How to program Java Card 3.0 platforms ?

Samia Bouzefrane

CEDRIC Laboratory

Conservatoire National des Arts et Métiers

samia.bouzefrane@cnam.fr

http://cedric.cnam.fr/~bouzefra

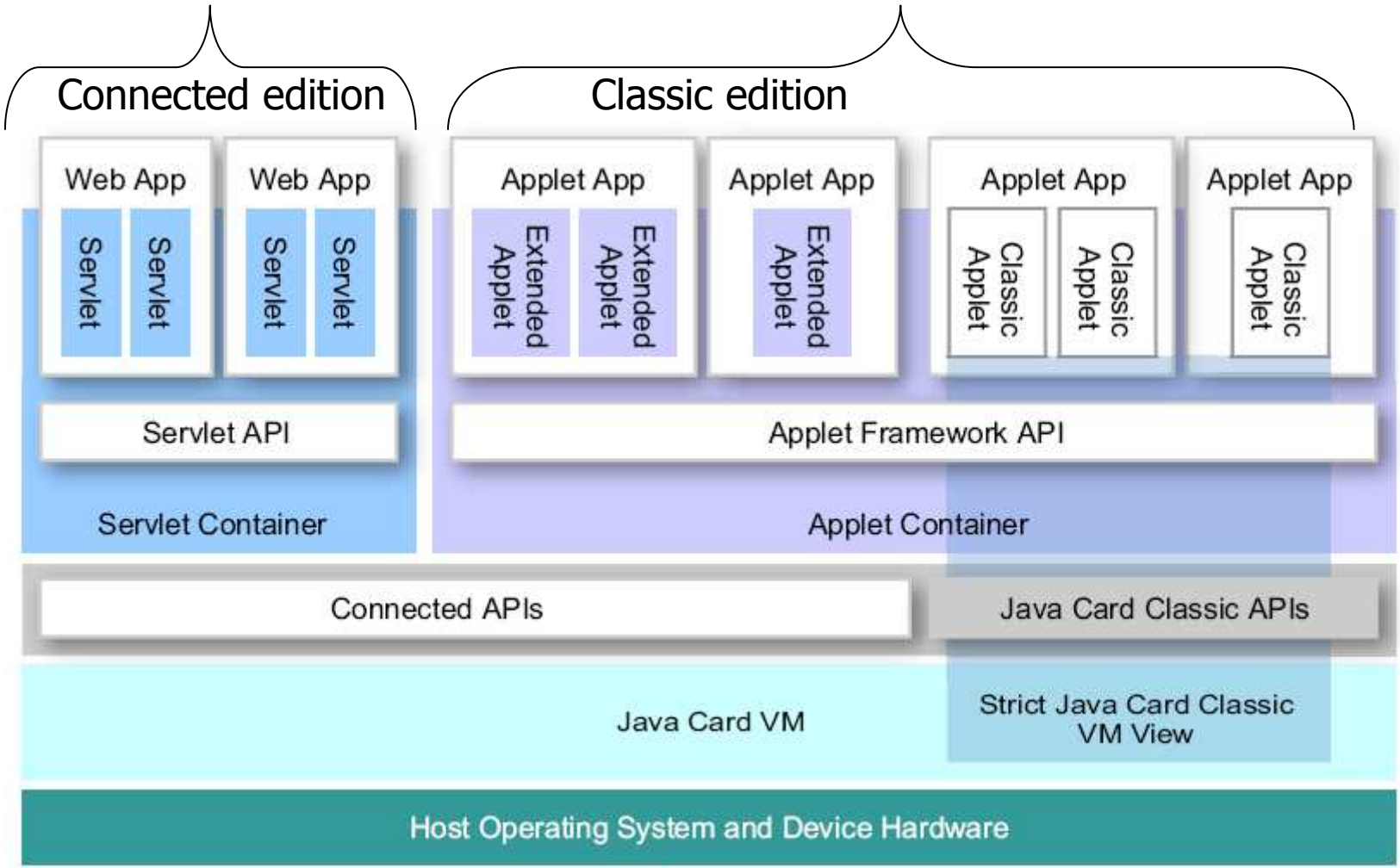
Smart University
Nice, Sophia Antipolis, September 22th 2009

Agenda

- Part 1: How to program a Web Application in JC3.0 ?
  - Java Card 3 overview
  - Java Card 3 features
  - Servlet-based applications
- Part 2: Evolution of the Java Card Framework
  - Java language subset
  - Persistence
  - Transactions
  - Multi-threading
  - Restarting Tasks
  - SIO
  - Event Handling

How to program Web Applications in JC3.0 ?

Java Card 3.0 Architecture



Connected Edition

Java Card 3.0, Connected Edition

- Includes many new features
  - 32-bit virtual machine,
  - multithreading,
  - event handling, *etc*.
- Targets IP-based Web applications
  - Suitable for Smart Card Web Servers
- Supports Java Card 2.2 applications

Classic Edition

- Java Card 3.0, Classic Edition
  - Evolution of Java Card 2.2.2
  - Very similar restrictions at the language level
    - 16-bit virtual machine and APIs
  - Targets APDU-based applications
  - Simple improvements
    - Enhanced support of contactless, new ISO7816-4
    - New cryptographic algorithms

Agenda

- Part 1: How to program a Web Application in JC3.0 ?
  - Java Card 3 overview
  - **Java Card 3 features**
  - Servlet-based applications
- Part 2: Evolution of the Java Card Framework
  - Java language subset
  - Persistence and atomicity
  - Multi-threading
  - Restarting Tasks
  - SIO
  - Event Handling

Principal Features/1

- At the virtual machine level
  - Standard distribution formats
  - Support for more basic types
  - Mandatory bytecode verification
- At the runtime environment level
  - Enhanced memory management (full GC, transactions)
  - Enhanced sharing framework
  - Improved security model

Principal Features/2

- At the API level
  - Support for String
  - Support for significant java.util subset
  - Support for generic connection framework (GCF)

- At the application level
  - New servlet application model
  - TCP/IP-based communication model
  - Enhanced sharing framework

Targeted devices

- The typical hardware configuration is as follows:
  - A 32-bit processor
  - 128 kilobytes of EEPROM
  - 512 kilobytes of ROM
  - 24 kilobytes of RAM

Agenda

- Part 1: How to program a Web Application in JC3.0 ?
  - Java Card 3 overview
  - Java Card 3 features
  - **Servlet-based applications**

- Part 2: Evolution of the Java Card Framework
  - Java language subset
  - Persistence
  - Transactions
  - Multi-threading
  - Restarting Tasks
  - SIO
  - Event Handling

What is a Web Application ?

- An extension of a Web server
- A Web server
  - Handles HTTP requests over a network
  - Returns content to the requesters
    - HTTP to be displayed by browsers
    - XML to be processed by applications
    - Other data types, as requested (images, videos, zip, …)
- A Web application contains
  - Some code that runs on the server
    - Produces content dynamically
  - Some static resources that complement the dynamic content

Example /1

File index.html

```
<html><head><title> Addition of integers </title></head><body>
<form method="POST" action="http://localhost:8019/AddServlet">
<center>Type the first number : <input name=nb1 type=text> <br>
<br>
Type the second number : <input name=nb2 type=text><br>
<br>
<input type=submit value=Send>
<input type=reset value=Cancel>
</center>
</form>
</body></html>
```

