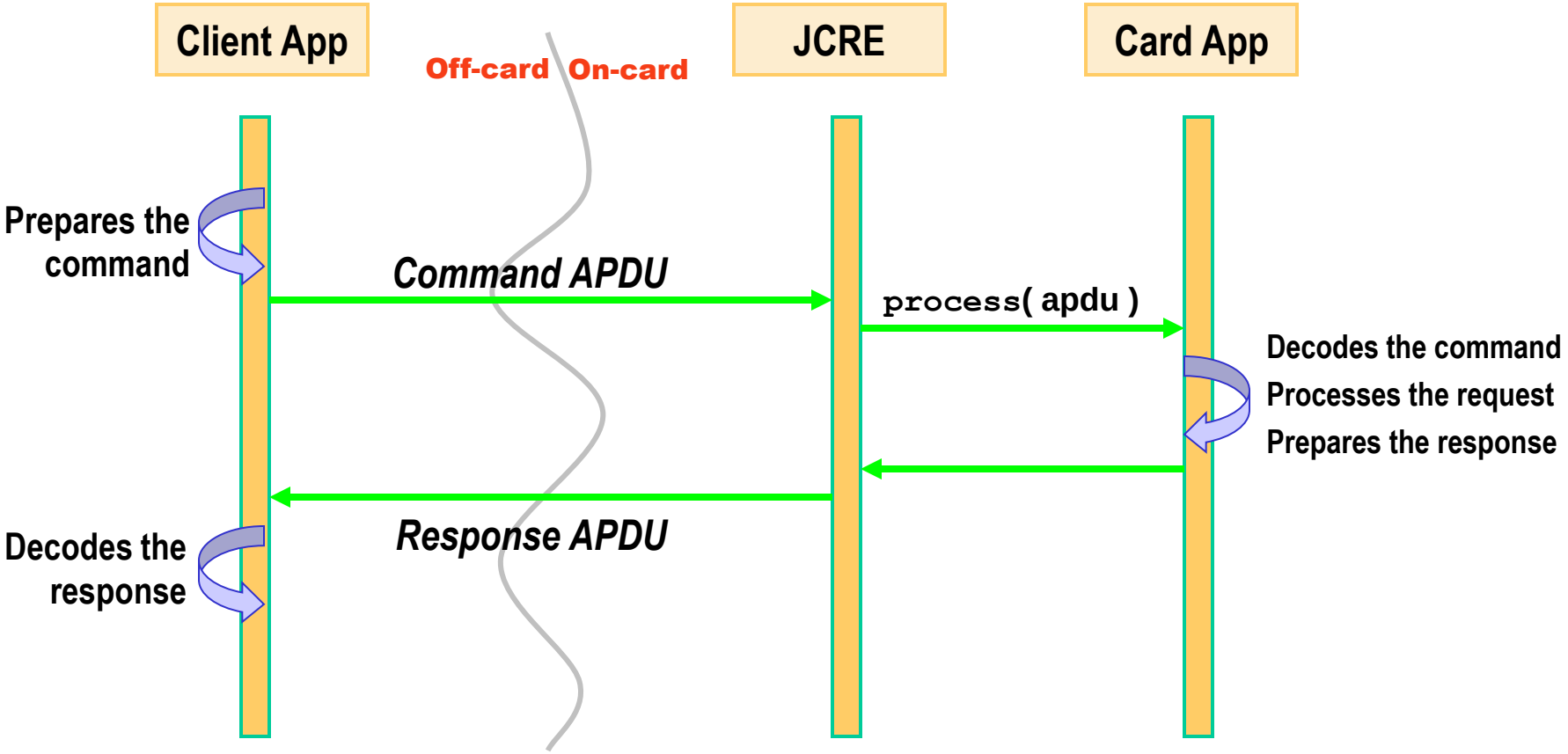
- Install your service with the Card Skeleton in the card

Five-step development

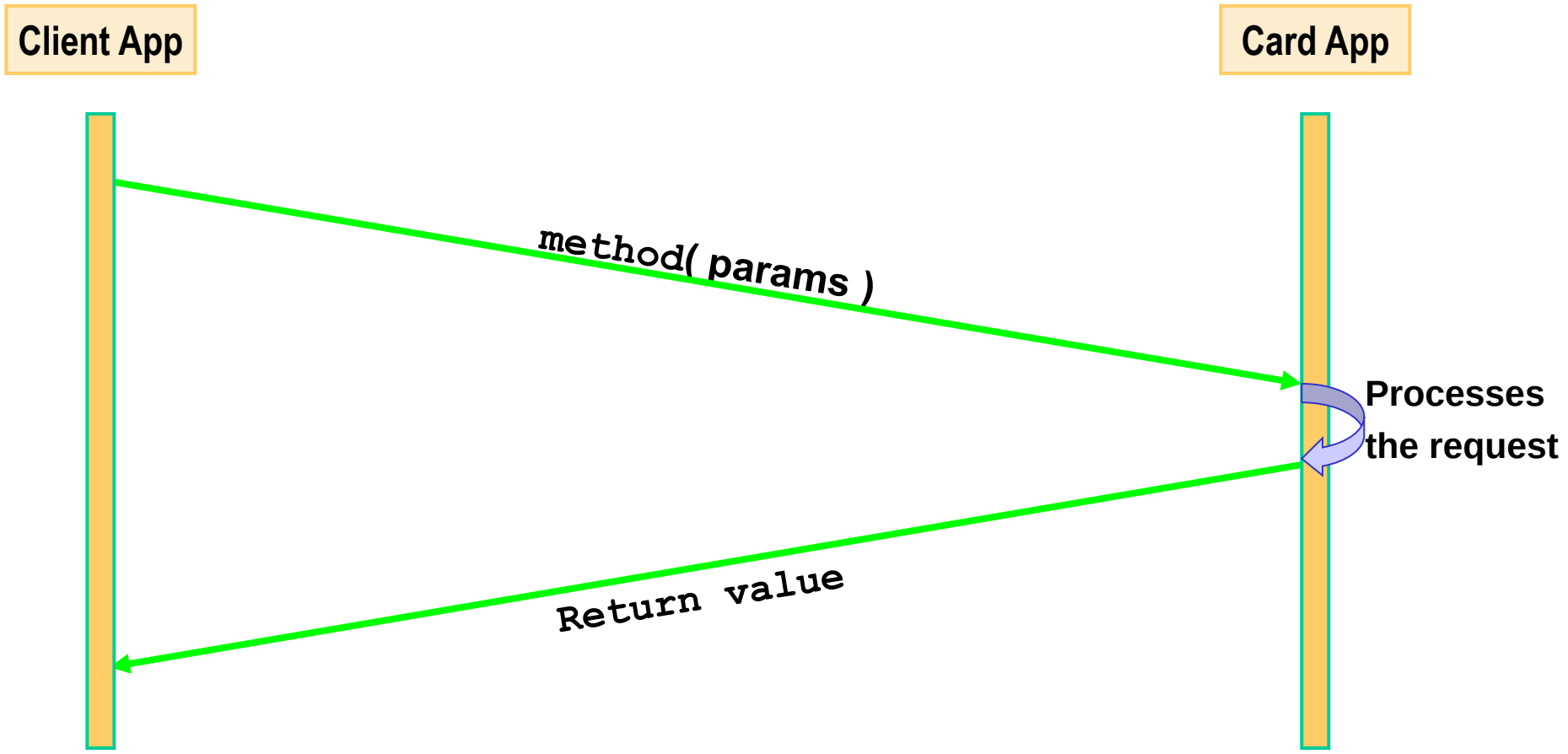


- Use the Client Proxy to invoke card service's methods from your client application

