



JavaCard

Ein Workshop für NXP Semiconductors Germany GmbH



Cegos Group

inspire
qualify
change



Inhalts- verzeichnis



Die Java Plattform



Best Practices



Java Programmierung



Object-oriented Programming



Exkurs: Eclipse



Java Card Programmierung



Die Java Plattform



Überblick



Die Java Card Virtual Machine



Überblick

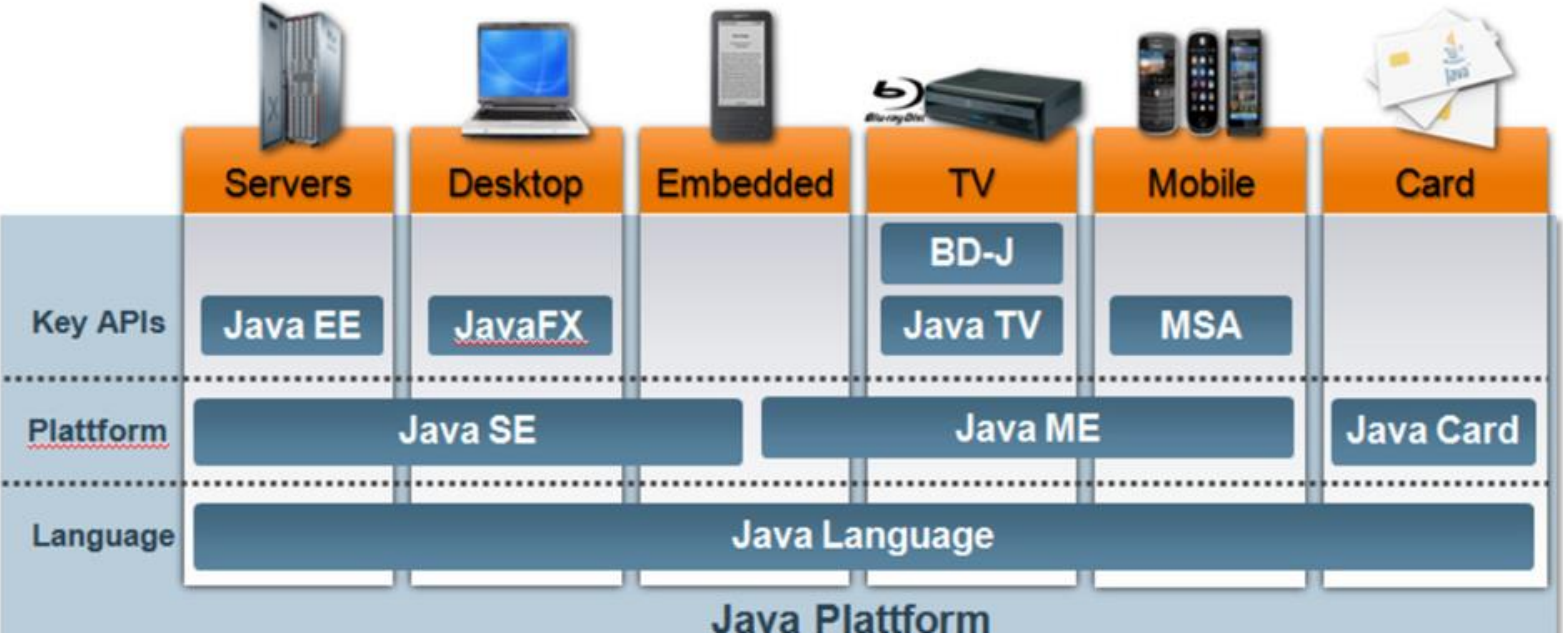


Java Development Kit und Java Runtime

- Das Java Runtime Environment (JRE) besteht aus:
 - Der Java Virtual Machine (JVM)
 - Programm `java`
 - Der Bytecode-Spezifikation
 - Der Befehlssatz der JVM
- Die Programmiersprache Java besteht aus
 - Der Syntax
 - Dem Compiler
 - `javac`
 - Den Klassenbibliotheken