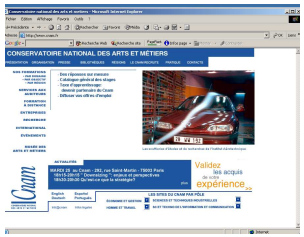
Example /2

File MyFirstServlet.java

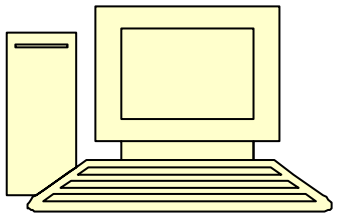
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyFirstServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
   // Step1. Specify the MIME type of the response
      response.setContentType("text/html");
  // Step2.  get the PrintWriter
  PrintWriter out = response.getWriter();
 // Step 3.  Send the data to the client
       out.println("<html>");
       out.println("<head><title> Servlet</title></head>");
       out.println("<body>");
       out.println("At this time : " + new java.util.Date());
       int a = Integer.parseInt(request.getParameter("nb1"));
       int b = Integer.parseInt(request.getParameter("nb2"));
       int c = a+b; out.println("The result is : " + c);
      out.println("</body></html>");
      out.close();
    }}
```

Standard Web Application

Web Browser Client



Web Server



Servlet Container

index.html

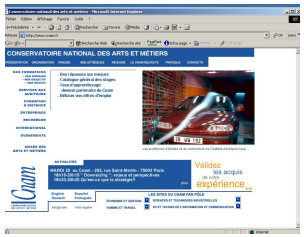
HTTP requests
(nb1=x, nb2=y)

MyFirstServlet execution

HTTP response (HTML page)

Other technologies

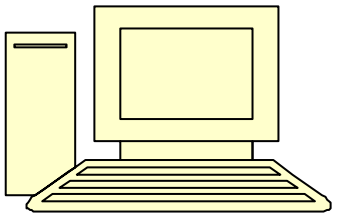
Web browser
Client



Requests

Responses

Web Server



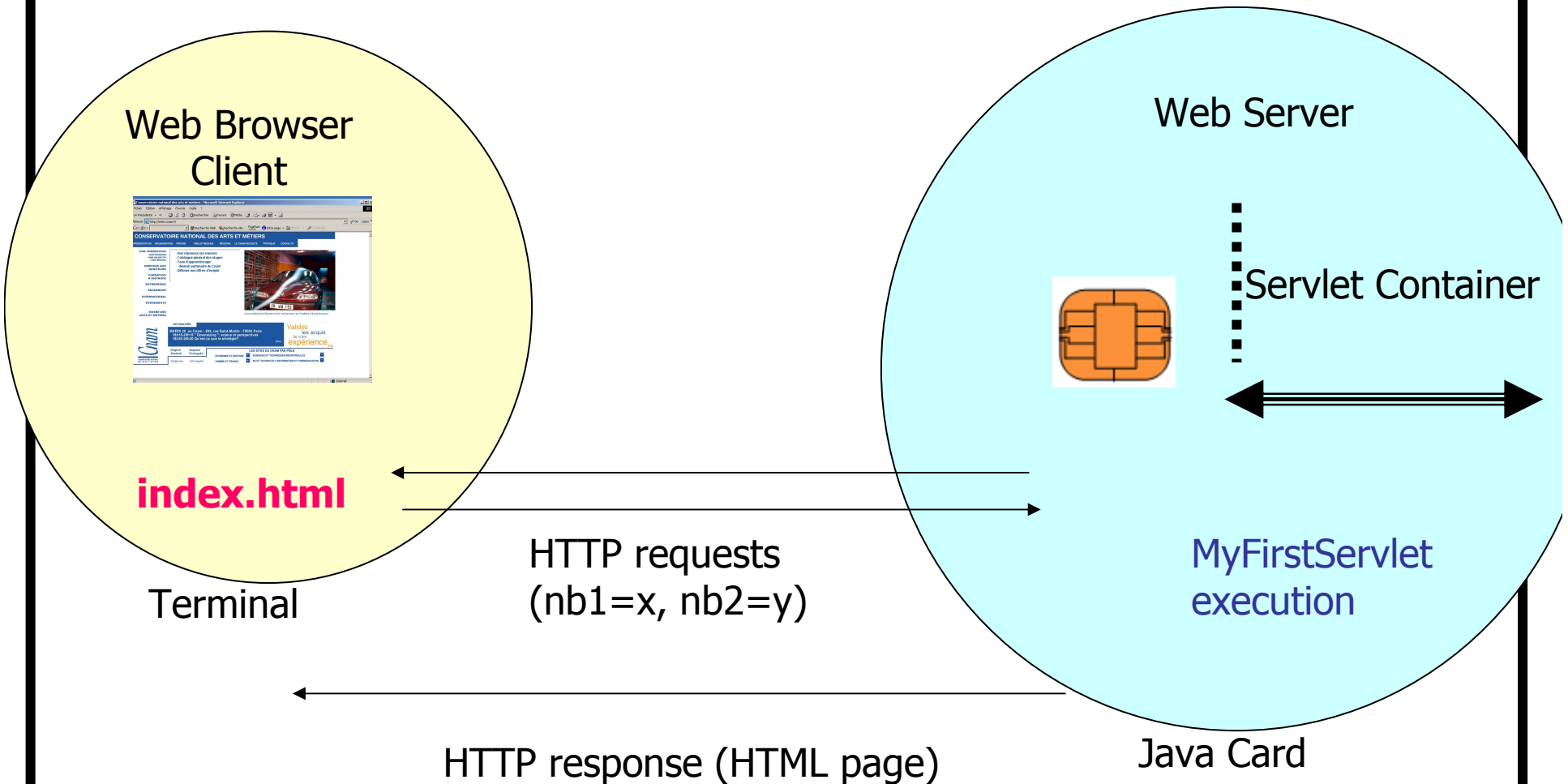
XML

Applet

Servlet/JSP
EJB

ASP

JC Web Application



HTTP Protocol

-Based on a client/server model

-Is used to transfer MIME typed resources :

text (HTML), image (JPEG, GIF), Audio (mpeg), application, Video (mp4), etc.

- HTTP request =

GET, POST, HEAD, PUT, DELETE, etc.

HTTP Methods

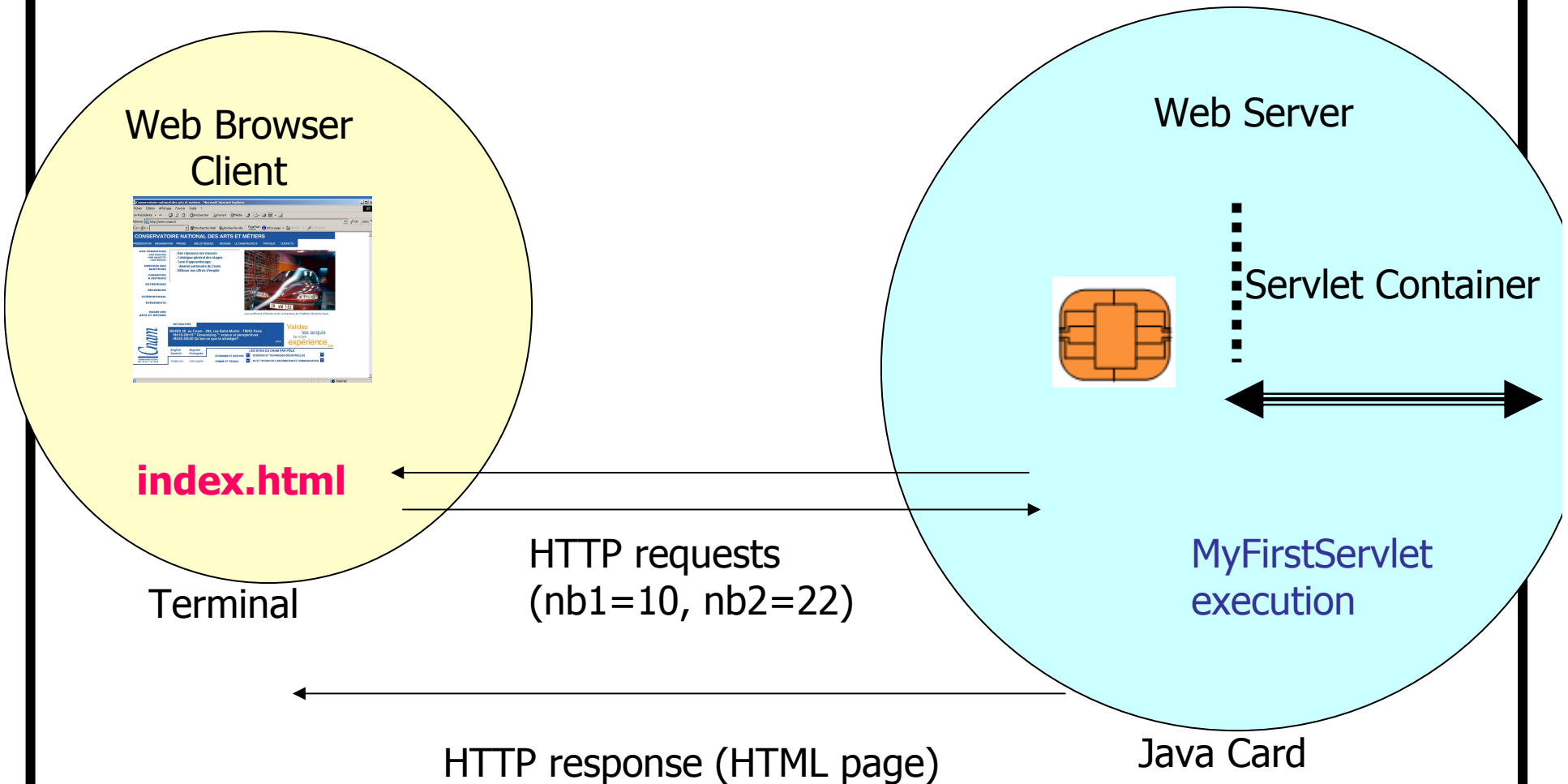
- HTTP 1.0 defines three methods
  - GET to request a document
  - POST to send data (from a form) to a script
  - HEAD to receive only the response header

- HTTP 1.1 adds five methods
  - OPTIONS to know which options apply to a resource
  - PUT to send a document to the server at a given URL
  - DELETE to delete the specified resource
  - TRACE to ask the server to return the request as-is
  - CONNECT to connect to a *proxy* that can do *tunneling*

The Servlet Application Model

- A servlet is an application that runs on a Web server
  - It runs on top of a "Web container"
  - The Web container dispatches requests to servlets
- A servlet application can
  - Process incoming requests
- Servlets often are an application's presentation layer

JC Web Application

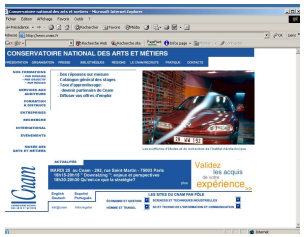


Servlet Methods to handle HTTP requests

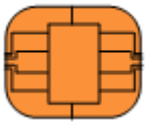
- doGet() method handles HTTP GET requests
- doPost() method handles HTTP POST requests
- doPut() method handles HTTP PUT requests
- doDelete() method handles HTTP DELETE requests
- doHead() method handles HTTP HEAD requests
- doOptions() method handles HTTP OPTIONS requests
- doTrace() method handles HTTP TRACE requests

service Method

Web browser Client



Web Server



Servlet Container

index.html

GET/POST HTTP requests
(nb1=x, nb2=y)

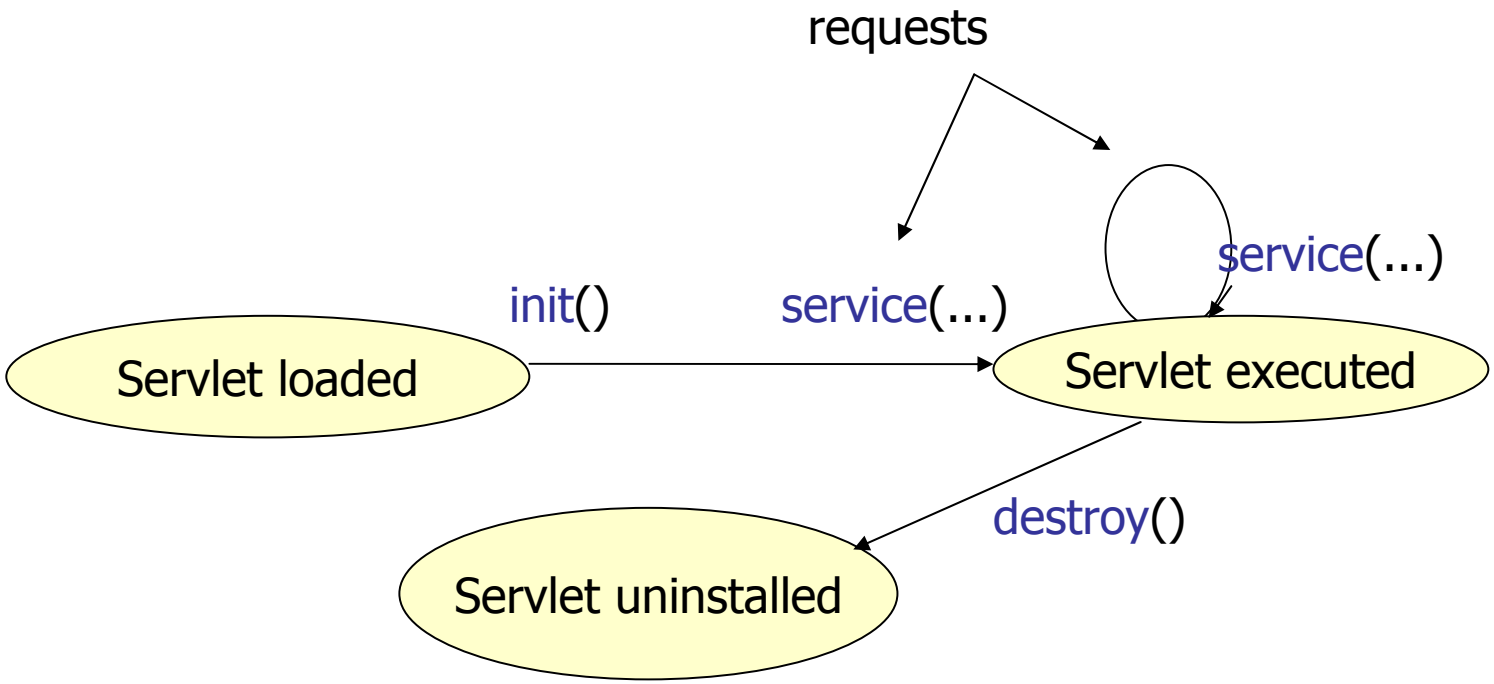
service() method

doGet()/doPost() method

HTTP response (HTML page)

MyFirstServlet()

Servlet life cycle



The Servlet loading & the state transition are done by the servlet container

`service`() method is triggered for each request.

How to program Servlets ?

- Use two packages :
  - javax.servlet : generic package
  - javax.servlet.http : package for Web servers

Principal Methods

❖ Three methods launched automatically :

– public void init(),
– public void service(...),
– public void destroy()

Defined in the abstract class javax.servlet.http.HttpServlet that extends javax.servlet.GenericServlet

❖ A Servlet must extend HttpServlet

service() Method

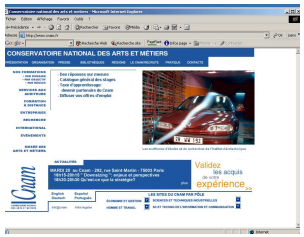
- service(...) contains two parameters :