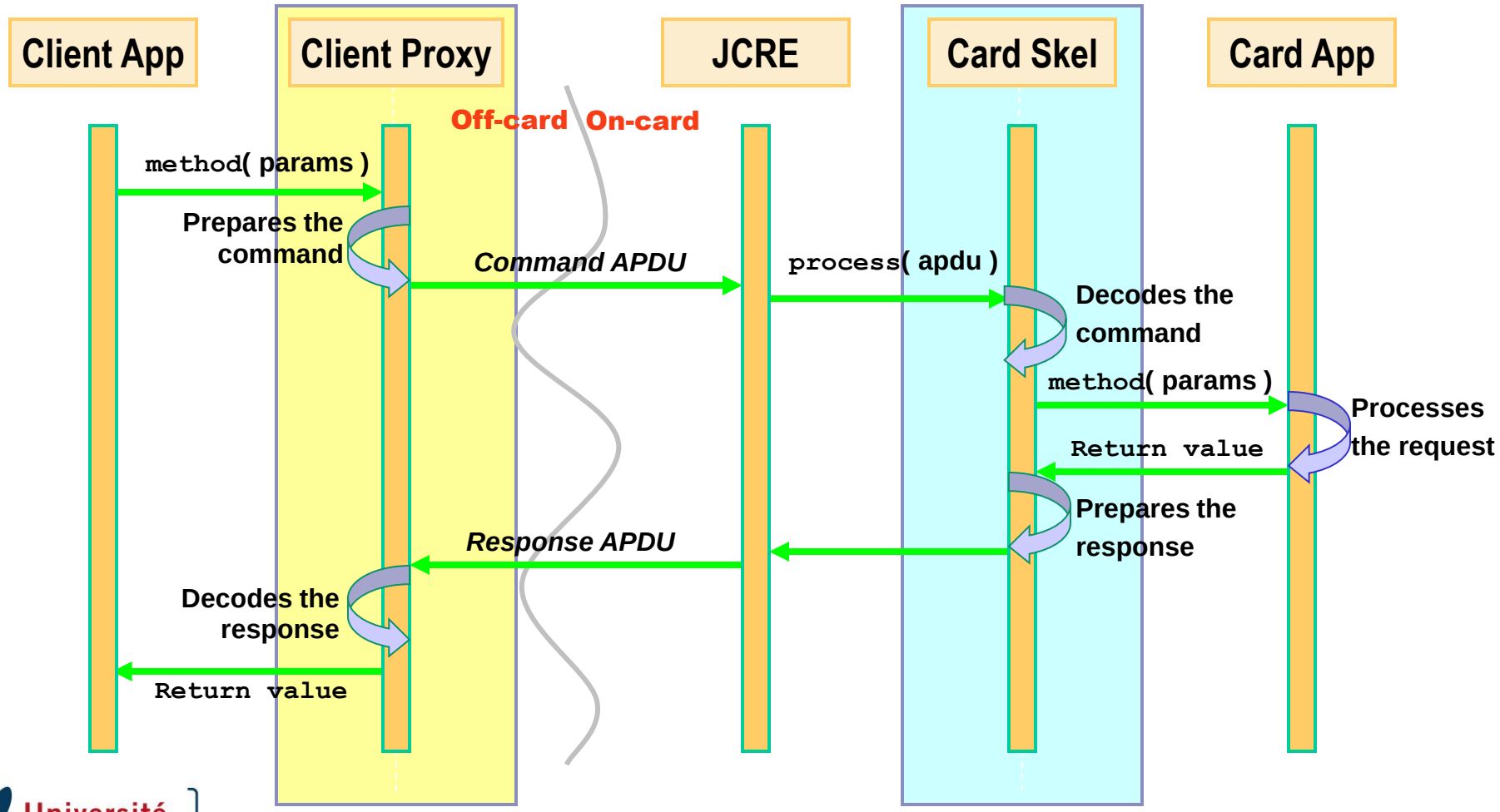
Java Card 2.2 sequence diagram



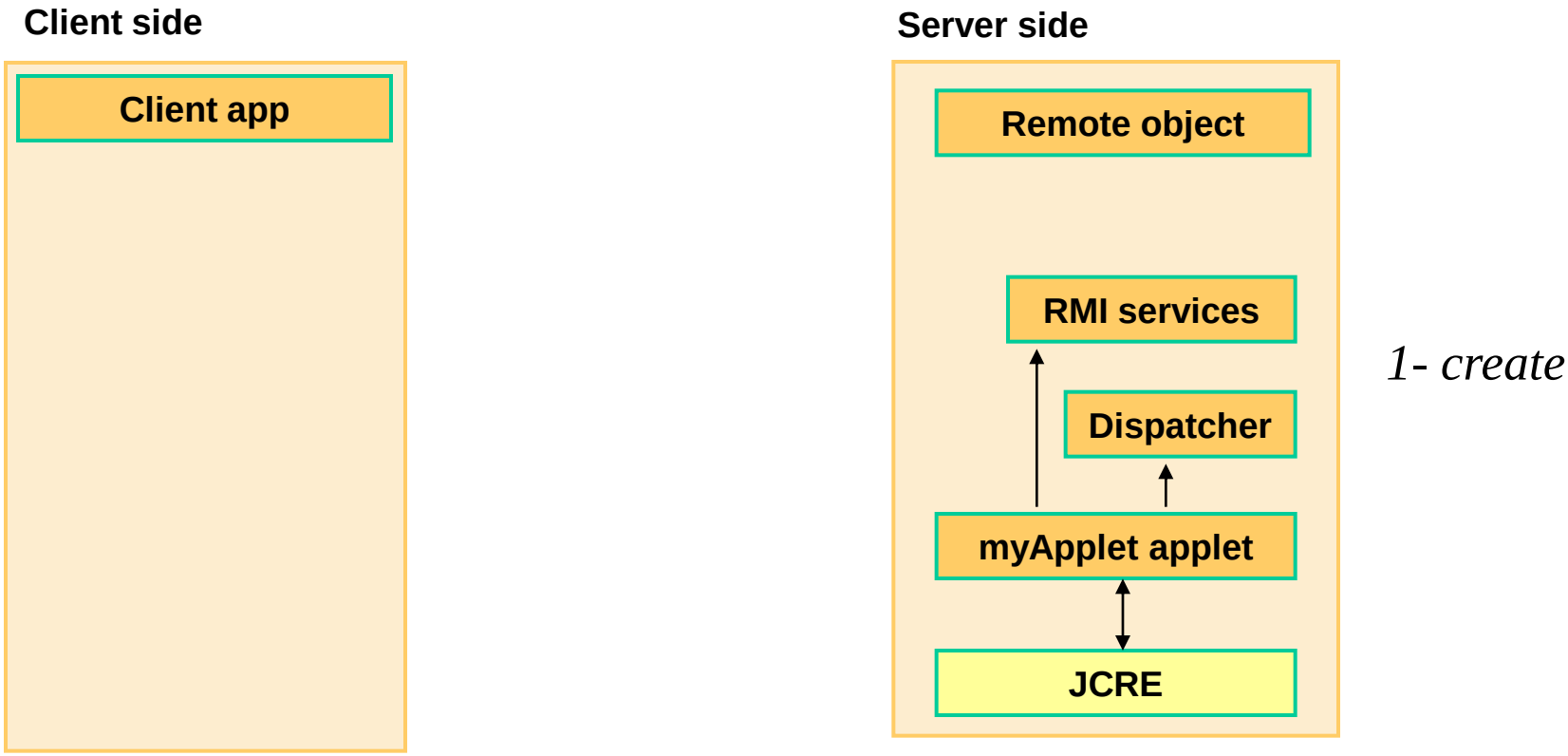