Java Spezifikationen





Bytecode

- Wikipedia.org
 - *"Bytecode, also known as p-code (portable code), is a form of instruction set designed for efficient execution by a software interpreter. Unlike human-readable source code, bytecodes are compact numeric codes, constants, and references (normally numeric addresses) which encode the result of parsing and semantic analysis of things like type, scope, and nesting depths of program objects. They therefore allow much better performance than direct interpretation of source code."*
- The Java Virtual Machine implements security features to detect corrupted bytecode
 - Bytecode format
 - Strict typing
 - Variables and data structures must be initialized before first use
 - No direct heap access, no pointer manipulation
- Bytecode is compiled from source code and stored in a class file
 - A single class is the smallest possible Java application
 - More complex applications are distributed using a zip format, the Java Archive (JAR)



Java Card

- Eine Umsetzung der Java-Plattform auf Systemen mit (wirklich) beschränkten Ressourcen
 - Deutlich Beschränkungen der Java Virtual Machine
 - Eingeschränkter Befehlssatz
 - Einige Features einer JVM können nicht auf der Karte abgebildet werden
 - Bytecode-Verifier
 - Nur optionale Garbage Collection
 - Kein Update möglich, die JVM ist fest im ROM eingebrannt
 - Angepasstes Memory-Layout
 - RAM mit sehr beschränkter Größe
 - Persistentes RAM (EEPROM) mit beschränkter Größe und limitierter Anzahl von Lese/Schreibzyklen
 - Angepasstes Ausführungs-Modell
 - Ausschließlich kurzfristige Stromversorgung vorhanden
 - Ein Wegbrechen der Stromverbindung und damit kompletter Abbruch eines Vorgangs ist jederzeit möglich



Java Card

- Smart cards enthalten eine Embedded Integrated Circuit (IC)
- Low-Level-Kommunikation über APDUs
 - Application Protocol Data Units
 - Ein strukturiertes Daten-Array mit Steuerzeichen und Nutzdaten
 - Das Terminal sendet eine APDU zur Karte, diese schickt eine Antwort-APDU



Die Java Card Virtual Machine



Grundprinzip

JVM



Aufgaben einer Java Virtual Machine

- Interpreter für Bytecode
- Sicherheitsmechanismen
 - Bytecode-Verifier
 - Korrektheit des Bytecode-Formats
 - Übereinstimmung der Typen bei Zuweisungsoperationen und Aufrufen von Routinen
 - Kein Zugriff auf nicht-initialisierte Speicherbereiche oder Variablen
 - Keinerlei direkter Speicherzugriff
 - Sandbox-Konzept
 - Anwendungen sind komplett gekapselt
 - Umgang mit Ausnahmesituationen
 - Exceptions
 - Kein Programmabsturz oder ähnliches
- Auflösen und dynamisches Binden von Programm-Bibliotheken
 - Classloader



Wie soll eine Card
diese Aufgaben
übernehmen?





Gar nicht...

- Die Java Card Virtual Machine wird konzeptuell zweigeteilt:
- Der reine Bytecode-Interpreter befindet sich auf der Karte
- Alle anderen Aufgaben werden im Rahmen eines Build-Prozesses auf einer mächtigen Desktop-Maschine ausgeführt
 - Dieser erzeugt ein CAP-File, das kompakt auf der Karte installiert wird



Vorsicht

- Java Card ist beim Thema „Deployment und Management“ komplett unterspezifiziert!
 - Kein verbindlicher Security-Mechanismus, der für die Installation einen Schutz vor nicht-vertrauenswürdigen Anwendungen bietet
 - CAP-Files können manipuliert werden, um im Endeffekt unverifizierte Anweisungen zu enthalten
- Damit sind potenzielle Angriffsvektoren gegeben
 - Installation einer Anwendung auf einer Karte, um Schwachstellen der Implementierung eines bestimmten Kartentyps zu erkennen
 - Um daraus weitere Angriffe zu konzipieren
 - Nachträgliche oder zusätzliche Installation einer Spionage-Anwendung auf einer Karte
 - Java Card erlaubt die Installation mehrerer Anwendungen auf eine Karte
 - Moderne Karten ermöglichen teilweise sogar eine Remote Installation!