```
protected void service(HttpServletRequest
req, HttpServletResponse resp) throws
ServletException, java.io.IOException
```

- HttpServletRequest req is the request
- HttpServletResponse resp refers to the response
- doGet() & doPost() have the same parameters.

How to get the request data ?

Web browser
Client



form.html

GET/POST HTTP requests
(nb1=x, nb2=y)

Web Server

Servlet Container

service() method

doGet()/doPost() method

String getParameter(String VariableName)

MyFirstServlet()

How to build a response ?

- We fix the MIME type :

```
void setContentType(String type)
```

- We use an output channel :

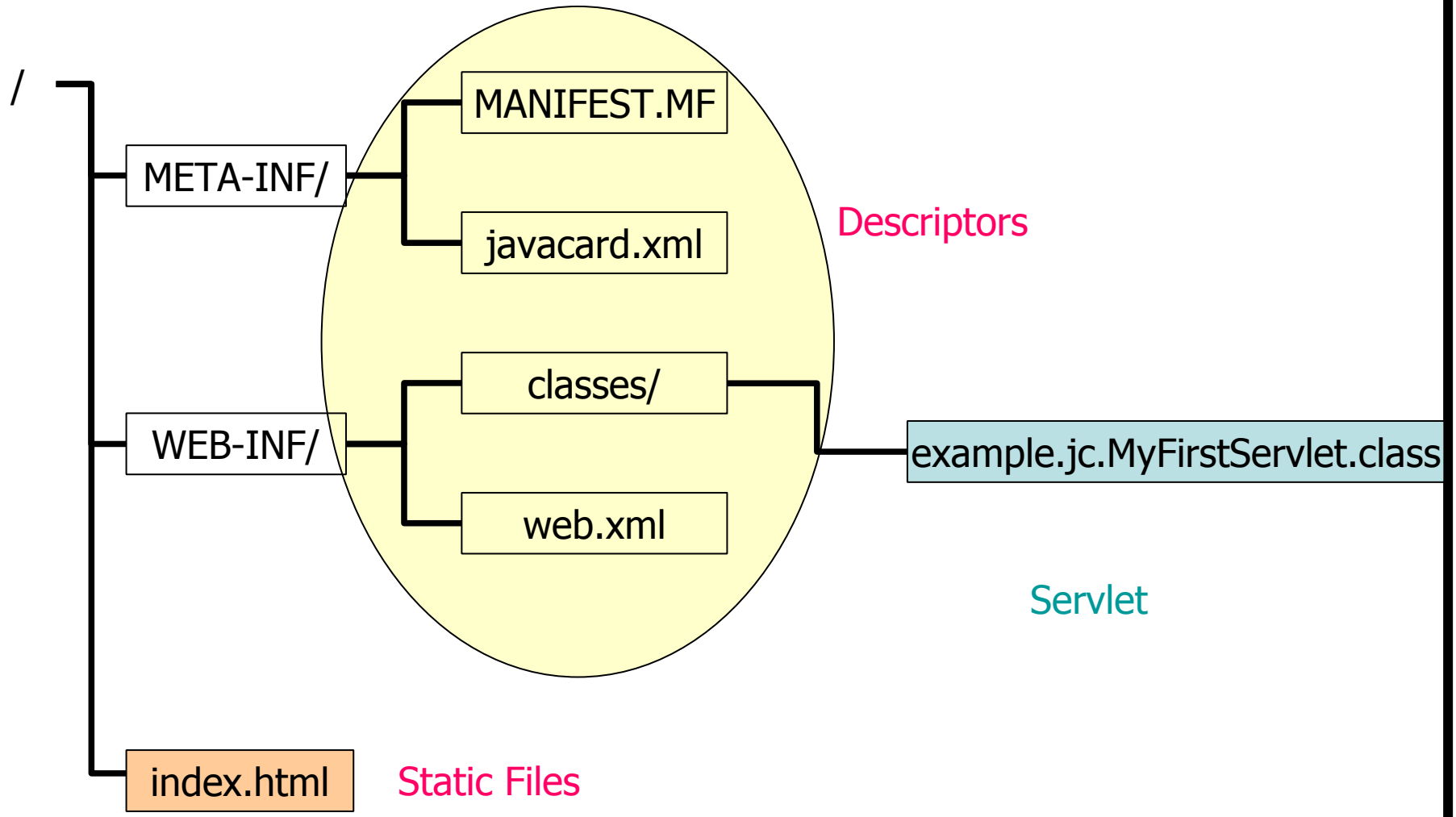
```
PrintWriter getWriter()
```

- to send the HTML response

A Servlet structure

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyFirstServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
   // Step1. Specify the MIME type of the response
      response.setContentType("text/html");
  // Step2.  get the PrintWriter
  PrintWriter out = response.getWriter();
 // Step 3.  Send the data to the client
       out.println("<html>");
       out.println("<head><title> Servlet</title></head>");
       out.println("<body>");
       out.println("At this time : " + new java.util.Date());
       int a = Integer.parseInt(request.getParameter("nb1"));
       int b = Integer.parseInt(request.getParameter("nb2"));
       int c = a+b; out.println("The result is : " + c);
      out.println("</body></html>");
      out.close();
    }}
```