Java Card 2.2 RMI sequence diagram



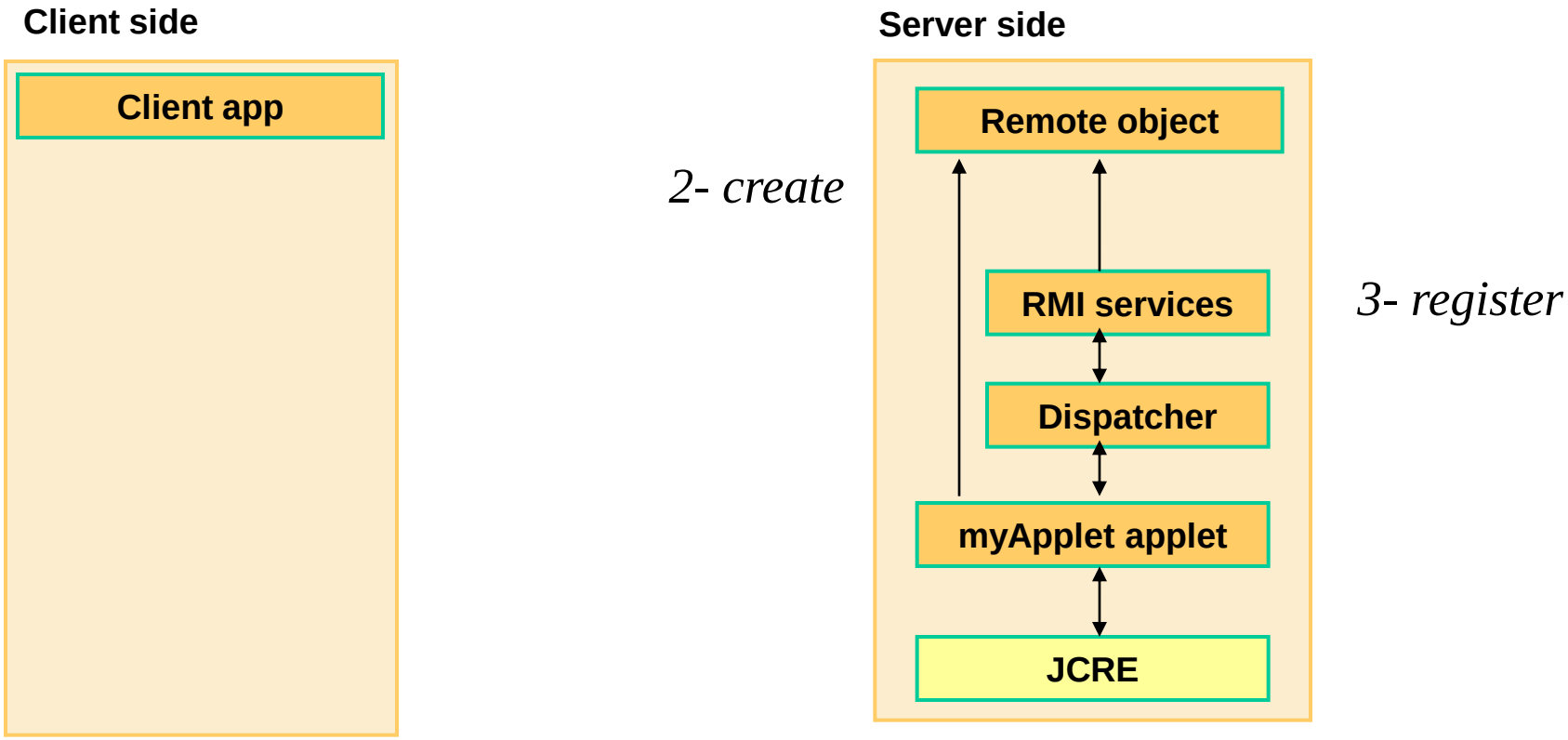
Java Card 2.2 RMI sequence diagram (under the hood)



