JAVA DAYS ´98


# Java™ Card
# the Java™ Standard for Smart Cards

Thomas Frey
thomas.frey@gdm**.de**
Research&Development
Giesecke&Devrient

Giesecke & Devrient

# What is Java™ Card ?


- A standard of SUN™
- A new way of programming smart cards in high level language
  -> Java
- The customer possibility to write his own chip card application
  -> fast time to market
- Run the same application on every chip card vendors Java Card
  -> interoperable / second source
- Running an interpreter
  -> subset of the Java Virtual Machine
- Secure Applications protected against other Applications
  -> firewall



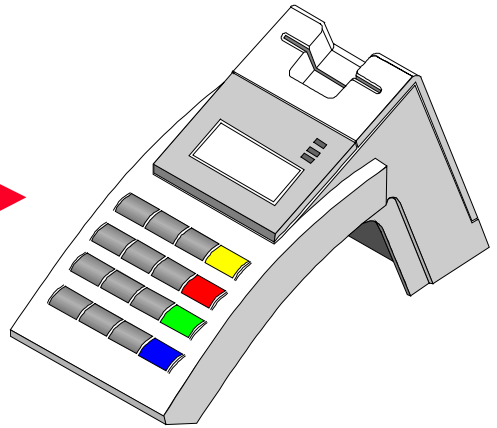
Giesecke & Devrient



Sun microsystems

# What is a Smart Card


ICC (Intelligent Chip Card)



IFD (Interface Device)

- A plastic carrier containing an chip card module with a microprocessor
- hardware defined in ISO/IEC 7816 Part 1 and 2



Giesecke & Devrient

# What are the use case of Smart Cards


- Save and secure user specific data
- An user specific token to identify user or other tokens against a systems
- Authentication, Authorization and Certification
- Provide use of cryptography in tolerable way in the case of export license
- Handle Electronic money and electronic payment transactions
- Protect secure Algorithms against reengineering
- SAM's (Security Administration Module)







# Standards Identifying Smart Cards


- ISO 7810: 1985
  Identification cards – physical characteristics
- ISO 7811: 1985
  Identification cards – recording technique
  Magnetic stripe and embossing
- ISO/IEC 7816
  Identification cards – integrated circuit cards with contact
  Part 1-11
- ISO/IEC 10373: 1993
  Identification cards – test methods
- CEN/TC 224/WG 9 = ETSI/TE9
  Multifunctional cards for use in telecommunications
- EMV'96
  for using cards in electronic payment systems of Europay, Mastercard and VISA

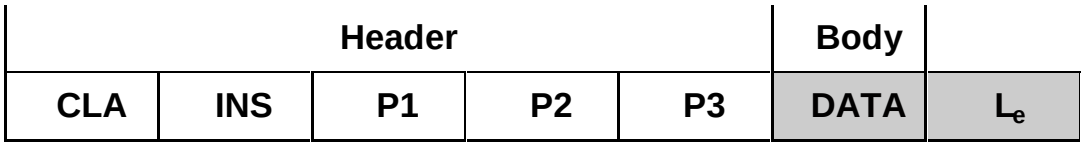


Giesecke & Devrient

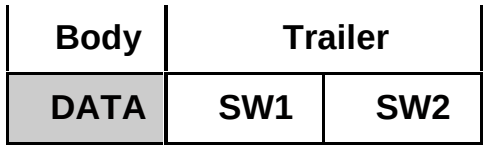


Sun microsystems

# Smart Card Application Protocol Data Unit (APDU)




Command APDU




Response APDU

CLA: Class Byte '00' -> command according to ISO 7816-4
'80' -> private use command

INS: Instruction Byte

P1: parameter 1

P2: parameter 2

P3: case 1: neither send nor receive data -> P3 = '00'
case 2: send data, but do not receive data -> P3 = $L_c$
case 3: do not send data, but receive data -> P3 = $L_e$
case 4: send and receive data -> P3 = $L_c$, $L_e$ = data expected

DATA: data to be transferred

$L_e$: length of data expected

$L_c$: length of data to be sent



Giesecke & Devrient



Sun microsystems

# Smart Card Application Protocol


- defined in ISO/IEC 7816 Part 4 - 11 for common use
- defined in ETSI (f.l. GSM11.11) specifications for telecommunication
- defined in EMV'96 (Europay, Master Card, VISA) for payment system
- diverse national, and applications specific specifications of bigger card issuers, like Geldkarte, VISA Cash, mostly according to ISO, EMV or ETSI



Giesecke & Devrient



Sun microsystems

# Smart Card Basics : AID - Application Identifier (1)


Unique identifier for Applications on Smart Cards

Registration authorities are in charge of national registrations (category national registration)