Structure of the binary file



The Web Application Deployment Descriptor

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
      version="2.4"
      xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
          http://java.sun.com/xml/ns/jcns/jcweb-app_3_0.xsd">
  <display-name>Addition</display-name>
  <!-- Servlet classes -->
  <servlet>
    <servlet-name>myServlet</servlet-name>
    <servlet-class>example.jc.MyFirstServlet</servlet-class>
  </servlet>
  <!-- Servlet Mappings -->
  <servlet-mapping>
    <servlet-name>myServlet</servlet-name>
    <url-pattern>/AddServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

The Java Card platform-specific Application Descriptor

javacard.xml

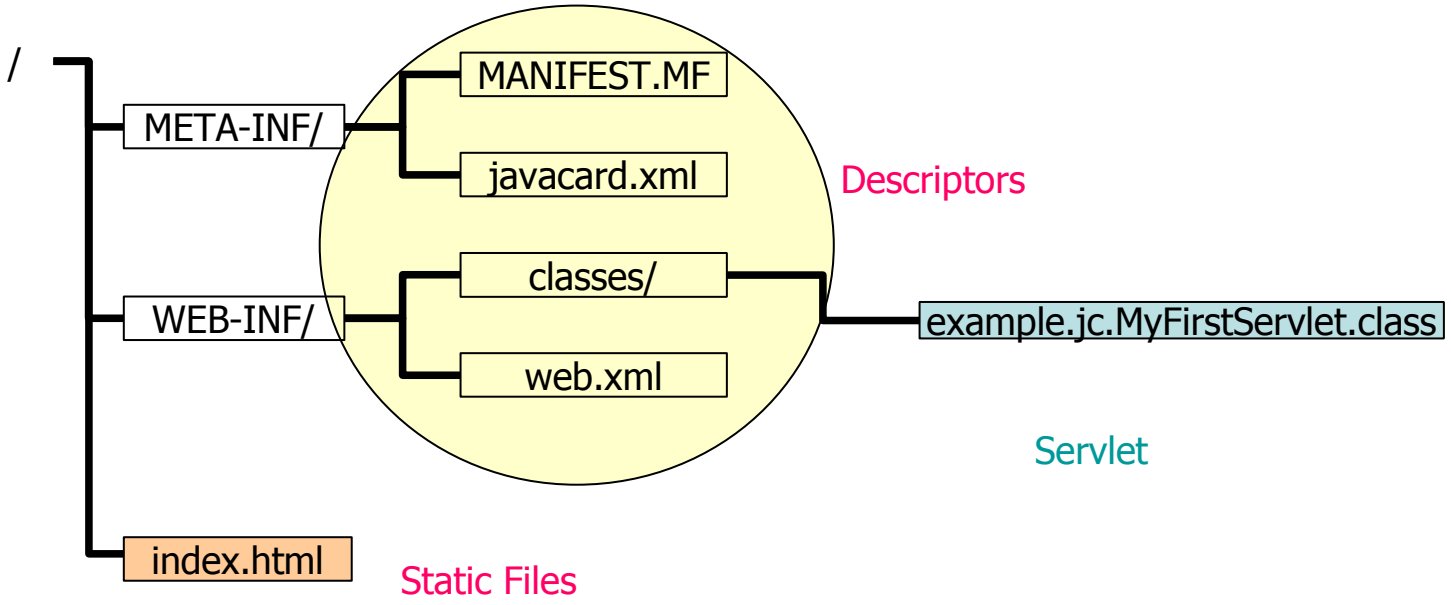
```
<?xml version="1.0" encoding="UTF-8"?>
<javacard-app
      version="3.0"
      xmlns="http://java.sun.com/xml/ns/javacard"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javacard
          http://java.sun.com/xml/ns/jcns/javacard-app_3_0.xsd">
  <!--  -->
</javacard-app>
```

The Java Card Runtime Descriptor

MANIFEST.MF

```
Manifest-Version: 1.0
Application-Type: web
Web-Context-Path: /AddServlet
Runtime-Descriptor-Version: 3.0
```

Application packaging



File structure stored in a WAR file

Part 2:

Evolution of the Java Card Framework

Agenda

- Part 1: How to program a Web Application in JC3.0 ?
  - Java Card 3 overview
  - Java Card 3 features
  - Servlet-based applications
- Part 2: Evolution of the Java Card Framework
  - **Java language subset**
  - Persistence
  - Transactions
  - Multi-threading
  - Restarting Tasks
  - SIO
  - Event Handling

Supported types

- More basic types are now supported
  - **`byte, short, int, long`**
  - **`boolean, char`**
  - Floating-point types still not supported

- More basic data structures are supported
  - All arrays, including multidimensional arrays
  - **`String`** and **`StringBuffer`**

More utility classes are supported

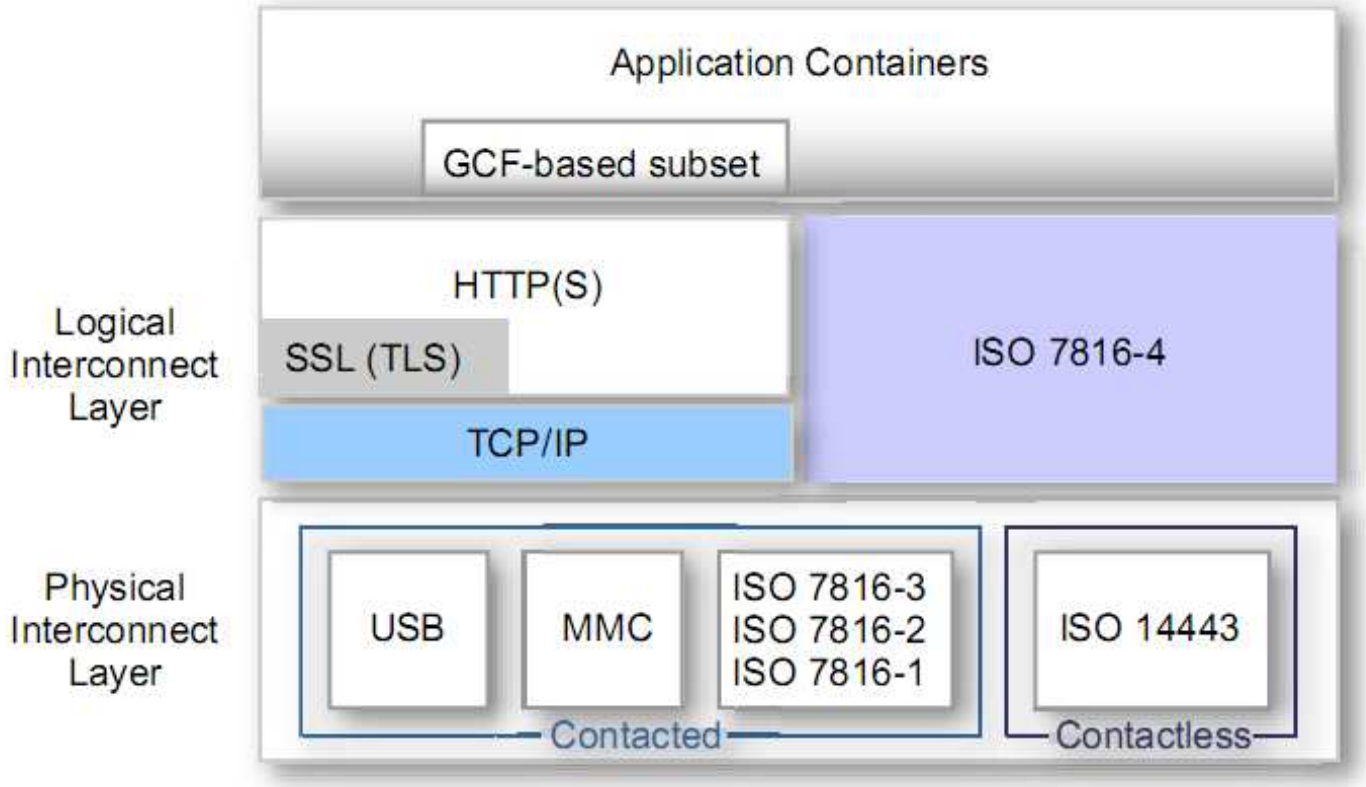
- A significant subset from **`java.util`**:
  - **`Stack, Vector, Hashtable, Enumeration, Iterator`**
  - **`Date, Calendar, TimeZone`**
  - **`EventObject, EventListener`**
  - **`Locale, ResourceBundle, ListResourceBundle`**
  - **`Random, StringTokenizer`**