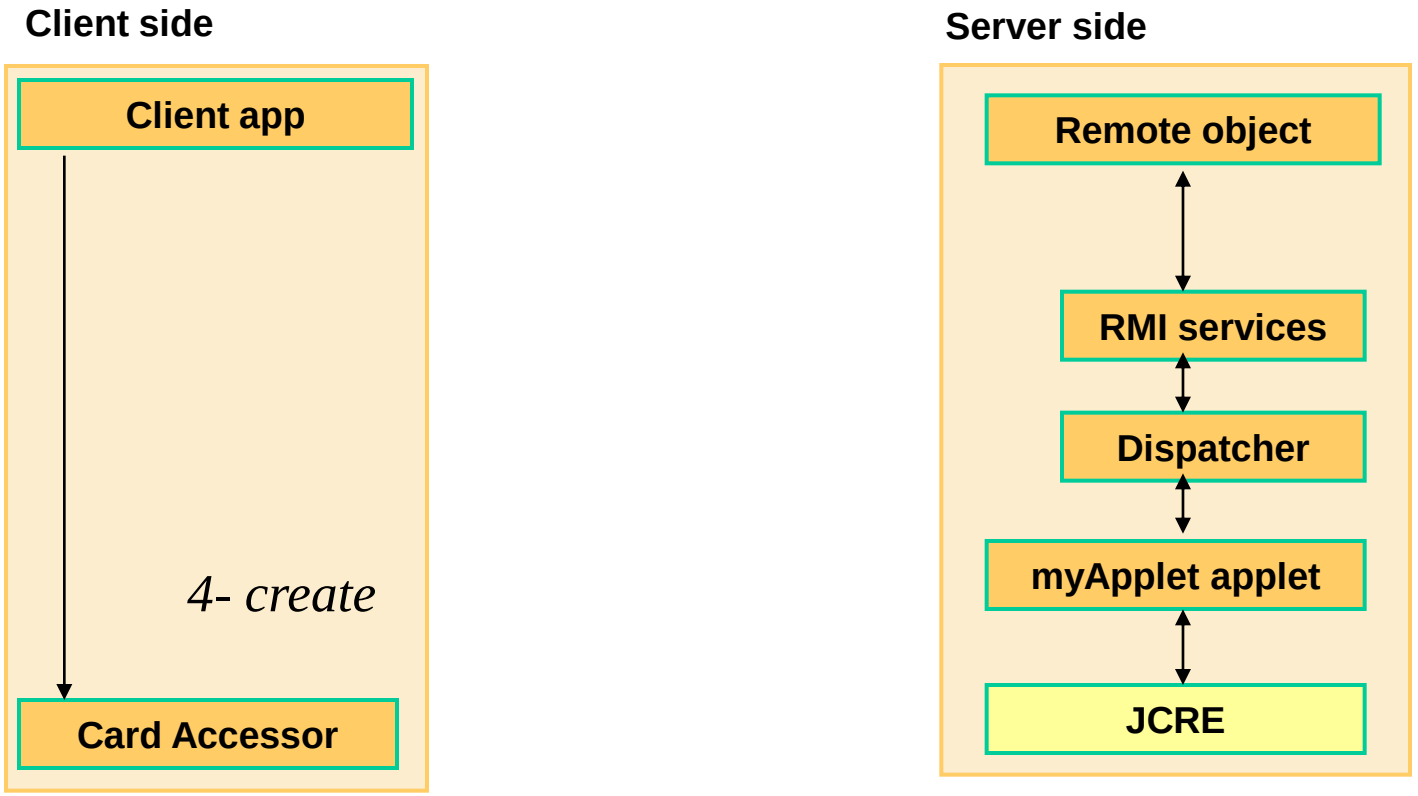
The complete picture



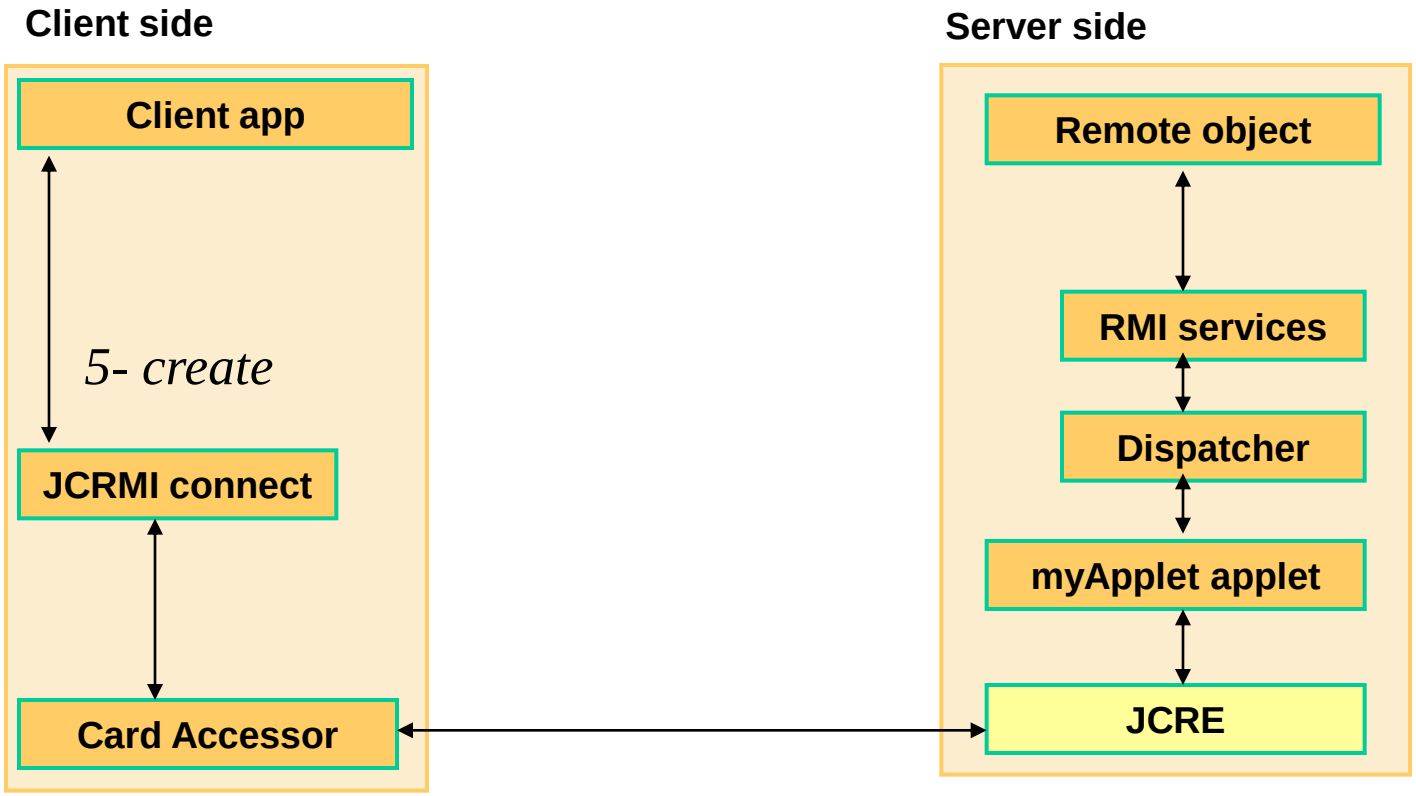
The complete picture



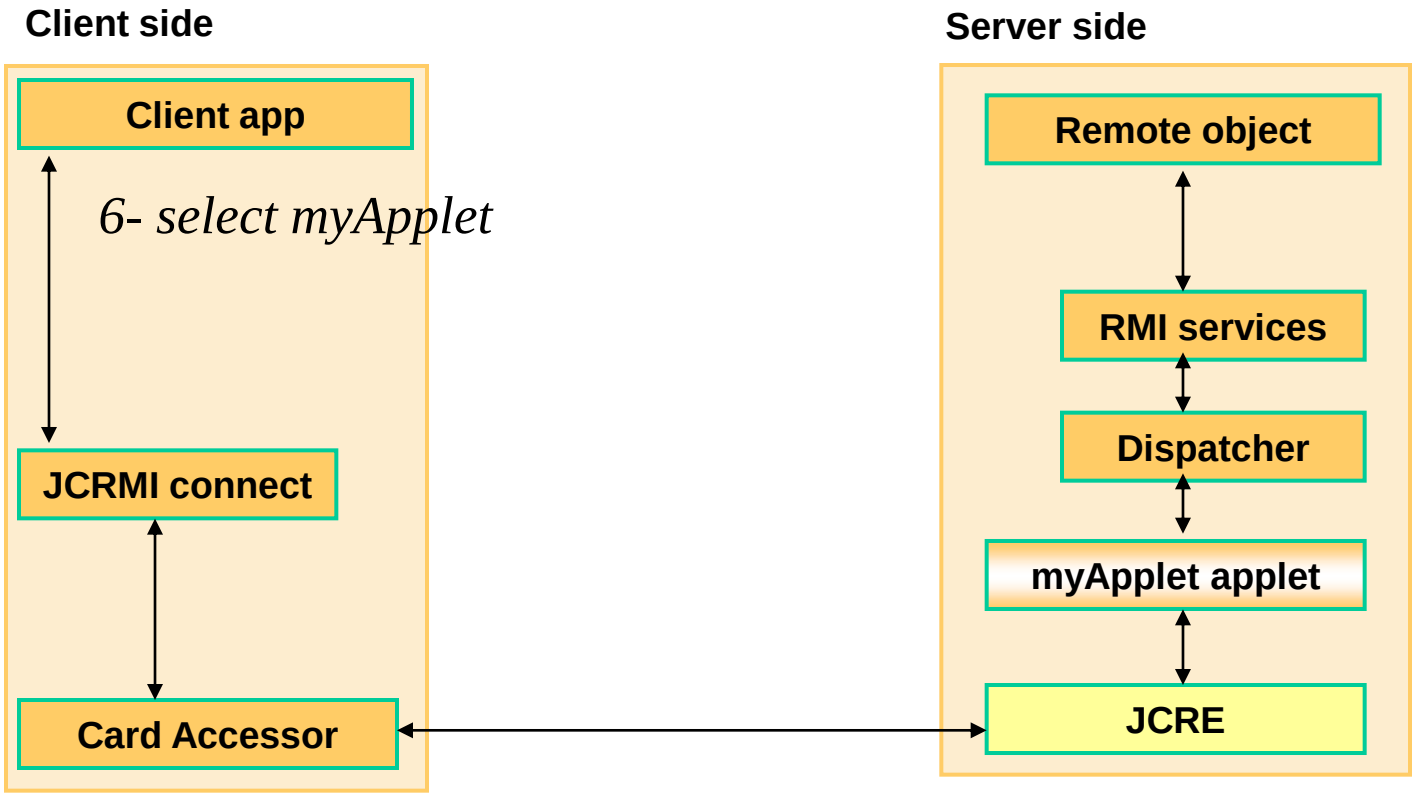
The complete picture



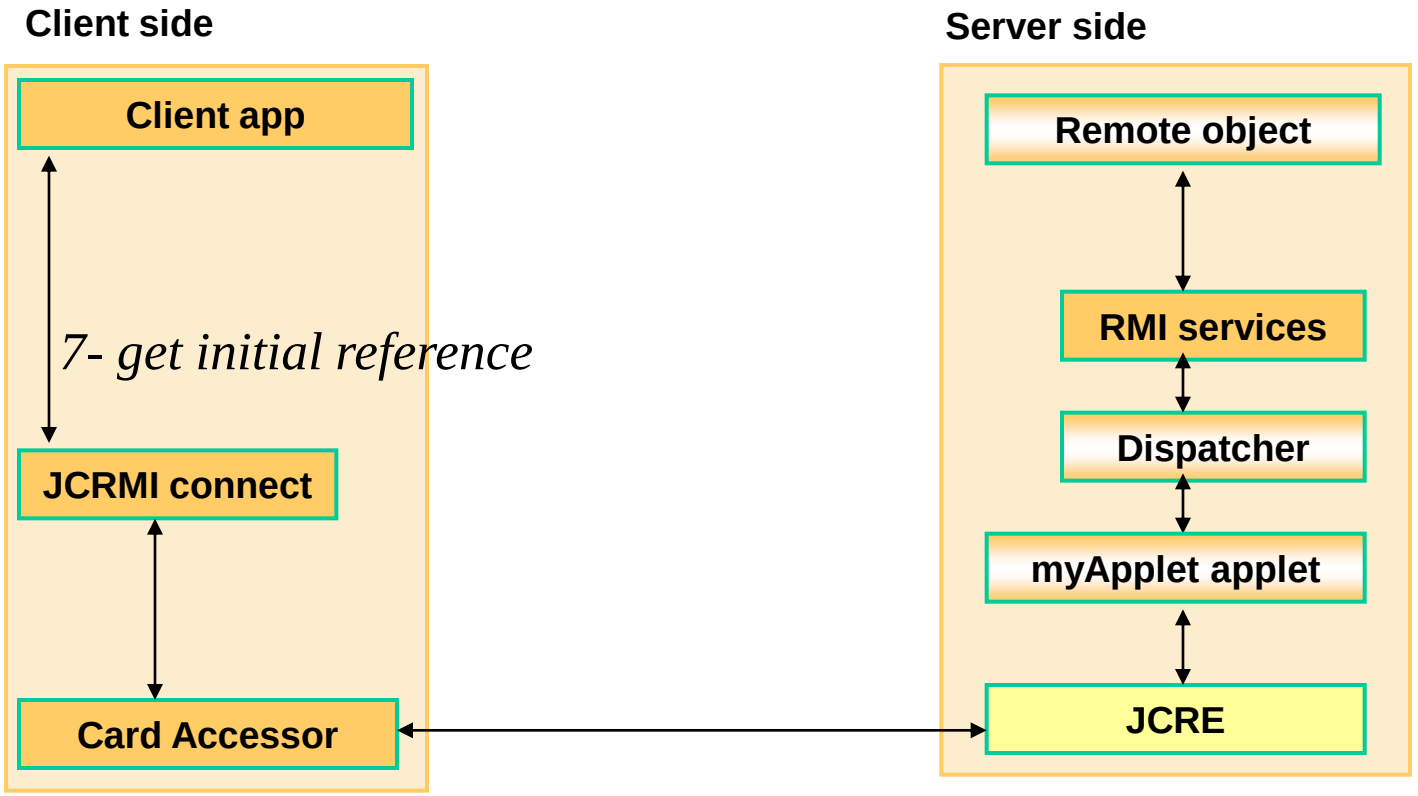
The complete picture



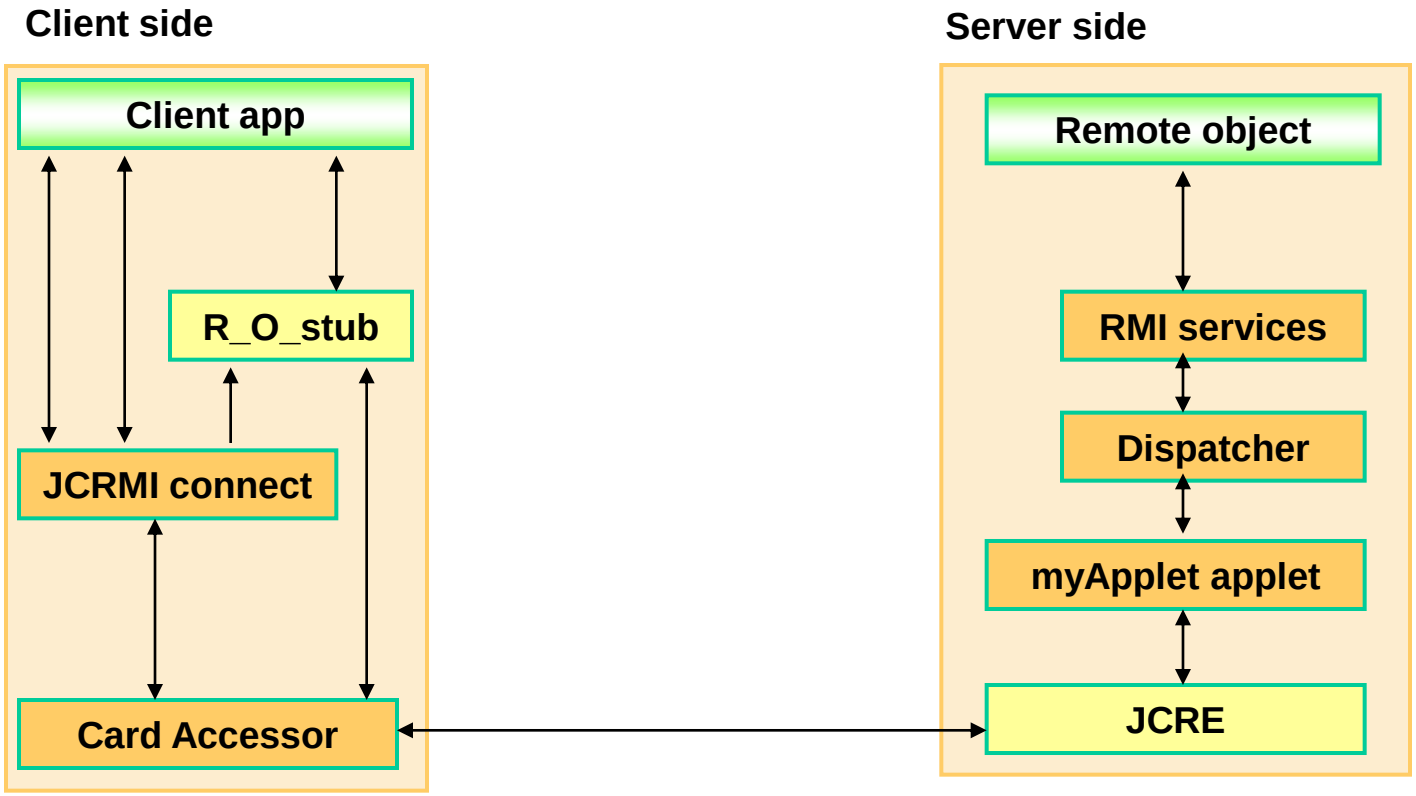
The complete picture



The complete picture



The complete picture



Development with Java Card 2.2 RMI

- Design the Counter interface
  - Design the method signatures
  - Define required constants
  - Add checked exceptions
- Then, implement a class ...

Interface ICounter

```
import java.rmi.Remote ;
import java.rmi.RemoteException ;
import javacard.framework.UserException ;

  public interface ICounter extends Remote
  {
    public static final short NEGATIVE_VALUE = 1;

    public short read()
      throws RemoteException;

    public short increment( short amount )
      throws RemoteException, UserException;

    public short decrement( short amount )
      throws RemoteException, UserException;
  }
```

Interface ICounter

```
import java.rmi.Remote ;
import java.rmi.RemoteException ;
import javacard.framework.UserException ;

  public interface ICounter extends Remote
  {
    public static final short NEGATIVE_VALUE = 1;

    public short read()
      throws RemoteException;

    public short increment( short amount )
      throws RemoteException, UserException;

    public short decrement( short amount )
      throws RemoteException, UserException;
  }
```

Implementation class Counter

```
public class Counter extends CardRemoteObject
implements Icounter {
private short value;

public Counter() { value = 0; }

public short read() throws RemoteException {
  return value;
}

public short decrement(short amount)
  throws RemoteException, UserException {
  if (amount<0 || value-amount<0)
    UserException.throwIt(NEGATIVE_VALUE);