GlobalPlatform.org

- Hauptfokus liegt auf dem Thema Anwendungs-Management
 - Hierzu definiert die Global Platform Spezifikationen und stellt Zertifizierungen zur Verfügung



Features

- GP 2.2.1 card specification, core
 - issuer centric or simple model
 - delegated management
 - authorized management
 - verification authority (GP 2.1.1 controlling authority)
 - Data Authentication Pattern (DAP)
 - Secure Channel Protocol 02 (SCP), pseudo random, C-MAC, C-ENC, R-MAC, R-ENC
 - all the privileges are supported
- Amendment A - Confidential Card Content Management (C3M)
 - see implementation details in UICC configuration and Amendment E
- Amendment C - Contactless Services
 - every protocol but Felica is supported
 - additionally MIFARE Classic and DESFIRE is supported
 - HCI notifications are supported
- Amendment D - Secure Channel Protocol 03 (AES)
 - only AES-128
 - all options are supported (random)
- Amendment E - Security Upgrade
 - SHA-256 and EC-256
 - C3M scenario #3
 - UICC configuration
- scenarios #1, #2A and #2B
 - SCP 80 and 81 is not supported (ETSI)



API der GlobalPlatform

All Classes

Application

Authority

AuthoritySignature

CVM

GlobalService

GPRRegistryEntry

GPSystem

HTTPAdministration

HTTPReportListener

Personalization

SecureChannel

SecureChannelx

SecureChannelx2

GLOBALPLATFORM™

PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package org.globalplatform

Provides a framework of classes and interfaces related to core services defined for smart cards based on GlobalPlatform specifications.

See: Description

Interface Summary

Interface	Description
Application	This interface defines a method through which an Application may forward input data to another Application.
Authority	This interface allows performing operations such as recovering a cryptographic key or signing data.
AuthoritySignature	This interface allows generating a signature over the input data provided by an Application.
CVM	This interface defines basic Cardholder Verification Method services (e.g.
GlobalService	This interface allows requesting a Global Services Application for a Shareable Interface Object (SIO) providing the actual service.
GPRRegistryEntry	This interface allows querying and potentially modifying the registry data of an Application registered within the GlobalPlatform Registry.
HTTPAdministration	This interface defines a method to trigger a new HTTP administration session.



JavaCard Open Platform - JCOP

- Eine Vereinigung von JavaCard und GlobalPlatform mit eigenen proprietären Erweiterungen



Java Programmierung



Language Basics



Expressions and Statements



Methods



Complex Types and References



Language Basics



A Word of Warning

- Java is a complex language
 - You need to write quite a lot of code to run even the simplest application
 - Developer tools will help a lot
- But:
 - to learn Java from the beginning you have to put aside some "magic" or unknown code sequences
 - What is a package and a class?
 - What's the meaning of "public static"
 - Why do I have to code `System.out.println("Hello")` to write output to the console?
- Keep cool!
 - everything will be explained during this training
 - one step after the other...



Java Sourcecode

- Every Java program is contained in a simple text file
 - Unicode is supported
- Whitespaces can be used as needed
 - some conventions do exist, but feel free to format your code as you like
- Developer tools like Eclipse or IntelliJ provide some helpful features
 - Syntax highlighting
 - Code formatter
 - Line numbering
 - Folding
 - ...



Comments

- Comments
 - line comment
`//`
 - block comment
`/*`
`Block-comment`
`*/`
 - javadoc comment
`/**`
`Javadoc-comment`
`*/`



Identifiers

- A lot of things in Java are identified using a unique identifier
 - e.g. variables
- Identifiers may contain
 - letters
 - numbers
 - underscore
- Naming conventions do exist
 - Camel case
 - `myFirstIdentifier`
 - Using capital letters hints a special usage
 - Program names start with capitals
 - Constant values contain only capital letters and an underscore for readability
 - `MY_FIRST_CONSTANT`