- There are limitations, though
  - Classes themselves have been subseted
  - More recent collection classes are not available

Java Card Connected Edition

- Imported from CLDC/MIDP
- The **`java.io`** package is partly supported
  - **`PrintReader, PrintWriter`**
  - **`InputStream, DataInputStream`**
  - **`OutputStream, DataOutputStream`**
  - **`ByteArrayInputStream, ByteArrayOutputStream`**
  - …
- The **`javax.microedition.io`** package is supported
  - Supporting stream-based connections
  - **`Connector, InputConnection, HttpConnection`**, …

Java Card Connected Edition



Based on Java 6

- Many new constructs are supported
  - Generics
  - Enhanced for loops

```
Vector<String> vec = new Vector();
                     ...
for(String s:vec)
  translate(s) ;
```

  - Enums

```
public enum Deed { GOOD, BAD } ;

switch(deed) {
  case GOOD: ...
}
```

Agenda

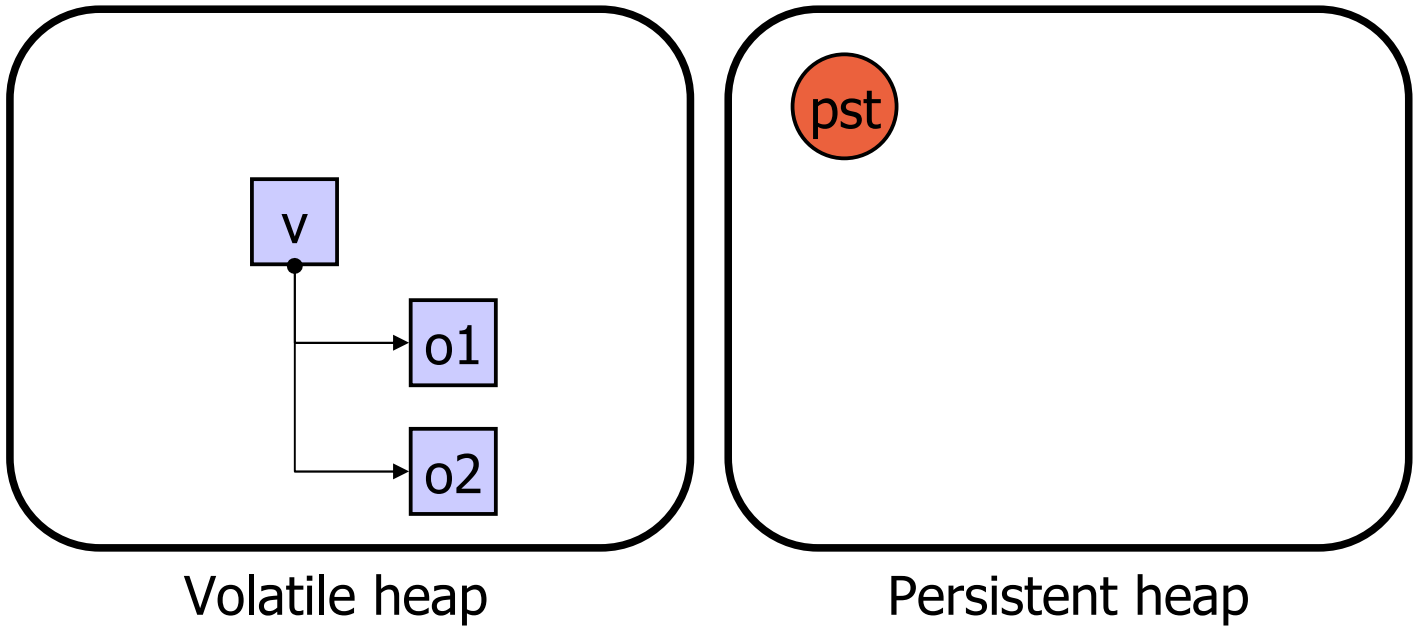
- Part 1: How to program a Web Application in JC3.0 ?
  - Java Card 3 overview
  - Java Card 3 features
  - Servlet-based applications
- Part 2: Evolution of the Java Card Framework
  - Java language subset
  - **Persistence**
  - Transactions
  - Multi-threading
  - Restarting Tasks
  - SIO
  - Event Handling

Reachability

- All objects that are reachable from persistent roots are persistent
- Persistent roots are:
  - Static fields
  - Instances of **`javacard.framework.Applet`**
  - Instances of **`javax.Servlet.ServletContext`**
- Newly created objects are volatile
- Transient objects are not persistent
- All unreachable objects are reclaimed by the garbage collector

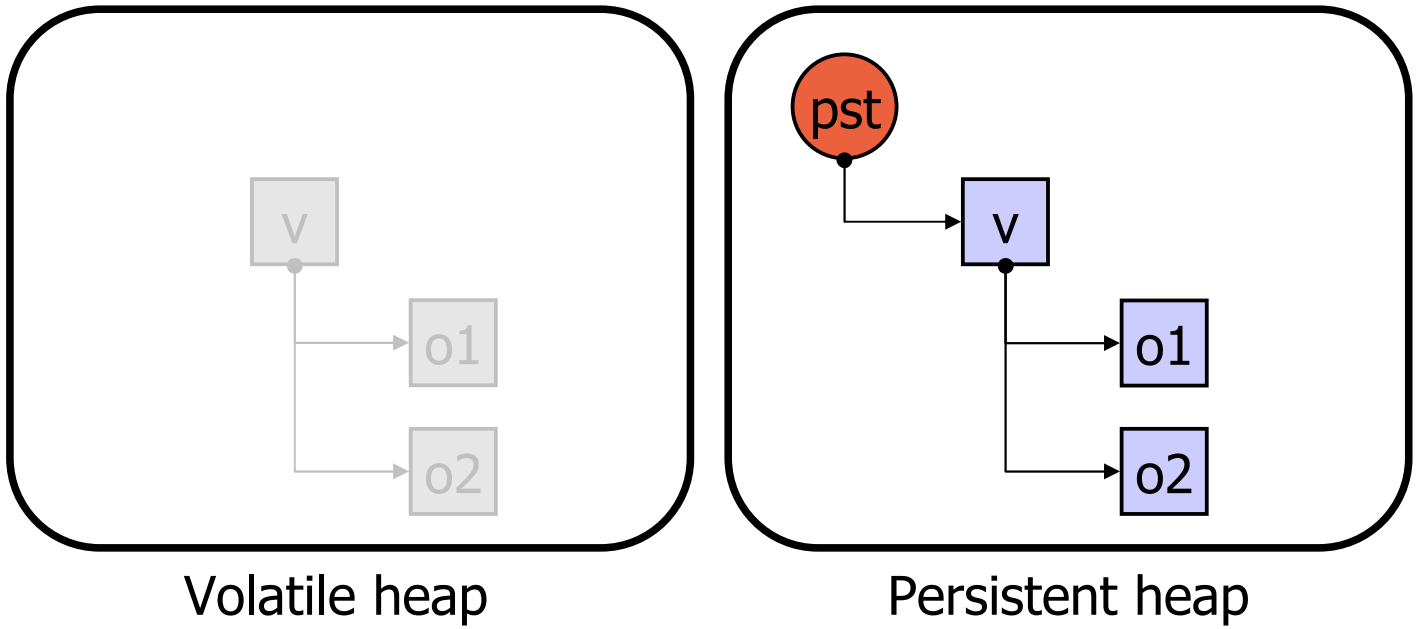
Reachability in practice

```
static Vector pst;

Vector v = new Vector();
v.addElement(o1);
v.addElement(o2);
pst = v ;

v.removeElement(o1);
```