  value -= amount;
  return value;
}
}
```

Implementation class Counter

```
public class Counter extends CardRemoteObject
  implements Icounter {
  private short value;

  public Counter() { value = 0; }

  public short read() throws RemoteException {
    return value;
  }

  public short decrement(short amount)
    throws RemoteException, UserException {
    if (amount<0 || value-amount<0)
      UserException.throwIt(NEGATIVE_VALUE);
    value -= amount;
    return value;
  }
}
```

Implementation class Counter

```
public class Counter extends CardRemoteObject
  implements Icounter {
  private short value;

  public Counter() { value = 0; }

  public short read() throws RemoteException {
    return value;
  }

  public short decrement(short amount)
    throws RemoteException, UserException {
    if (amount<0 || value-amount<0)
      UserException.throwIt(NEGATIVE_VALUE);
    value -= amount;
    return value;
  }
}
```

Implementation class Counter

```
public class Counter extends CardRemoteObject
   implements Icounter {
   private short value;

   public Counter() { value = 0; }

   public short read() throws RemoteException {
      return value;
   }

   public short decrement(short amount)
      throws RemoteException, UserException {
      if (amount<0 || value-amount<0)
        UserException.throwIt(NEGATIVE_VALUE);
      value -= amount;
      return value;
   }
}
```

myApplet Applet

```
import javacard.framework.service.RemoteService;
import javacard.framework.service.RMIService;
import javacard.framework.APDU;

public class myApplet extends Applet {
   private Dispatcher dispatcher;
   private RemoteService remoteService;
   private ICounter iCounter;
…
}
```

myApplet Applet

```
   …
public myApplet() {
   disp = new Dispatcher((short)1);
   iCount = new ICount();
   remoteService = new RMIService(iCounter);