Registration categories:

| | |
|---|---|
| 0 – 9 | defined in ISO 7812 |
| A | international registration |
| D | national registration |
| F | unregistered applications |



# Smart Card Basics : AID - Application Identifier (2)


maximum length: 16 Bytes

registered application provider (RID): 5 first Bytes

proprietary application identifier extension (PIX) : ≤ 11 following Bytes

e.g.: ‘D2 76 00 00 05 xx xx ... xx’

‘D2 76 00 00 05’ - RID

'D' – category national registration

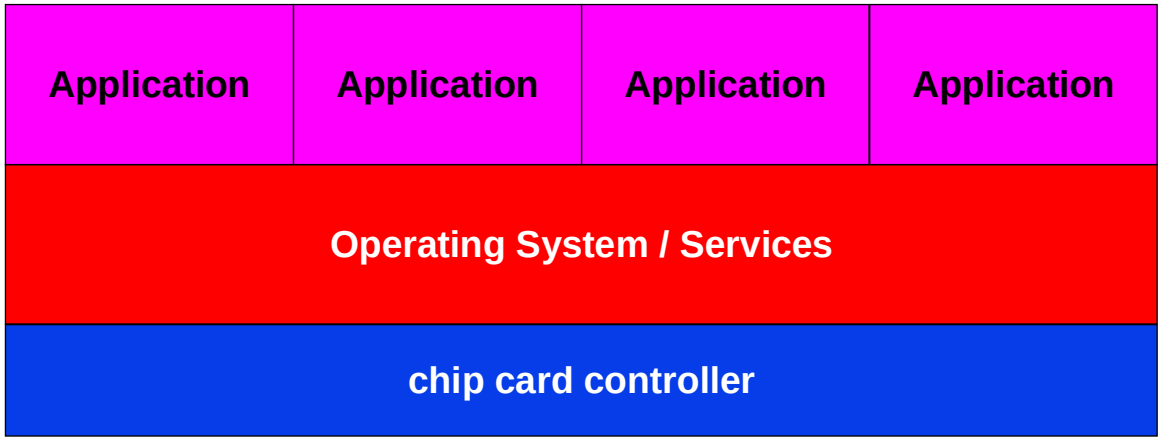
'2 76' – country code for Germany

'00 00 05' – national registration number for G&D

'xx xx ... xx' – PIX (max. 11 Bytes)



# Smart Card Software today ?




- File system based Operating System and one or more specific application running
- Operating Systems are depending on card vendors as well as on the system using the specific smart cards (GSM, Geldkarte, VISA Cash, ...)
- There is actual no common operating system defined for all markets and card applications
- Smart card operating systems are mostly optimized to the requirements of their field of use



Giesecke & Devrient



Sun microsystems

# The Java Card Technology


- Actual version is 2.1
- It's based on the Java Standard and Java Virtual Machine Standard from SUN
- The system is designed for memory constrained devices, especially smart cards
- Transfer the benefits of Java to smart cards, security, robustness, tools, interoperability and portability
- The Java standard for smart cards is a subset of the Business Java standard







# JCVM: unsupported features


- dynamic class loading
- garbage collector and `finalize()` method
- threads
- cloning
- default visibility override
- `package` visible interface can not be extend to `public` visibility
- `double`, `float` and `long` are not allowed
- static instantiated classes



# JCVM: differences to standard JVM


- Intentional a 16-bit system, with 16-bit stack size
- security manager policy is implicit implemented in the VM and not an explicit class
- standard Java classes are different, in general none of the standard classes are supported
- There is a one System class, `javacard.framework.JCSystem`
- persistence and transience are not the same like in business Java
- loadable parts are packages not classes
- arrays are single dimensional







# JCVM: explicit supported features


- Packages
- Dynamic object creation
- virtual methods
- interfaces, abstract classes, exception handling
- visibility attributes `private`, `package`, `protected`, `public`
- root class is `class Object`



Giesecke & Devrient



Sun microsystems

# JCVM: optional items


**using optional features are in contradiction with interoperability**

- **data type integer**

It is recommended to build a `class LongArithmetic` instead !!!