Basic Data Types: Java

- **Integer numbers**
 - `byte`, `Byte`
 - `short`, `Short`
 - `int` , `Integer`
 - Used as default
 - `long`, `Long`
- **Floating-point numbers**
 - `float`, `Float`
 - `double`, `Double`
 - Use this type as default
- **Strings**
 - `char`, `Character`
 - `String`
 - unlimited range
- **Logical**
 - `boolean`, `Boolean`



Basic Data Types: Java Card

- **Integer numbers**
 - `byte`, `Byte`
 - `short`, `Short`
 - `int`, `Integer`
 - not supported as part of specification, some Card manufacturers and frameworks may provide `Integer`
 - strange: still the default
- **Strings**
 - a kind of String support is added to JavaCard 3.1
- **Logical**
 - `boolean`, `Boolean`



Literals

- Numbers
 - `(byte) 4`
 - `(short) 42`
 - `4711`
 - caution: that is an Integer and probably not working
 - Hexadecimal is working:
 - `byte b = 0x20`
 - `short s = 0xAA`
- Booleans
 - `true`
 - `false`



Declaring variables

- Everything in Java is strictly typed
 - so the Java compiler is able to detect a lot of coding errors
 - "Multiply a number with a string"...
- Syntax:
 - `type identifier;`
 - `type identifier = initial value;`
- Example:
 - `byte b = 4b;`
 - `Boolean checked = true;`



Compiler checks

- A value must be assigned before usage

```
short number = 42;  
short number2;  
short result;  
short result = number; //OK  
short result = number2; //Compiler error
```

- Variables must be declared

```
number2 = 42; //Compiler error
```

- Assigned values must be of correct type

- `number2 = number; //OK`
- `number2 = false; //Compiler error`



Expressions and Statements



Operators

- Assignment

- =

- Arithmetic

- $a + b$

addition

- $a - b$

subtraction

- $a * b$

multiplication

- a / b

division

- $a \% b$

modulo division (integers only, so probably not

be supported)

- shortcuts

$a += b$

$a -= b$

$a *= b$

$a /= b$

$a \% = b$

$a++$

$++a$



Operators II

- Comparison
 - ==
 - !=
 - >
 - <
 - >=
 - <=
- Logical
 - ! not
 - & & and
 - | | or
- Bitwise
 - >>, >>>
 - <<, <<<
 - &
 - |
 - ^



Control Structures: if

▪ Syntax:

```
if (boolean expression) {  
    statement  
    statement  
}  
else {  
    statement  
    statement  
}
```



Control Structures: switch

```
▪ Syntax:
switch( Integer, Character or String) {
    case constant1: {
        statement1.1
        statement1.n
        break;
    }
    case constant2:{
        statement2.1
        statement2.n
    }
    default: {
        statementd.1
        statementd.n
    }
}
```



Loops: while, do while

- Syntax:

```
while(boolean expressions{  
    statements;  
}
```

```
do {  
    statements;  
} while(boolean expression);
```



Loops: for

- Syntax:

```
for( expression1;  expression2;  expression3 )  {  
    statement;  
    statement;  
}
```

- where

- expression1: initialization
- expression2: condition
- expression3: statement



Methods



Declaration

- Java methods are named containers containing arbitrary statements
 - other languages use "functions" or "procedures"
- Every method must be declared
 - it always has a return type
 - `boolean check`
 - if you don't need a return type use `void`
 - yes, it's lower case
 - it has a comma-separated list of typed parameters
 - `boolean check(byte b1, byte b2)`



Definition

- A method has a block containing statements and must return a suitable value

```
boolean check (byte b1, byte b2) {  
    return (b1 < b2) && (b1 > 0);  
}
```




Calling methods

- To call a method simple use it's identifier and provide arguments between parentheses
 - `check(4, 3);`
 - `check(-5, 3);`
- The returned value may be used in an assignement
 - `Boolean state = check(4, 3);`
- Method parameters will be "called by value"
 - So reassigning a parameter with another value has no effect on the calling variables



Compiler checks

- Return type and parameter types are checked
- If you use conditional statements the compiler checks every branch to return a value
- There must not be another statement after `return`



Complex Types and References



Arrays

- Arrays are fixed-sized data containers
- Declaration:
 - `byte[] data;`
- Literal:
 - `byte[] data = {6, 12, 21, 2, 1, 8};`