   disp.addService(remoteService,Dispatcher.PROCESS_COMMAND);
}
public static void install(byte[] aid,short s,byte b) {
   myApplet server = new myApplet();
   server.register();
}
public void process(APDU apdu) throws javacard.framework.
   ISOException {
disp.process(apdu);  }
```

Client Slide (1/3)

```
import opencard.core.service.*;
import com.sun.javacard.javax.smartcard.rmisclient.*;
import com.sun.javacard.ocfrmiclientimpl.*;
import javacard.framework.UserException;
public class CounterClient extends java.lang.Object{
  public CounterClient()
  public static void main (java.lang.String[] argv)
  // arg[0] contains the debit amount
  try {
   // initialize OCF
   SmartCard.start();
   // wait for a smart card
   CardRequest cr = new CardRequest
   (CArdRequest.NEWCARD,null,OCFAccessor.class);
```

Client Slide (1/3)

```
import opencard.core.service.*;
import com.sun.javacard.javax.smartcard.rmisclient.*;
import com.sun.javacard.ocfrmiclientimpl.*;
import javacard.framework.UserException;
public class CounterClient extends java.lang.Object{
  public CounterClient()
  public static void main (java.lang.String[] argv)
  // arg[0] contains the debit amount
  try {
   // initialize OCF
   SmartCard.start();
   // wait for a smart card
   CardRequest cr = new CardRequest
   (CArdRequest.NEWCARD, null, OCFAccessor.class);
   SmartCard myCard SmardCard.waitForCard (cr)
```

Client Slide (2/3)

```
try { …
// 1) Obtain an RMI Card Accessor  CardService for the JCRE
CardAccessor myCS = (CardAccessor) myCard.getCardService
   (OCFCardAccessor.class, true)
// 2) Create an RMI connector instance
JavacardRMIConnect jcRMI = nex JavaCardRMIConnect (myCS)
// 3) Select the applet
byte[] aidCounter new byte[] {0x57,…}
jcRMI.selectApplet (aidCounter);
// 4) obtain the initial reference on the counter interface
Counter myCounter = (Counter) jcRMI.getInitialReference();
```