# JCVM: limitations

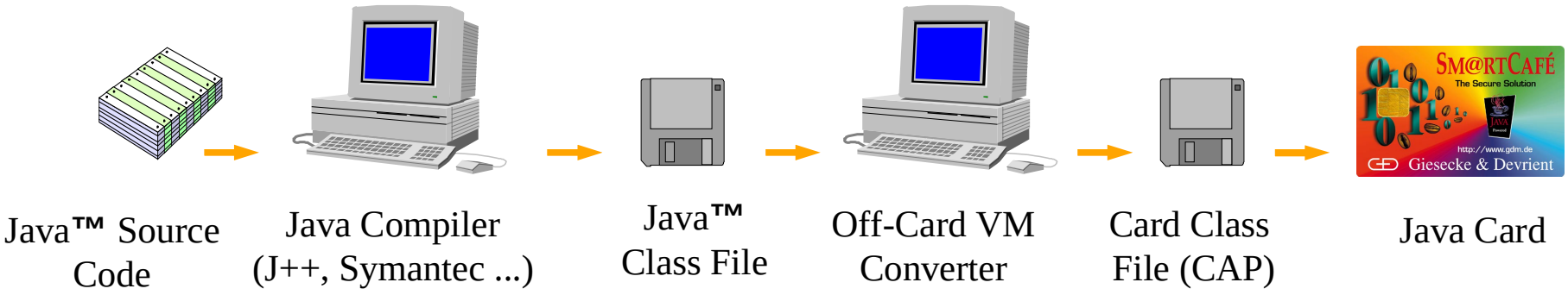

- Memory space is limited to 64KB per applet
- in maximum 128 public/protected instance and 128 package visible instance methods per class, with all inheritance are possible
- in maximum 256 instance fields (related to stack size, 16-bit) per class are possible
- array size is limited to 32767 fields
- stack frame is limited to 127 field elements (16-bit)
- switch statement is limited to 65536 cases
- class initialization (`<cinit>`-method) is limited to generic data types or array's of generic data types







# How fits a JCVM on a Chip Card ?




Java™ Source Code

Java Compiler (J++, Symantec ...)

Java™ Class File

Off-Card VM Converter

Card Class File (CAP)

Java Card

- The Java™ source will be converted to class files with standard tools
- Input of the JCVM are class files, containing byte code
- All static work of the JVM is done out of the card (Step 1 - 3, how described in the SUN Specification to the JVM)
- A new simplified and smaller card class file (CAP-Format) is generated
- The CAP-file with the applet is loaded into the card
- The apple will be interpreted on the chip card