Loops: for each

- **Syntax:**

```
for( type element : array)  {  
    statement;  
    statement;  
}
```

- **Example:**

```
for (byte number: data){  
    //do something with number;  
}
```



Memory Managements: Variables

- Take a simple Example
- `short number1 = 1`
`short number2 = 1;`
- `so number1` and `number2` are named variables
- Both variables contain the same value, but this value will be held in memory two times:



Changing the value of Variables

- if you increment e.g. `number2` using `number2++` we'll get



Assignments using variables

- But what happens if we assign `number1` to another variable, `number3`
 - `short number3 = number1;`
- Now the inventors of Java had different possible approaches



Use an alias



Take a copy



What's correct?

- If you come up with some test applications you will favour the copy-mechanism
- To implement the test simply write a method and use parameters



We cheated!

- We used the type `short` instead of `Short`
 - these are the so called "basic Java types"
 - they work exactly as described before
 - but you will not use them on a regular basis
- Java's memory management is something completely different!
 - So we have to start all over again!
 - This time we use an array



Arrays

- Let's use an array

```
byte[] data = [1, 2, 3, 4];
```

```
byte[] data2 = data;
```



References

- A graphical representation will use "pointers":
 - technically spoken a pointer is an internal memory address
 - Java does not allow pointer arithmetic
- A reference will be copied like the basic types before
 - but the interpretation is totally different
 - a copy of a reference value points to the same object



Copying references



References and method parameters

- If you call a Java-method all parameters are references and will be copied by value
- So changing an array inside an invoked method will change the original array!