Client Slide (3/3)

```
try { …
// Obtain an RMI …
// obtain the initial reference … myCounter
try {
  short balance = myCounter.decrement (amount);
  } catch (UserException jce) {…}
System.out.println (balance);

Bla, bla, bla
```

Development with Java Card 2.2 RMI

- Prepare the Card application
  - Generate the skeleton from the interface
  - Convert in a card execute-in-place format the card applet classes
  - Upload the converted card applet classes to the card
  - Install the card applet on-card
- Prepare the client application
  - Generate the client proxy from the interface
  - Write a client application using the proxy
  - Install the client application on card terminal hosts

Introduction

- The Global Platform provides specifications to define security policies and cryptographic mechanisms to protect download and delete of applications.
- Each applet can be securely loaded and removed using either Public Key or Symmetric key cryptography.
- Need to embed the Card Manager as an applet or as native code
- Need to implement the GP API
  - Services: Cardholder verification, personalization, security services,…
  - Card Content management services : card locking, Application life cycle state update,…

GP Card Domain

- Issuer representative
- Provides card global services:
  - installation of applets on the card
  - management of the applet life cycle
  - personalization and reading card global data (such ICC serial number)
  - management of the card life cycle
  - blocking card service
  - auditing services when the card is blocked
- Acts as the security domain for the issuer's applets

Introduction GP 2.2

- Re-engineering of Global Platform Card Framework
  - Architectural extensions with Privileges and Security Domain hierarchies support additional business models
  - New Global Services, i.e. on-card client-server support
  - Improved logical channel support
- Backward compatibility
- Enhancement for contactless interfaces
- Secure Channel Protocol based on Public Key Infrastructure SCP 10
- Card Specification 2.2 Overview Generic card architecture applicable to both GP 2.1.1 (& 2.0.1) and new 2.2 PK features
- Card Remote Application Management over HTTP - Card Specification v 2.2
- To be published SCP03 based on AES.

GP functionalities

- Applet & life cycle management,
  - Need authentication and integrity for :
    - Load, Install and Make Selectable
    - Delete,
    - Set-Get Status.
- Secure communication protocol,
  - Entity authentication,
    - Current Security level = `Authenticated` (SCP01,SCP02) `Any_Authenticated` (SCP10)
  - Confidentiality and/or Integrity and authentication,
    - Integrity or Confidentiality and Integrity of a command sent to the card,
    - Integrity of the sequence of APDU command sent to the card

SCP 01

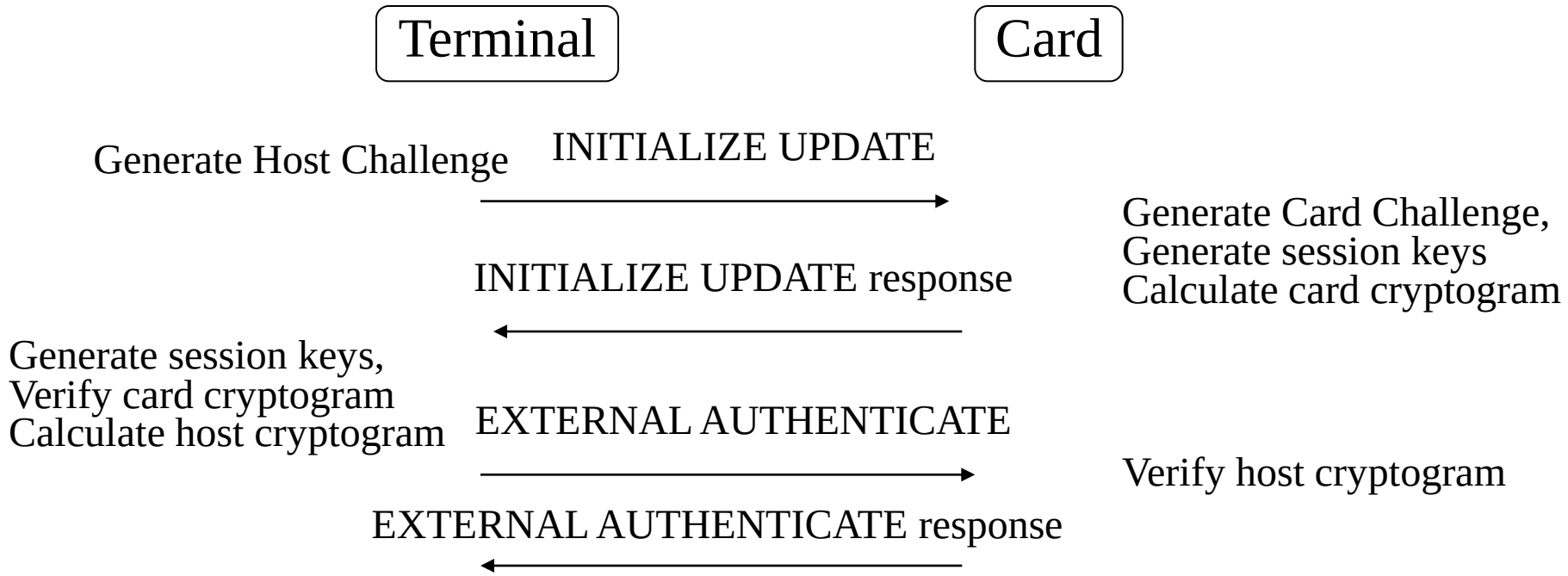
- This protocol modifies APDUs, using some pre-established symmetric keys on both sides, to secure the original APDUs with MAC checks and optional encryption.
- Symmetric key, SCP01 is deprecated, backward compatibility with GP 2.0.1
  - Replaced by SCP 02 symmetric key protocol,
  - Three levels of security
    - Mutual authentication
    - Integrity and data origin authentication
    - **Confidentiality** : **in which data being transmitted from the off-card entity to the card, is not viewable by an unauthorized entity**

SCP01 versus SCP02

- SCP02:
  - **Confidentiality : in which data being transmitted from the sending entity (the off-card entity or card) to the receiving entity (respectively the card or off-card entity) is not viewable by an unauthorized entity**.
- For SCP01, data from host to card is not susceptible to sniffing but no mention of the reverse to be true. For SCP02, both directions are not susceptible to sniffing.
- SCP01 supports mutual auth while for SCP02, only the card authenticates the host, with an option for the reverse.
- There is no encryption from the card side. Be aware that R-MAC is optional, depending on the security policy of the issuer.
- Another differences between SCP01 and SCP02:
  - The DEK in SCP02 is a session key, and in SCP01 it is static,
  - The INITIALIZE UPDATE command is different regarding the P2 parameter and the structure of the response

Mutual Authentication

- It provides assurance to the card and the terminal they are communicating with authenticated entity.