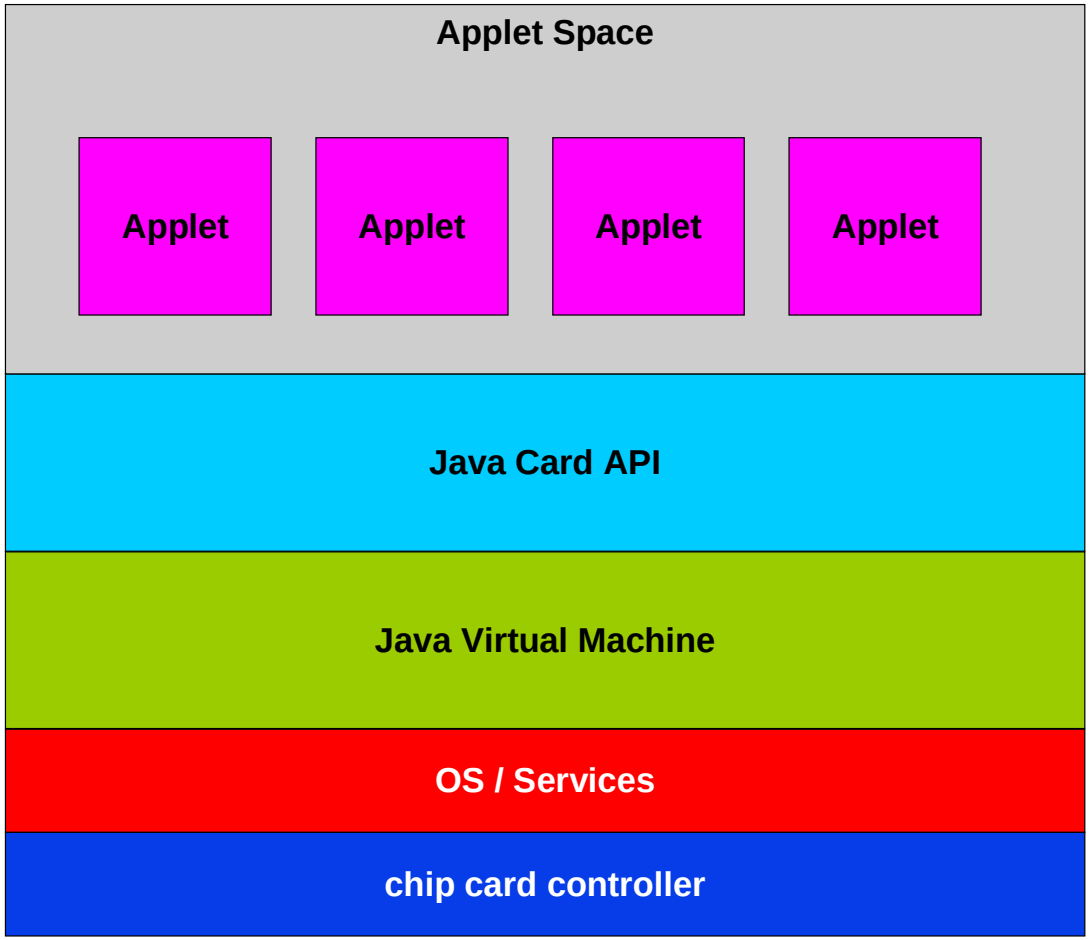


Giesecke & Devrient



Sun microsystems

# Java Card : System Overview



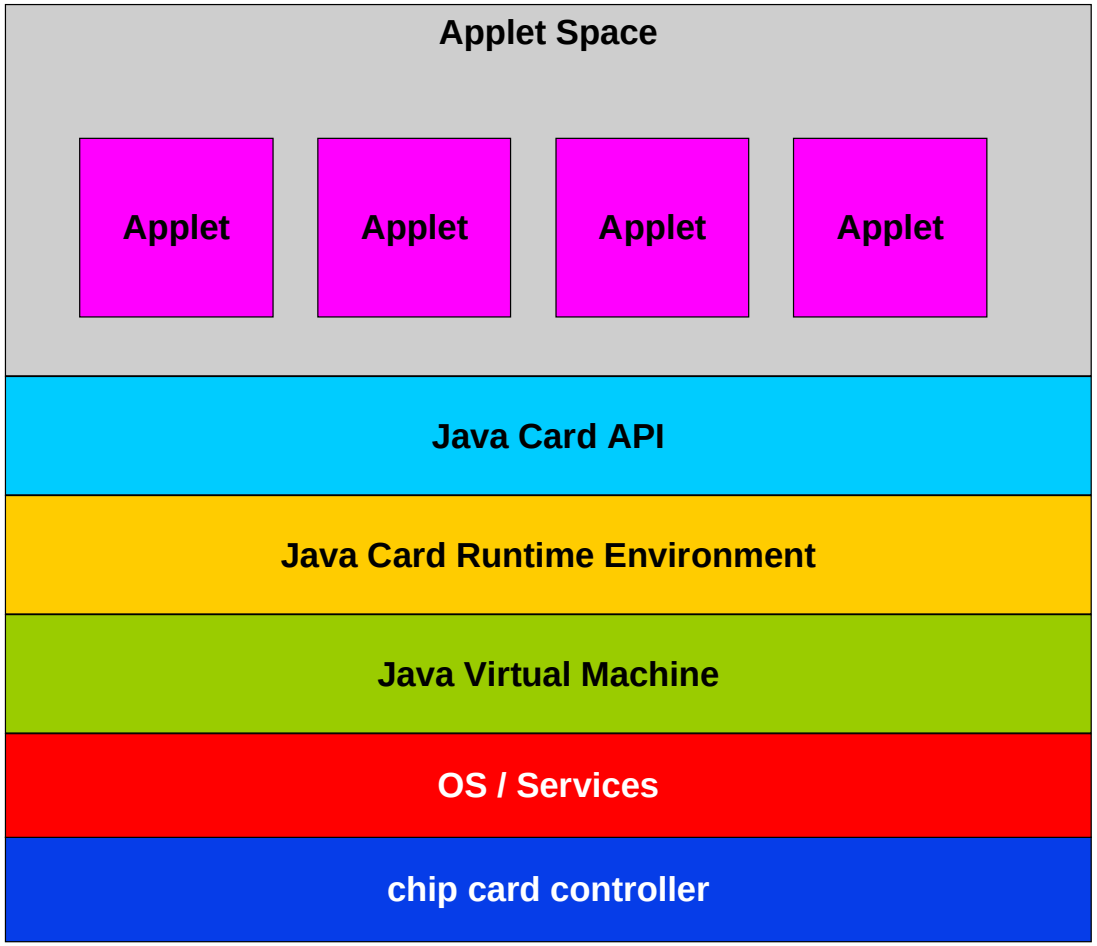



Giesecke & Devrient



Sun microsystems

# Java Card : location of Runtime Environment






Giesecke & Devrient



Sun microsystems

# What is with the Java Card OS


**OS / Services**

- Not a real Operating System but
  - EEPROM service routines
  - Cryptography
  - I/O services
  - memory
  - checksums
  - etc.