Object-oriented Programming



Object-oriented design



CLASSES AND OBJECTS



A short UML Primer



Classes and Java



Polymorphism and Overriding Methods



Static Elements



Packages



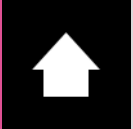
Exception Handling



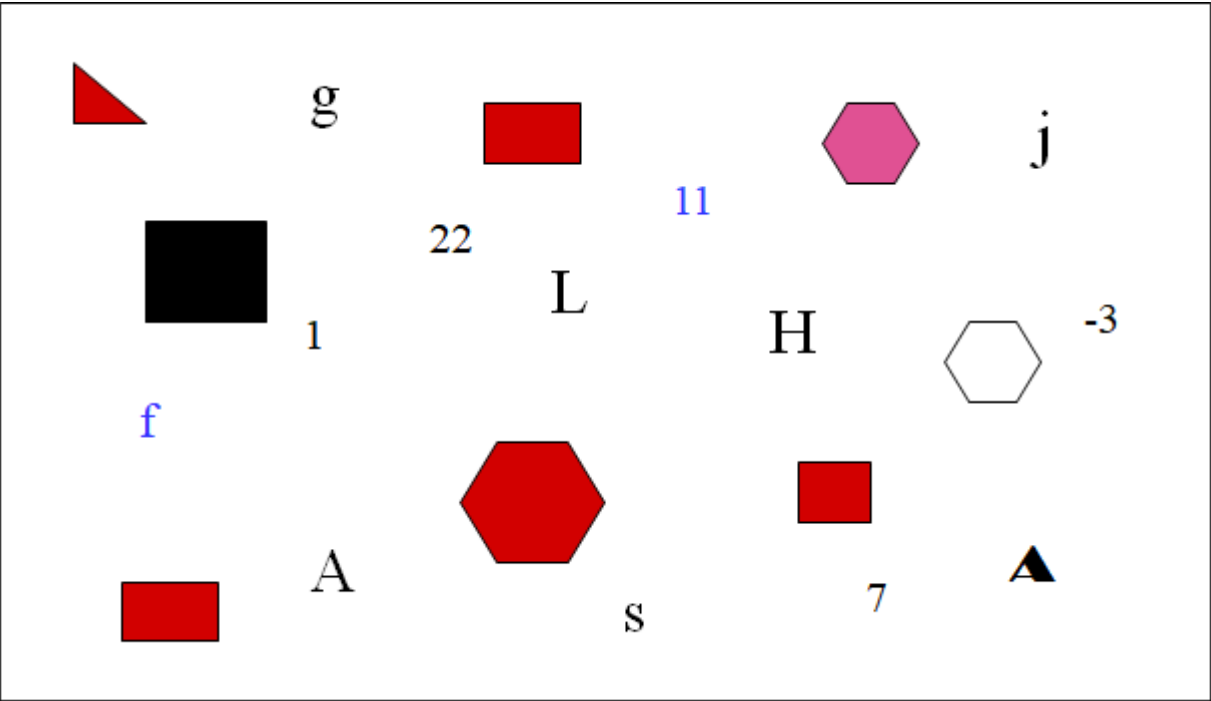
Annotations



Object-oriented design



A complex,
unordered system





Ordering the chaos

- Are there any correlations?
- What is really important?
- Are there any dependencies?



An object-oriented approach

- Classification
 - Abstraction
 - Hierarchies
-
- A quite "human" approach

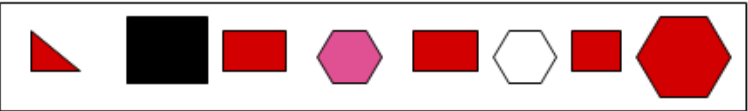


Classification

- Classification
 - Abstraction
 - Hierarchies
-
- A quite "human" approach

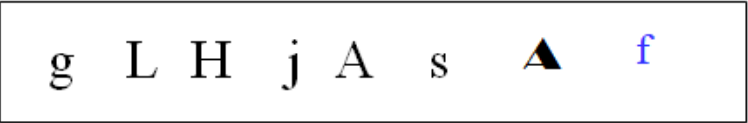


Abstractions



shapes

- color
- position
- size



letters

- character
- font
- color
- position
- size



numbers

- value
- color
- position
- size

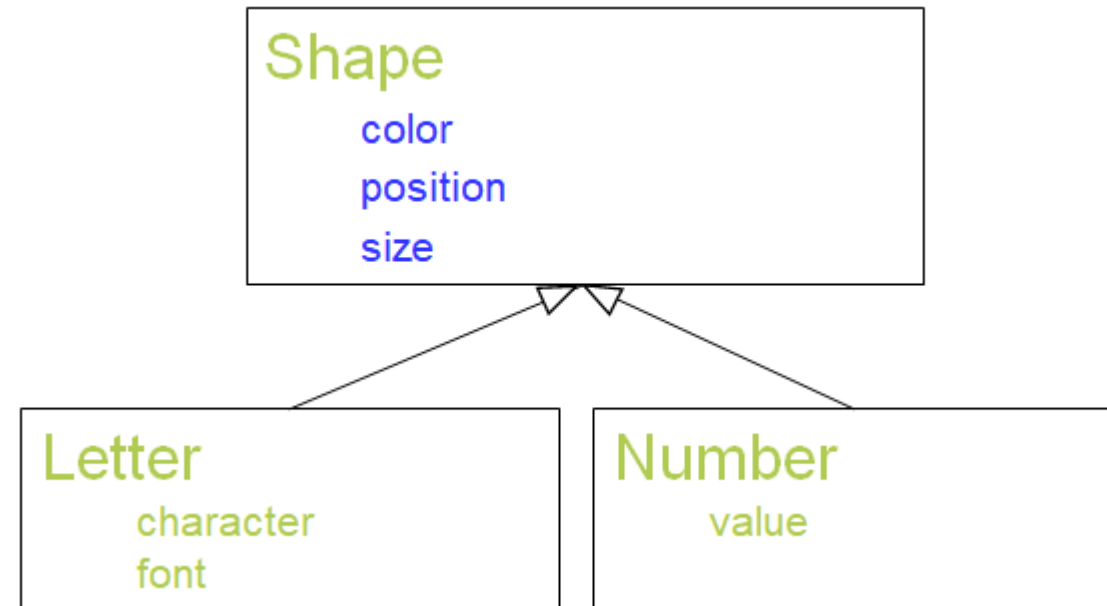


Interpretation of abstraction

- A shape
 - has a color,
 - position
 - and size,
- It has attributes
 - A complex shape is a composition of simpler types



Hierarchies





Interpretation of the hierarchy

- a letter is a shape
 - that draws a character using a font
- a number is a shape
 - that represents a value
- Letters and numbers are special shapes that inherit all attributes from a shape



CLASSES AND oBJECTS



What is an object?