Initialize Update

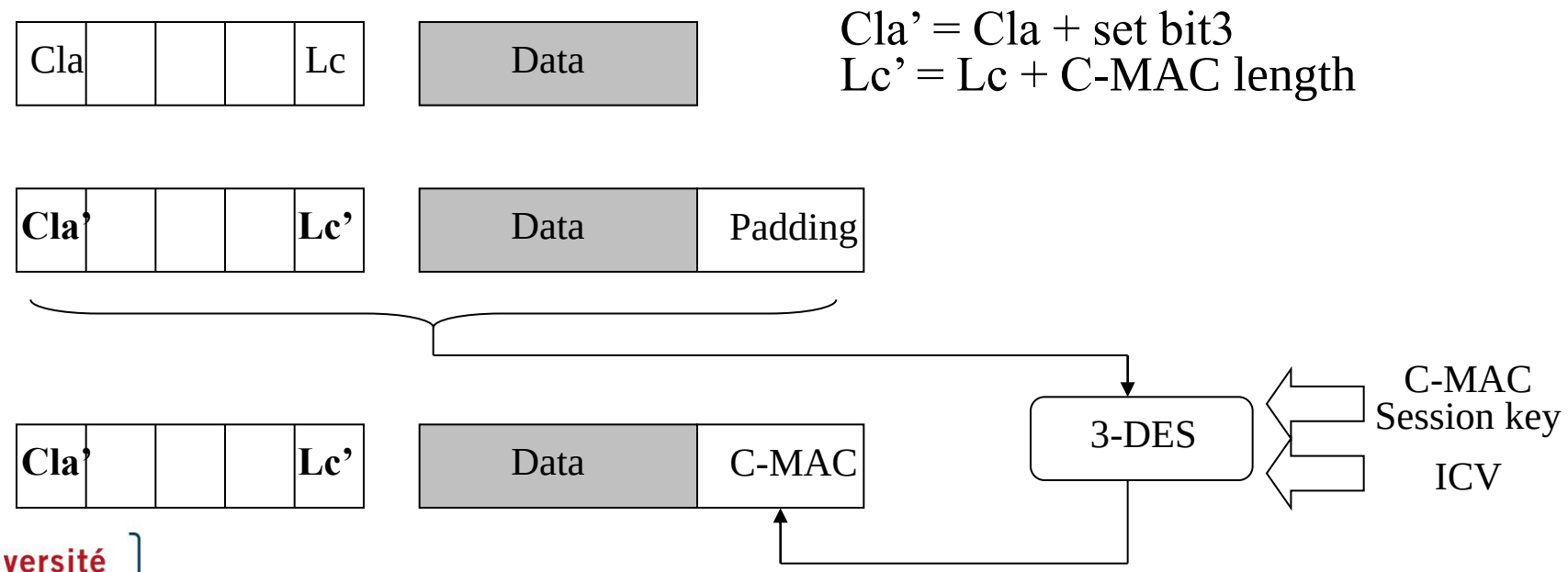
- APDU INIT_UPDATE
  - P1 key version number
  - P2 key set index
  - Data : host challenge
- RESPONSE
  - Card cryptogram + Card Challenge
  - Or 0x6A88 Referenced data not found

External Authenticate

- APDU EXTERNAL_AUTHENTICATE
  - P1 Security Level
    - 0x00 No Secure messaging
    - 0x01 C-MAC
    - 0x03 C-DECRYPTION and C-MAC
  - Response:
    - DATA Host Cryptogram and MAC
    - Or 0x6300 Authentication failed

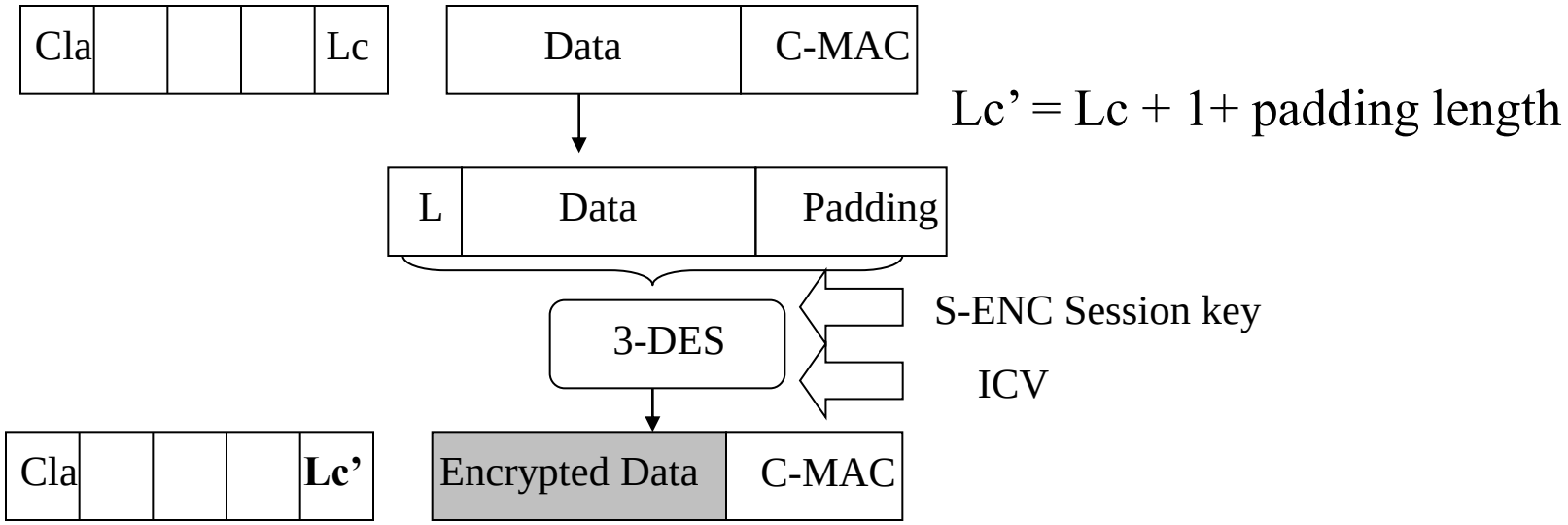
APDU Command MAC generation

- A C-MAC is generated by an off-card entity and applied across the full APDU command being transmitted to the card including the header (5 bytes) and the data field in the command message



APDU data field encryption

- If confidentiality is required, the off-card entity encrypts the “clear text” data field of the command message being transmitted to the card.



The Cryptographic Keys

- S-ENC, Secure Channel Encryption Key
  - A static key to generate a session key: 16 bytes
  - Used to Authentication and encryption (DES)
- S-MAC, Secure Channel Message Authentication Code Key,
  - A static key used to generate a session key: 16 bytes,
  - Used to MAC verification (DES),
- DEK, Data Encryption Key
  - Used as a static key, 16 bytes,
  - For decrypting sensitive data (e.g. secret key)

SCP 01 drawbacks

- The main drawback of SCP01 is the lack of protection of the card response : no MAC no sequence number.
- SCP02 includes a sequence number and a complete response including a R-MAC.
- Expected weakness scenario:
  - There is no proof that a transaction finished correctly,
  - E.g : while loading an applet, an attacker can modify the Load-Install-MakeSelectable response (9000 -> 6xxx).

Any question ?