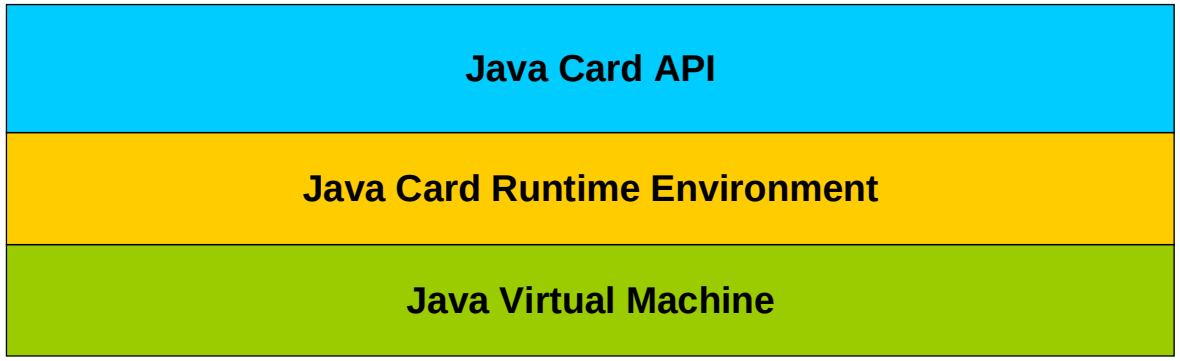


Giesecke & Devrient



Sun microsystems

# The Java Card System in a more deep view




- A full Java System with something special
  - transient array's acting as a kind of auto clear objects, triggered by special outside events
  - a smart card specific event model
  - smart card protocol specific behavior
  - transaction and atomicity
  - loading of packages not of single classes
- specification of the system ensures application interoperability



Giesecke & Devrient

# The Java Card Virtual Machine


**Java Virtual Machine**

- A full Virtual Machine but a subset
  - the VM life cycle is equivalent to the cards one
  - `char`, `double`, `float`, `long`, `string` is not supported
  - access flags `volatile` and `transient` is not supported
  - `synchronized` and `native` is not supported
  - `int` is handled as to 2 stack elements like the `long` data type in the standard VM



Giesecke & Devrient



# The Java Card Runtime Environment


## Java Card Runtime Environment

- Official definition: The JCRE consists of the JCVM, the core classes in the
  JC-API and a specific defined behavior for this Java Environment
- the defined behavior of JCVM from Applet perspective
  - differentiate Java Card Applet over a context concept
  - implements a firewall system to protected the applet context
  - Object access across contexts over a sharing concept
  - applet selection and activation control
  - initiate the applet installation process
- it is possible for a better understanding to see the JCRE as an layer between the Interpreter and the API, which realize the specified behavior



Giesecke & Devrient



Sun microsystems

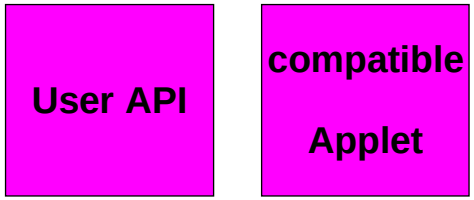
# The Java Card Application Programmer Interface


**Java Card API**

- Differentiation in 4 packages
- package `java.lang` (contains only Exception classes)
- package `javacard.framework`
  - class Applet
  - smart card specific helper classes
  - smart card event class APDU
  - JCRE support classes
- packages `javacard.crypto` and `javacard.security` (implementing access to cryptography regarding to export license issues)



# The Application or Package Format, CAP File




- To get a second level of interoperability on different Java Card platforms, there is an Converted Applet (CAP) File Format defined
- A CAP file can contain only an API, or an API and one or multiple Applets, or one or multiple Applets
- All CAP Files components are contained in a JAR-File
- There is a Off-Card verifiable constant pool component available
- There is no standardized possibility for On-Card Verification



Giesecke & Devrient



Sun microsystems

# Java Card Details: What is a Java Card Applet ?


- Any valid Java Card Applet have to be derived from the class **Applet** from package **javacard.framework**

- 
```
public class Applet extends Object {
  //
  // methods which called by the application
  //
  protected final void    register();
  protected final void    register(<specific AID>);
  public boolean          select();
  public void             deselect();
  protected final boolean selectingApplet();
  //
  // methods which are called by JCRE
  //
  public static void install(<install data>);
  public boolean     process(APDU theApdu);
  public Shareable   getShareableInterfaceObject(<data>);
}
```



Giesecke & Devrient



Sun microsystems

# Java Card Details: Applet Installation


- The JCRE calls the static class **Applet** method **install()**

- The Applet performs a self-instantiation, and register itself

- ```
  class MyApplet extends Applet {
    MyApplet(byte[] myData) {
      ... // do some initialization / personalitation
      register();
    };

    public static void install(byte bArray[], ... ) {
      new MyApplet(bArray);
    }

    ... // further Applet code
  }
  ```