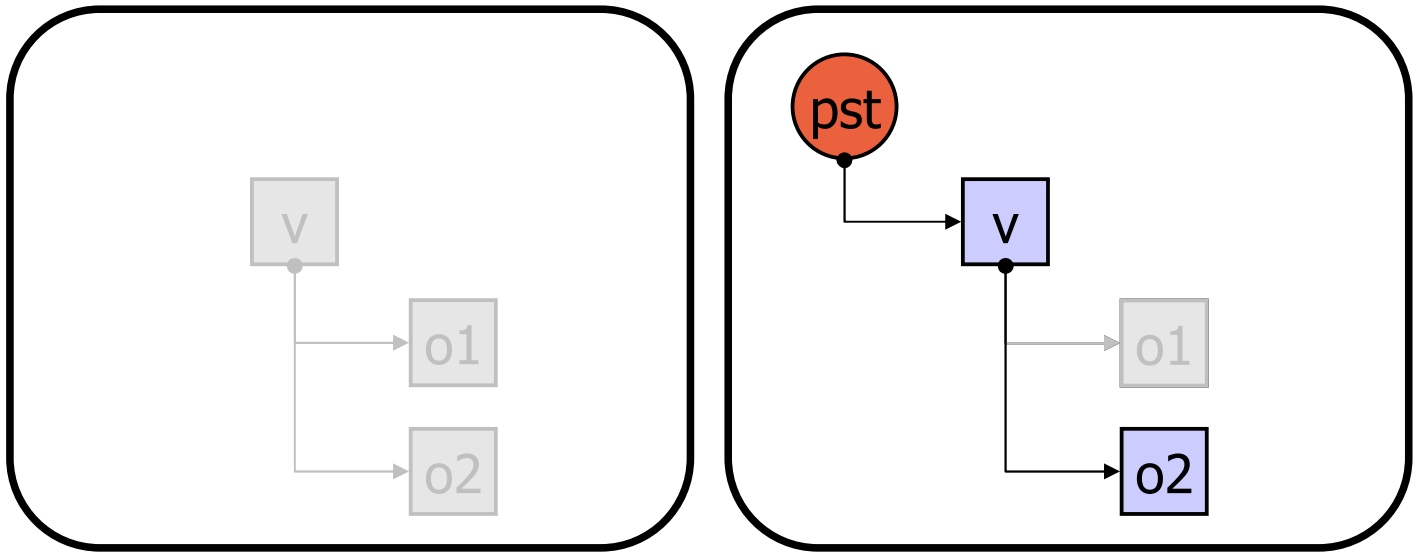
Reachability in practice

```
static Vector pst;

Vector v = new Vector();
v.addElement(o1);
v.addElement(o2);
pst = v ;

v.removeElement(o1);
```



Reachability in practice

```
static Vector pst;

Vector v = new Vector();
v.addElement(o1);
v.addElement(o2);
pst = v ;

v.removeElement(o1);
```



Agenda

- Part 1: How to program a Web Application in JC3.0 ?
  - Java Card 3 overview
  - Java Card 3 features
  - Servlet-based applications
- Part 2: Evolution of the Java Card Framework
  - Java language subset
  - Persistence
  - **Transactions**
  - Multi-threading
  - Restarting Tasks
  - SIO
  - Event Handling

Transactions

- Transactions have greatly evolved
  - Concurrent transactions are supported
  - Nested transactions are supported
  - Transaction control has changed

What was wrong before?

- Leading to problems in the case of a rollback

```
byte doSomething()
{
  byte[] ba ;
  JCSystem.beginTransaction();
    shortField = 12 ;
    ba = new byte[10];
    shortArray = ba ;
  JCSystem.abortTransaction();
  return ba[0];
}
```

How has it been fixed?

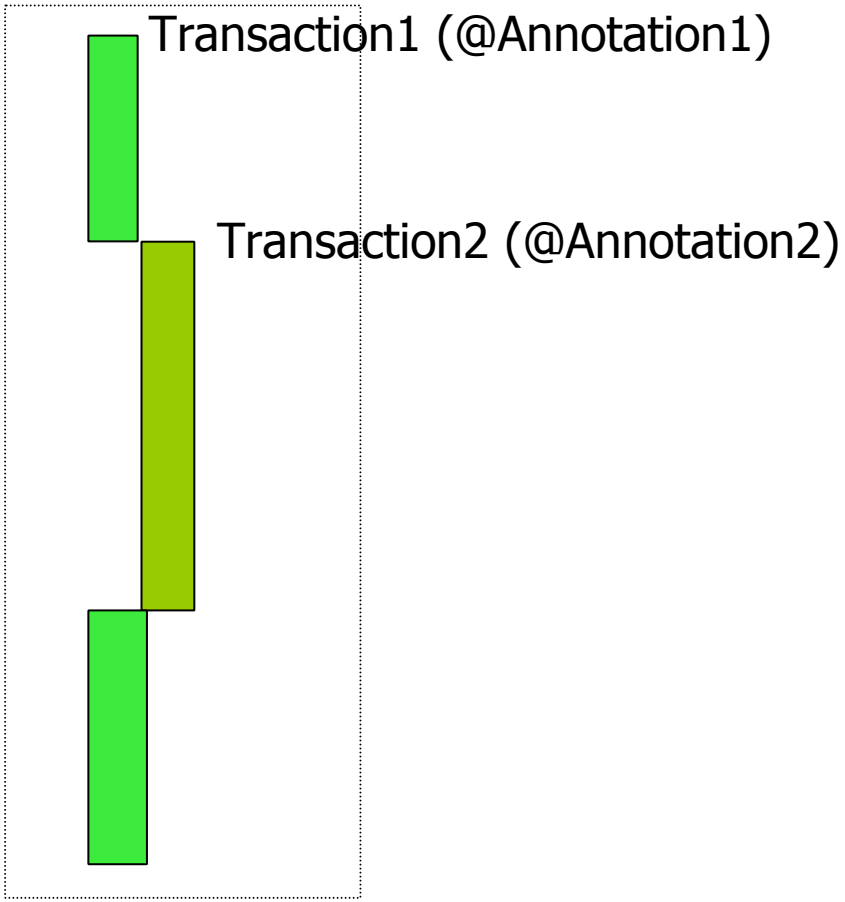
- Transaction status becomes a property of the method

```
@Transaction(REQUIRES_NEW)
byte doSomething()
{
  byte[] ba ;
  shortField = 12 ;
  ba = new byte[10];
  shortArray = ba ;
  return ba[0];
}
```

Principles

- Transaction starts at the beginning of a method
- Transaction is committed upon normal return
- Transaction is aborted on exception
- Behavior depends on annotation

Nested transactions with annotations



Behavior

			Method TransactionType Annotation					
		EVENT	REQUIRES_NEW	REQUIRED	SUPPORTS	NOT_SUPPORTED	MANDATORY	NEVER
Calling State	No transaction in progress	Enter method	Create new commit buffer **	Create new commit buffer **	No action	No action	Throw exception	No action
		Normal return	Commit updates **	Commit updates **	No action	No action	N/A	No action
		Exception return	Rollback updates in commit buffer **	Rollback updates in commit buffer **	No action	No action	N/A	No action
	Transaction in progress **	Enter method	Create new commit buffer **	Use caller's commit buffer **	Use caller's commit buffer **	Suspend transaction	Use caller's commit buffer **	Throw exception
		Normal return	Commit updates **	No action **	No action **	No action	No action	N/A
		Exception return	Rollback updates in commit buffer **	No action**	No action **	No action	No action	N/A

Agenda

- Part 1: How to program a Web Application in JC3.0 ?
  - Java Card 3 overview
  - Java Card 3 features
  - Servlet-based applications
- Part 2: Evolution of the Java Card Framework
  - Java language subset
  - Persistence
  - Transactions
  - **Multi-threading**
  - Restarting Tasks
  - SIO
  - Event Handling

Concurrency and multithreading

- Several applications may run concurrently
- An application may create threads
  - Some APIs have blocking methods
    - For instance, the Generic Connection Framework
- Programming with multiple threads is tricky
  - Especially the synchronization of data
    - A thread must gain exclusive access before a write
  - And then the potential deadlocks
    - Release early, require one exclusivity at a time, ...