- Simply everything...



Objects

- Every object has
 - a state
 - attributes/fields
 - behaviour
 - methods
- compare this with structured programming
 - attributes
 - data structures
 - methods
 - functions and procedures
- Objects are a form of abstraction:
 - an object is build using a formalized template.
 - An object is an instance constructed from a class definition



Classes

- A class describes similar objects
 - Classes are always an abstraction: in the real world all objects are unique
- A class is a template for objects



A short UML Primer



UML – the class diagram

- Unified Modelling Language is a formal language to describe classes and their dependencies
- A class diagram contains the visual representations of one or many classes



Example: class Person

Person
vorname: String nachname: String
getVorname() : String getNachname() : String setVorname(: String) : void setNachname(: String) : void getName() : String

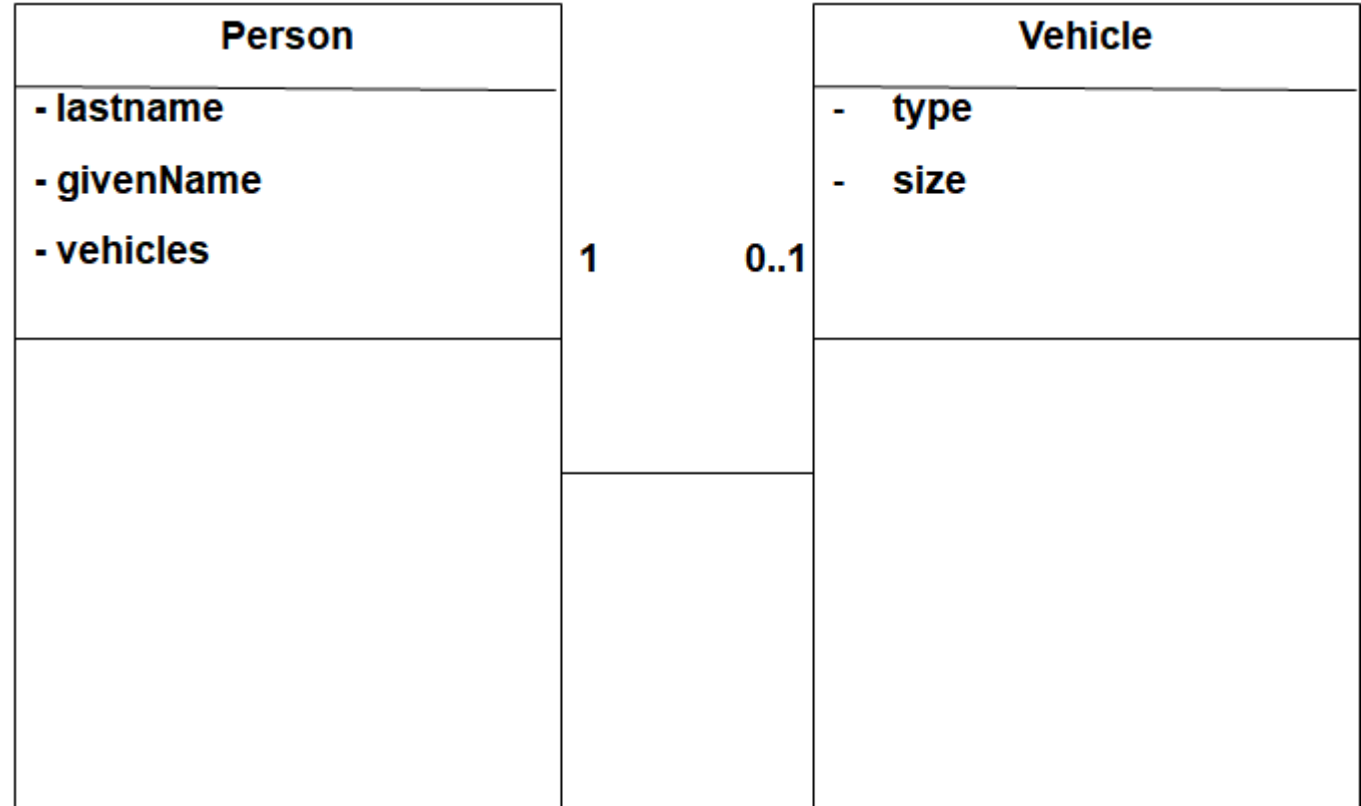


Association

- If you describe the relation of two objects with "has a" or "uses a" it will be an association
- The cardinality describes if you have a kind of one-to-one or one-to-many association



Association: Class diagram



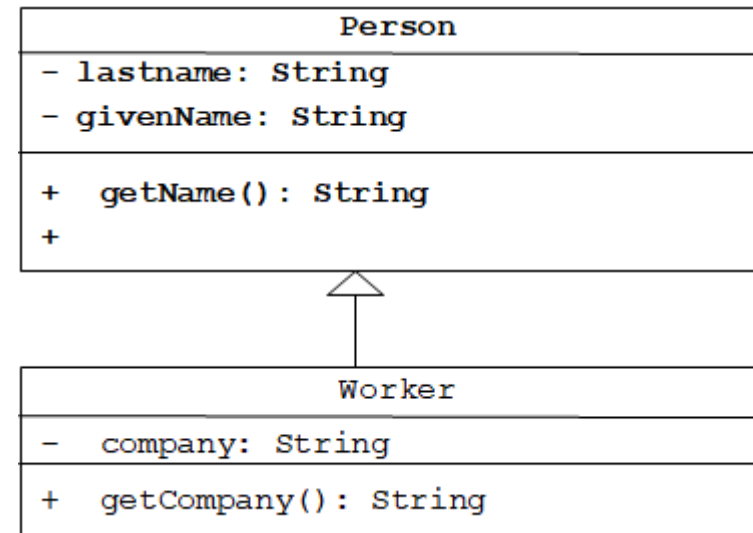


Inheritance

- something "is a" specialization of something else
 - Example: A worker is a person



Inheritance: Class Diagram





Classes and Java



Class definition

```
visibility class name {  
    // body containing  
    // attribute and method definitions  
}
```

- **Visibility:**
 - `public`
 - **access allowed**



Attributes

- `visibility type name;`
- Accessing an attribute:
 - `reference.attributename;`
- Visibility:
 - `private`
 - access is restricted to it's own class file
 - `public`
 - everyone has access (not recommended)



Methods