Giesecke & Devrient



Sun microsystems
  ```

# Java Card Details: Applet Selection / Activation


- Applets are identified over their specific in maximum 16 bytes long AID (ISO/IEC7816-4 defined Application Identifier)
- The `class AID` encapsulate the Java Card Applet AID
- Applets are selected via the ISO/IEC7816-4 “SELECT BY AID” command
- If the JCRE detects a valid “SELECT BY AID” command, the previous (if not the first select) Applet is notified via the `deselect()` method and the new Applet is notified via the `select()` method from class `Applet`
- 
```
class MyApplet extends Applet {
  public boolean select() {
    ... // do some checking
    if (ok) return true;  // applet is prepared for selection
    else    return false; // refuse applet selection
  }
}
```



Giesecke & Devrient



# Java Card Details: Event handling


- The Java Card Event is the APDU
- All APDU's are routed directly to the received applet, only the "SELECT BY AID" is processed by the JCRE !
- The class **APDU** encapsulate the smart card IO buffer
- 
```
class MyApplet extends Applet {
  public void process(APDU apdu)
  {
    apdu.setIncomingAndReceive();
    if (selectingApplet()) {
      ... // select process, produce FCI f.Instance
    }
    else {
      ... // analyze APDU, perform action and produce the
answer
    }
    // on normal return SW1/SW2 = 0x9000 is automatically
produced
  };
}
```

# Java Card Details: Transient and Persistent Objects


- The life time of the smart card VM is equal to the physical life time of the card
- Every instance which will be created life's forever (no garbage collection)
- The Name Persistent object is closer to the persistent memory technology, than to the OO-technology persistence
- Transient memory is available as array's of generic data types, provided by special factories in the `JCSystem` class
- The transient array's are more a kind of auto-clear objects than real OO-technology transience
- Transient contents can be auto cleared on deselect or on card reset



# Java Card details: Transaction and Atomicity


- In contradiction to other Virtual Machines, a Java Card can lose the power supply to any point of time
- Every write access to class or instance fields are defined atomic, this means in fact a transaction for every putfield / putstatic byte code is necessary
- To keep user defined data groups under transaction a switch on and switch off transaction-method in `JCSystem` class is defined
- Transaction can not be nested, only one level is allowed







# Java Card Details : JCRE and the firewall concept (1)


- Every Applet on a card got it's one environment or applet context
- There is a system context, the so called JCRE context
- The JCRE is privileged context, which managing every applet context
- The JCRE context is able to call any other context, but no Applet context is normally able to call the JCRE context
- Objects are created in their applet context and all objects are only accessible normally within their own applet context



# Java Card Details : JCRE and the firewall concept (2)


- Instances of classes which are implementing any from **Shareable** derived interface can be invoked from any other applet context
- All instantiated classes (objects) are under the control of the firewall and misuse generates an access violation exception
- JCRE Entry Point Objects are special JCRE objects which can be invoked from any other applet context. These are the gateways for requesting privileged system functionality (f.I. sharing request)



# Java Card Details : Global Data


- Global Data are not owned by any runtime context
- Classes are not owned by a specific context, so there are global
- Generic class attributes and methods (f.l. `static short`) are accessible from any application
- There are no static class instances allowed
- Array's of generic data as class instance are allowed and defined as well like global (without a specific context)







# Java Card Details: Sharing


- The controlled breakthrough in the firewall for interapplet communication
- There is a interface **`Shareable`**, only references of instances which implements from **`Shareable`** derived interfaces are passed through the firewall
- So only methods and stack parameter can be passed to an other applet context
- Objects from one context cannot be shared directly to another context
- To call an other applets method
  - invoke **`JCSystem.getAppletShareableInterfaceObject()`** method with the known AID from the applet we request to involve
  - **`JCRE`** is switching the context and calls the **`getShareableInterfaceObject()`** method from the other applet
  - The other applet can now pass the reference to an object which implements an from **`Shareable`** derived interface
  - The **`JCRE`** is switching back the context and the reference is returned



Giesecke & Devrient



Sun microsystems

# Java Card Details: Sharing, an example


```
interface MyShare extends Shareable {
  public void DoSomething()
}
```

```
class MyApplet extends Applet implements MyShare {
  public void DoSomething() {
    ... // do something
  }

  public Shareable getShareableInterfaceObject(…) {
    return this; // we implement a Sharable derived interface
  }
}
```