Defining and running a thread

- Two ways to do that
  - Subclass **`Thread`**, define a **`run`** method
  - Implement **`Runnable`**, define a **`run`** method
- How do you start a thread?
  - Instantiate a **`Thread`** class, invoke **`start`** method
- How does a thread stop?
  - When the **`run`** method terminates
    - Invoking **`join`** makes a thread wait for another one
  - When it is **`interrupt`**'ed (maybe)

Static thread-related methods

- **`Thread.sleep`** waits for some milliseconds
  - Depends on the internal clock's accuracy
- **`Thread.yield`** relinquishes control to the system
- **`Thread.currentThread`** returns
  - a reference to the current thread
- **`Thread.activeCount`** returns
  - the current number of active threads

Example of Threads

```
public class Triple extends Thread {
        int nb;
        Triple(int nb) {
        this.nb=nb;
        }
        public void run() {
                int b=3*nb;
        }
}
public class RunTriple {
        static void main(String a[]) {
                Triple tt= new Triple(5) ;
                tt.start() ;
                …
                tt.join();
                }
}}
```

Example of Thread Group

```
public class OtherManner extends Object {
{
Runnable objA = new Runnable() {
        public void run() {
        int a=Thread.currentThread().getPriority();
        Thread.sleep(a); }
        } ; // end of objA

ThreadGroup root = new ThreadGroup (“Root") ;
Thread ta= new Thread(root, objA, "ThreadA")
ta.start() ;
…
ta.join();
} // end of class
```

Synchronisation between Threads

```
public class AddSub  {
       private int Number =0 ;

public synchronized int Add (int a) throws
InterruptedException  {
              Number = Number +a;
              return Number ;
       }

public synchronized int Sub (int b) throws
InterruptedException  {
              Number = Number -b;
              return Number ;
       }


}
```

Synchronisation between Threads

```
class Thread1 extends Thread {
       AddSub as;
       Thread1 ( AddSub as ) {
              this.as=as;
       }
       public void run() {
       try {
         int n= (int)(1000*Math.random());
         int m=as.Add(n) ;
         Thread.sleep((int)(m);
       }
       } catch(InterruptedException e) {}
}// end of run
} // end of class
```

In Practice

- Programming parallelism is useful
- Remember, this is a smart card
  - The number of threads may be severely limited
- Be careful when programming concurrency

Sessions and events

- In Java Card 3.0, applications can monitor events
  - Registering a task for restarting automatically
  - Registering for being notified of important events

Agenda

- Part 1: How to program a Web Application in JC3.0 ?
  - Java Card 3 overview
  - Java Card 3 features
  - Servlet-based applications
- Part 2: Evolution of the Java Card Framework
  - Java language subset
  - Persistence
  - Transactions
  - Multi-threading
  - **Restarting Tasks**
  - SIO
  - Event Handling

Restartable Tasks

- Applications can register restartable tasks
  - Tasks are restarted when the container initializes
  - Tasks are started only once per session
  - Each task is started in a separate thread

- Why register a task?
  - Standard servers only need to receive requests
  - There are a few reasons
    - A Web app may initiate a connection on start-up
    - A Web app may prepare itself at start-up

Restartable Tasks are Threads

Task objects must implement the java.lang.Runnable interface

```
public class MyTask implements Runnable {
…
     public void run() {
      }
}
```

Task Registration

- The application must register the task object with the registry by calling **`TaskRegistry.register`** method
- The application may specify if the task is to be executed immediately or upon the next platform reset

```
public class InfoServerServlet extends HttpServlet {
    public void init(ServletConfig config) throws
                                    ServletException {
        ...
        MyTask task = new MyTask();
        TaskRegistry.getTaskRegistry().register(task, true);
        ...
    }
...
}
```

Task Restart/Unregistration

•Tasks is restarted after the application containers have been restarted but before any request gets dispatched to applications.

•Restartable tasks are owned by the Java Card RE.

•A registered task is removed from the registry if :
- the owning application unregisters it
- the owning application is deleted.

•The application must call the **`TaskRegistry.unregister`** to unregister a task.

•The TaskRegistry class is annotated with the NOT_SUPPORTED TransactionType annotation.

Agenda

- Part 1: How to program a Web Application in JC3.0 ?
  - Java Card 3 overview
  - Java Card 3 features
  - Servlet-based applications
- Part 2: Evolution of the Java Card Framework
  - Java language subset
  - Persistence
  - Transactions
  - Multi-threading
  - Restarting Tasks
  - **SIO**
  - Event Handling

SIO

- is a communication facility
- SIO : Shareable Interface Objects
- SIO mechanism allows for:
  - Applications to define and dynamically register (and unregister) SIO-based services
  - Applications to lookup SIO-based services registered by other applications

SIO entities

1. The Service Registry
   - is a singleton
   - is a permanent Java Card RE entry point object
   - is called thanks to ServiceRegistry.getServiceRegistry method
   - is annotated with the NOT_SUPPORTED TransactionType annotation

2. The server application
   - Registers a service factory within the service registry.

3. The client application
   - Looks up to the service

SIO interaction process

1. The server application registers a service factory within the service registry.

2. The client application looks up to the service

3. The service registry invokes the service factory

4. The service factory creates the SIO object

5. A reference is returned to the client

6. The client calls the SIO methods using the obtained reference

A shareable interface

• First we define a shareable interface with the offered services.

```
import javacard.framework.Shareable;

public interface MySIOInterface extends Shareable {
    void setValue(int value);
    int getValue();
}
```

A shareable object

- An SIO Object must implement an interface that extends the **`javacard.framework.Shareable`** interface

```
public class MySIO implements MySIOInterface {
    private int value = 0;

    public void setValue(int value) {
        this.value = value;
    }
    public int getValue() {
        }
        return value;
    }
}
```

Service Factory

• Service factory objects implement the javacardx.facilities.ServiceFactory interface.

```
import javacard.framework.Shareable;
import javacardx.facilities.ServiceFactory;
public class MySIOFactory implements ServiceFactory {
    MySIO sio = new MySIO();
    public MySIOFactory() {
        //
    }
    public Shareable create(String serviceURI, Object param) {
        return sio;
    }
    public MySIO getSIO() {
        return sio;
    }}
```

Service Factory Registration

- The server application must register a service factory under a unique service URI, by calling the ServiceRegistry.register method.

```
public class MyServlet extends HttpServlet {

    public static final String SIO_URI = "sio:///sioservice/mySIO";

    public void init(ServletConfig config) throws ServletException {
        ...
        MySIOFactory factory = new MySIOFactory();
        ServiceRegistry.getServiceRegistry().register(SIO_URI, factory);
        ...
    }
```

Client Application

• The client application must share with the server application the interface of the shareable object.

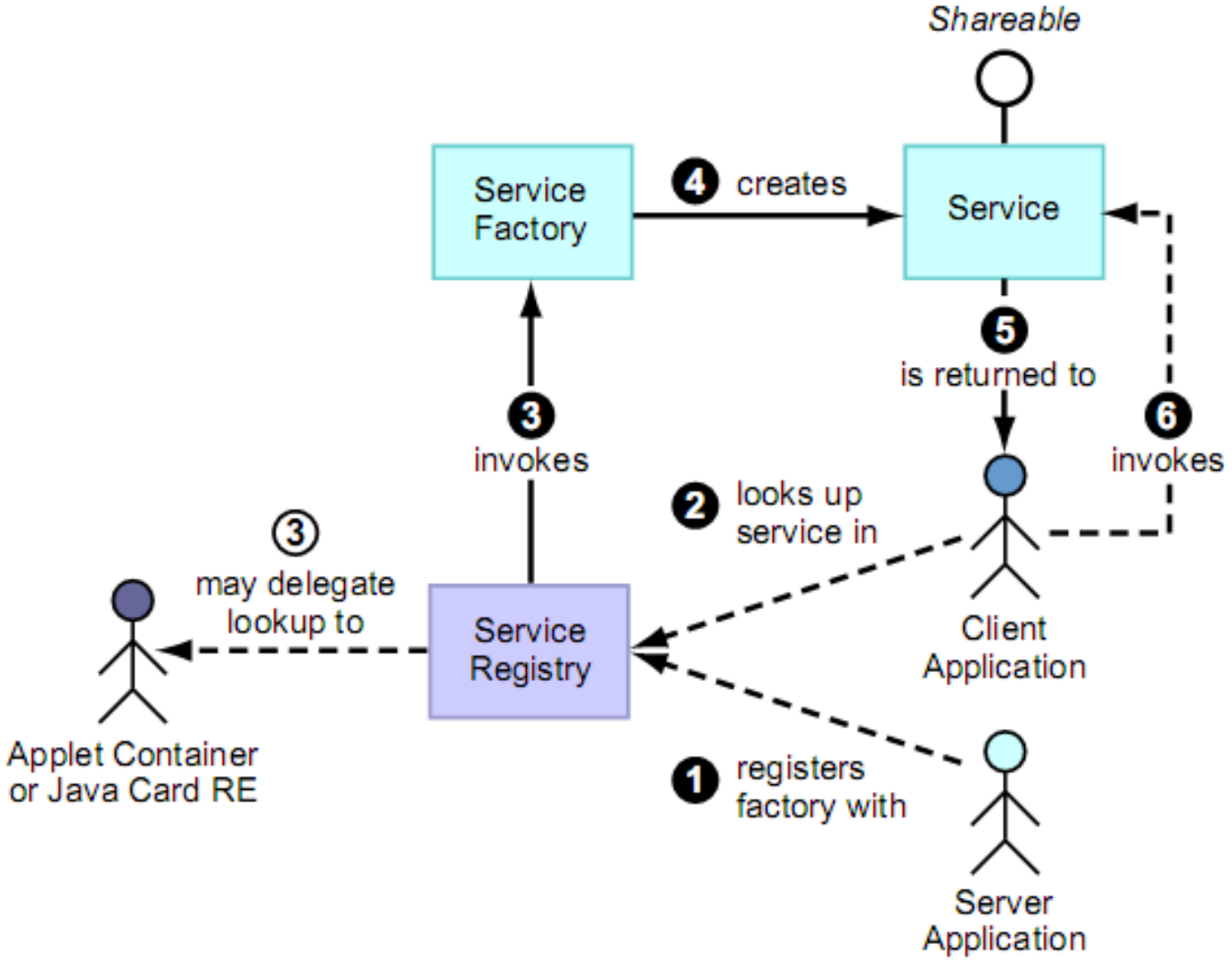
```
import javacard.framework.Shareable;

public interface MySIOInterface extends Shareable {
    void setValue(int value);
    int getValue();
}
```