```
class AnOtherApplet extends Applet {
  public void process(...) {
    ... // something is going on here
    // now get the help from MyApplet
    MyShare otherApp = getAppletShareableInterfaceObject(otherAID, 1);
    // now we invoke the other Applet
    if (otherApp != null) otherApp.DoSomething();
  }
```



Giesecke & Devrient



Sun microsystems

# Java Card Details: File System on a Smart Card


- With a plain Java Virtual Machine on a Smart Card, the way how data are stored becomes total different from the way how it was before
- Mainly all existing Smart Card Standards rely on a Operating System which are build around a hierarchical file system
- The Java Card is a paradigm change like the change from relational data bases to object oriented data bases in the client server world
- In a Java Card there is no hierarchical file system is not included by default
- An emulation of an exiting file specification used in the context of an application is no problem in the Java Card world
- The former defined File System API in Java Card was just a framework for implementing an emulation.
  It can be easily done by the developer self for every application where it is needed



# Java Card API: class Applet - main methods


- Applets are classes derived from **`javacard.framework.Applet`**
- Fundamental applet entry points: **`install`**, **`select`**, **`process`**, **`deselect`**
  - These methods are called by the JCRE (Java Card Runtime Environment)
  - Methods that really have to be implemented: **`install`** and **`process`**
    (the other methods can use the default implementation in `javacard.framework.Applet`)
- JCRE operation after receiving an APDU
  - Applet selection APDU (select by AID of existing applet)
    - `CurrentApplet.deselect ()`
    - `NewApplet.select ()`
    - `NewApplet.process ()`
  - Rest of APDUs
    - `CurrentApplet.process ()`



Giesecke & Devrient



Sun microsystems

# Java Card API: class Applet - description (1)


- **`public static void install (byte[] data, short offset, byte len) throws ISOException`**
  - Called by JCRE: after the applet has been loaded. No applet objects exist at this moment
  - Goals of install method:
    - to create and initialize the applet object
    - to register the applet within the JCRE, so that it becomes selectable
  - Parameters: personalization data - applet can be personalized at installation time
- **`public boolean select ()`**
  - Called by JCRE: when a "select by AID" APDU is received, containing this Applet's AID
  - Goals of select method:
    - allow/disallow selection of applet, depending on applet's internal state
  - Upon successful selection, the JCRE calls the `process()` method, so that the applet can send response data (f.e. an FCI)



Giesecke & Devrient



Sun microsystems

# Java Card API: class Applet - description (2)


- **`public void process (APDU apdu) throws ISOException`**
  - Called by JCRE:
    - whenever an APDU is received and this is the selected applet
    - when applet is being selected (APDU is "select by AID")
  - The applet must interpret the command and send a response accordingly
  - This is the most important entry point for a running applet
- **`public void deselect ()`**
  - Called by the JCRE: to inform this currently selected applet that another (or even the same) applet will be selected
  - The applet should override this method if it must do any cleanup before another applet is selected
  - Not called upon card reset!



Giesecke & Devrient



Sun microsystems

# Java Card API: APDU handling - Basics


- Class **`javacard.framework.APDU`** handles communication between card and terminal, format according to ISO 7816-4
- Everything starts in applet: **`process (APDU)`**
  - Only header (first 5 bytes) is available in the APDU buffer at this point
- SW return is handled by JCRE
  - ’9000’ is sent automatically upon return from `process`
  - other SW are sent whenever an `ISOException` is thrown
- (T=0, T=1)? Don’t worry, be happy
  - the Javacard API hides the details of the underlying transport protocol
- APDU buffer is a global array, owned by JCRE
  - all applets can access it without running into security/firewall problems
  - it is cleared upon applet de/selection, so that applets cannot spy on other applets’ command/response data



Giesecke & Devrient



Sun microsystems

# Java Card API: APDU handling - the simple API


- **`byte[] getBuffer ()`**
  - Obtain a reference to manipulate APDU buffer
  - Structure of first bytes in buffer = CLA | INS | P1 | P2 | P3
- The simple methods: when data fits in buffer
  - **`short setIncomingAndReceive () throws APDUException`**
    - Sets the transfer direction to be inbound and reads data into the buffer at offset 5
    - Gets as many bytes as will fit without overflowing the APDU buffer
    - Returns number of bytes read
  - **`void setOutgoingAndSend (short offset, short len) throws APDUException`**
    - Most efficient way to send a short response which fits in the APDU buffer
    - Send *`len`* bytes response from APDU buffer at specified *`offset`*
    - If Le is less than *`len`*, `ISOException` is thrown (SW='6Cxx')
    - No other APDU send methods can be invoked
    - APDU buffer must not be altered (actual transmission may happen on return from applet)



# Java Card API: APDU handling - other API calls


- Receiving longer command data: repeated calls to `receiveBytes`
  - **`short receiveBytes (short offset) throws APDUException`**
- Sending longer command data:
  - **`short setOutgoing ()`**
  - **`void setOutgoingLength (short len)`**
  - **`void sendBytes (short offset, short len) throws APDUException`**
  - **`void sendBytesLong (byte[] outData, short offset, short len) throws APDUException`**
- Requesting for extra processing time
  - **`void waitExtension ()`**



# Java Card API: class JCSystem - transactions


- Class **`javacard.framework.JCSystem`**
- One transaction is a set of operations that is handled atomically
  - Only persistent data is affected by transactions
- Transactions are restricted by size of commit buffer
  - **`short getUnusedCommitCapacity ()`**
- API methods for transactions:
  - **`void beginTransaction ()`**
    - begin a phase of conditional updates
  - **`void commitTransaction ()`**
    - end transaction confirming all conditional updates
  - **`void abortTransaction ()`**
    - end transaction undoing all conditional updates



Giesecke & Devrient



Sun microsystems

# Java Card API: class JCSystem - transient data


- Good for temporary data need not be persistent across sessions (f.I. Session Key)
- Transient data is stored in RAM - not in EEPROM
  - Write cycles are much faster
- Only transient arrays can be allocated
  - `boolean[] makeTransientBooleanArray (short size, byte event)`
  - `byte[] makeTransientByteArray (short size, byte event)`
  - `short[] makeTransientShortArray (short size, byte event)`
- Contents of transient data are cleared to default value (zero or false) depending on event:
  - `CLEAR_ON_RESET`
  - `CLEAR_ON_DESELECT`
- Transient data is not affected by transactions



Giesecke & Devrient



Sun microsystems

# Java Card API: class Util - array operations


- Class **`javacard.framework.Util`**
- Utilities to ease and speed up work with arrays:
  - **`byte arrayCompare (byte src[], short srcOff, byte dest[], short destOff, short length)`**
    - returns **0** if identical, **-1** if source *smaller*, **1** if source *greater*
  - **`short arrayCopy (byte src[], short srcOff, byte dest[], short destOff, short length)`**
    - copy array atomically
  - **`short arrayCopyNonAtomic (byte src[], short srcOff, byte dest[], short destOff, short length)`**
    - copy array non-atomically
  - **`void arrayFillNonAtomic (byte bArray[], short offset, short length, byte value)`**
    - fill array with one same byte value non-atomically



Giesecke & Devrient



Sun microsystems

# Java Card API: Exceptions


- Exceptions are the standard error handling approach in Java
- Exceptions in Javacard are classes derived from:
  - (checked) **`javacard.framework.CardException`**
  - (unchecked) **`javacard.framework.CardRuntimeException`**
- Exception objects should not be dynamically allocated!
  - A Javacard doesn't implement a garbage collector: space occupied by new created objects is never recovered
  - User-defined exception objects should only have one instance, reused using static method:

```
static void throwIt (short reason)
```

- ISOException: a standard class to handle SW conditions (runtime exceptions)
  - Whenever a situation occurs where a SW should be returned, use:
    - **`ISOException.throwIt (swCode);`**



# Java Card API: class PIN


- Class **`javacard.framework.OwnerPIN`** implements PIN functionality
- Some of the most important PIN methods are:
  - **`boolean check (byte[] pin, short offset, byte length)`**
    - Verifies the parameter against the internal PIN value
  - **`byte getTriesRemaining ()`**
    - Get the number of times that an incorrect PIN can still be presented before being blocked
  - **`boolean isValidated ()`**
    - True if a valid PIN has been `check`'ed during this session...
  - **`void update (byte[] pin, short offset, byte length)`**
    - Set a new value for PIN and reset the PIN try counter to the value of the PIN try limit.



Giesecke & Devrient



Sun microsystems

# Some words to interoperability


- Interoperability has more layers
  - on Application level, that's using common API's
  - on platform level, that's using a common CAP file
- Interoperability can not have optional elements, that's in deed in contradiction to SUN's actual specification
- Interoperable applications can not rely on proprietary API's
- define Wrapper classes whenever using optional or proprietary elements







# Java Card at G&D : Sm@rtCafé V1.1


- based on Java Card API 2.1 standard
- additional G&D Crypto-API (DES, DES3, RSA) with helpful crypto services and easier exportability
- ~12K Applet Space, 260 Byte APDU, 300 Byte Stack
  275 Byte Transient Space, 336 Transaction Space
- with the G&D Toolkit Sm@rtCafé Professional
- card available 12/98, toolkit with card and reader 01/98



Giesecke & Devrient



Sun microsystems

# What are the customers benefits ?


- Interoperability (Write Once run everywhere) !
- Well suited chip card applications
- Avoid expensive mask development
- Very short time to market
- New possibilities to use chip cards in the IT



# Once again, what is Java™ Card ?


- A standard of SUN™
- A new way of programming smart cards in high level language
  -> Java
- The customer possibility to write his own chip card application
  -> fast time to market
- Run the same application on every chip card vendors Java Card
  -> interoperable / second source
- Running an interpreter
  -> subset of the Java Virtual Machine
- Secure Applications protected against other Applications
  -> firewall



Giesecke & Devrient



Sun microsystems

# Glossary

- JVM Java Virtual Machine
- JCVM Java Card Virtual Machine
- JC-API Java Card API (Application Programmer Interface)
- AID Application Identifier
- APDU Application Protocol Data Unit
- TPDU Transport Protocol Data Unit
- ATR Answer To Reset