SIO Lookup

- When the client calls the *ServiceRegistry.lookup* method, the service registry
  - looks up the registered service factory and
  - invokes its *create* method.

```
public class ClientServlet extends HttpServlet {
    public static final String SIO_URI = "sio:///sioservice/mySIO";
...
    public void doGet(HttpServletRequest request, HttpServletResponse response)
     throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
MyShareable ss = (MyShareable) ServiceRegistry.getServiceRegistry().lookup(SIO_URI);
        ...
      out.println("       <b> SIO Value is: " + ss.getValue() + "!</b>");
}
```

SIO Interactions



Agenda

- Part 1: How to program a Web Application in JC3.0 ?
  - Java Card 3 overview
  - Java Card 3 features
  - Servlet-based applications
- Part 2: Evolution of the Java Card Framework
  - Java language subset
  - Persistence
  - Transactions
  - Multi-threading
  - Restarting Tasks
  - SIO
  - **Event Handling**

Event Handling

- What kind of events can be notified?
  - Not the start-up: restartable tasks cover that need
  - Application- and resource-related events
    - Creation, deletion
  - Clock resynchronization with a trusted source
    - Allows applications to perform date-related operations

Event Handling

- Is based on the inter-application communication facility

- Allows asynchronous communication between web applications and applet applications through events.

- The event notification facility allows for:
  - Applications to dynamically register and unregister events
  - Applications to define and fire events.

Event entities

1. The Event Registry
   - Invokes the event listeners to handle the events.

2. The producing application
   - Creates the event and notifies the event registry.

3. The consuming application
   - Registers an event notification listener for an event's URI

4. The events are Shareable Interface Objects.

The Event Registry

• The **`javacardx.facilities.EventRegistry`** class allows for applications to fire events and/or register for notification of events fired by the Java Card RE or other applications.

• The EventRegistry instance :

- is a singleton
- is a permanent Java Card RE entry point object
- is retrieved by calling the EventRegistry.getEventRegistry method.
- is annotated with the NOT_SUPPORTED TransactionType annotation.

• An application is allowed to use the event registry until it is registered, that is until it has been assigned an application URI.

Event Listener

Event listener objects implement the javacardx.facilities.EventNotificationListen interface.

```
import javacardx.facilities.EventNotificationListener;
import javacardx.facilities.SharedEvent;

public class MyListener implements EventNotificationListener {
...
    /**
     * The notify() method is called when requested event is fired
     */
    public void notify (SharedEvent sharedEvent) {
...}
}
```

Event Listener Registration

An application registers an event notification listener for that event's URI.

The registering application calls the EventRegistry.register method and provides:

- An optional event source URI identifying an application, a resource or the platform
- An exact or a path-prefix URI pattern identifying a set of events
- A listener object, which will handle the event notification.

The same event listener object may be registered for multiple event URIs.

```
public void init(ServletConfig config) throws ServletException {
        super.init(config);
        MyListener myListener = new MyListener();
        EventRegistry.getEventRegistry().register("event:///eventsender/testEvent",
                                                  myListener);
    }
```

Event class

- The events are Shareable Interface Objects.

- An event class must extend the **`javacardx.facilities.Event`** class, which implements the **`javacardx.facilities.SharedEvent`** interface (which itself extends the **`javacard.framework.Shareable`** interface)

Event Listener notification

- The producing application calls the **`EventRegistry.notifyListeners`** method and provides an instance of the Event class with an event source URI.

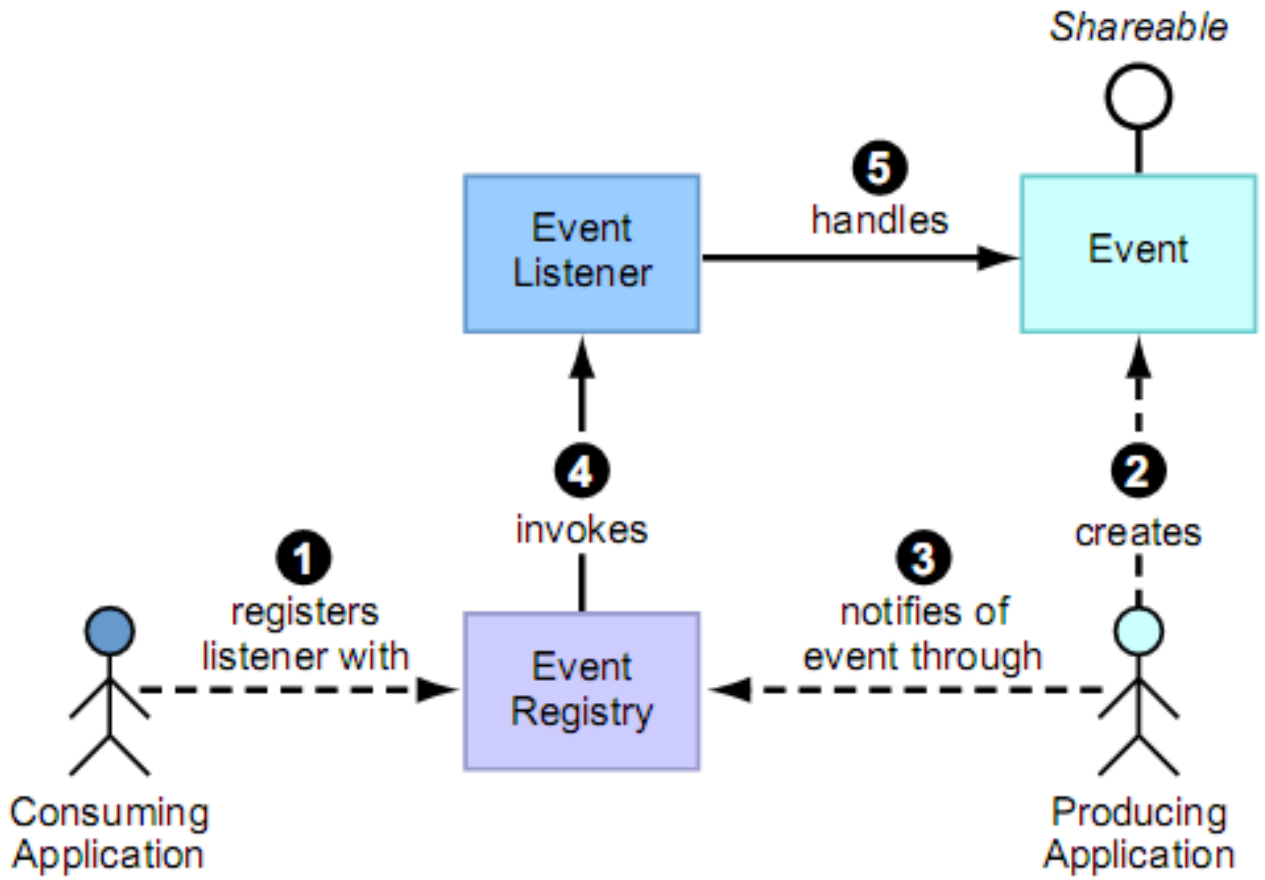
```
String data = request.getParameter("data").intern();
...
Event event = new Event("event:///eventsender/testEvent", data);
EventRegistry.getEventRegistry().notifyListeners(event, true);
```

Event notification

• The event registry looks up all event listeners registered for that event URI and it invokes each listener's notify method in sequence passing the event as a parameter.

• The order in which the event listeners may be notified is non deterministic.

Event Interactions



Conclusion

- Java Card 3.0 :
  - Is powerful
  - Integrates modern programming concepts
  - Handles complex applications
  - Is becoming an element of the Web.
- Don’t forget that the platforms are still limited
- Performance may remain an issue

References

- Java Card 3.0 : http://java.sun.com/javacard/3.0/
- Eric Vetillard, “The art of Java Card 3.0 Programming”, dec. 2008.
- Development Kit User’s Guide, Java Card™Platform, Version 3.0.3 Connected Edition, 2010, Oracle.