```
visibility return type name (type1 parameter1, type2  
parameter2, ... )  
{ method block  
  
    local variable declarations  
    and statements  
}
```

- **Calling methods:**

- `reference.methodname (param1, param2, ...);`

- **Visibility:**

- `private`
 - access is restricted to it's own class file
 - `public`
 - everyone has access (not recommended)
 - `<default>`
 - access only inside the package that contains the class
 - definition



References, Objects and Initialization

- Example:

```
Person p1;           // p1 a reference of type  
Person
```

```
p1 = new Person();  
// Construction an objects, p1 references the  
newly                      created object
```

```
Person p2 = new Person();
```

- A special reference: null

```
p1 = null;
```



Accessing attributes and methods

- Dot-operator

- Syntax:

```
reference.attributeName;
```

```
reference.methodName( );
```

- `this`

- `this` represents the context of the actual object and is used inside a class to access its own attributes and methods



Overloading methods

- The Java compiler and JVM uses the parameter list to distinguish between methods of the same name.

- Example:

```
boolean check(byte b)
{
    return b > 5;
}
```

```
boolean check (byte b1, byte b2)
{
    return b1 > b2;
}
```



Constructors

- a constructor is a special method named exactly like the containing class
- constructors may not declare a return type
- constructors are invoked only using the new operator
- Special constructors are
 - the no-arg constructor has no parameters. If no other constructor is defined the no-arg-constructor is automatically created by the compiler
 - the default constructor (`{...}`) is always invoked first
- constructors may be overloaded
- a constructor may call another overloaded constructor as the first statement

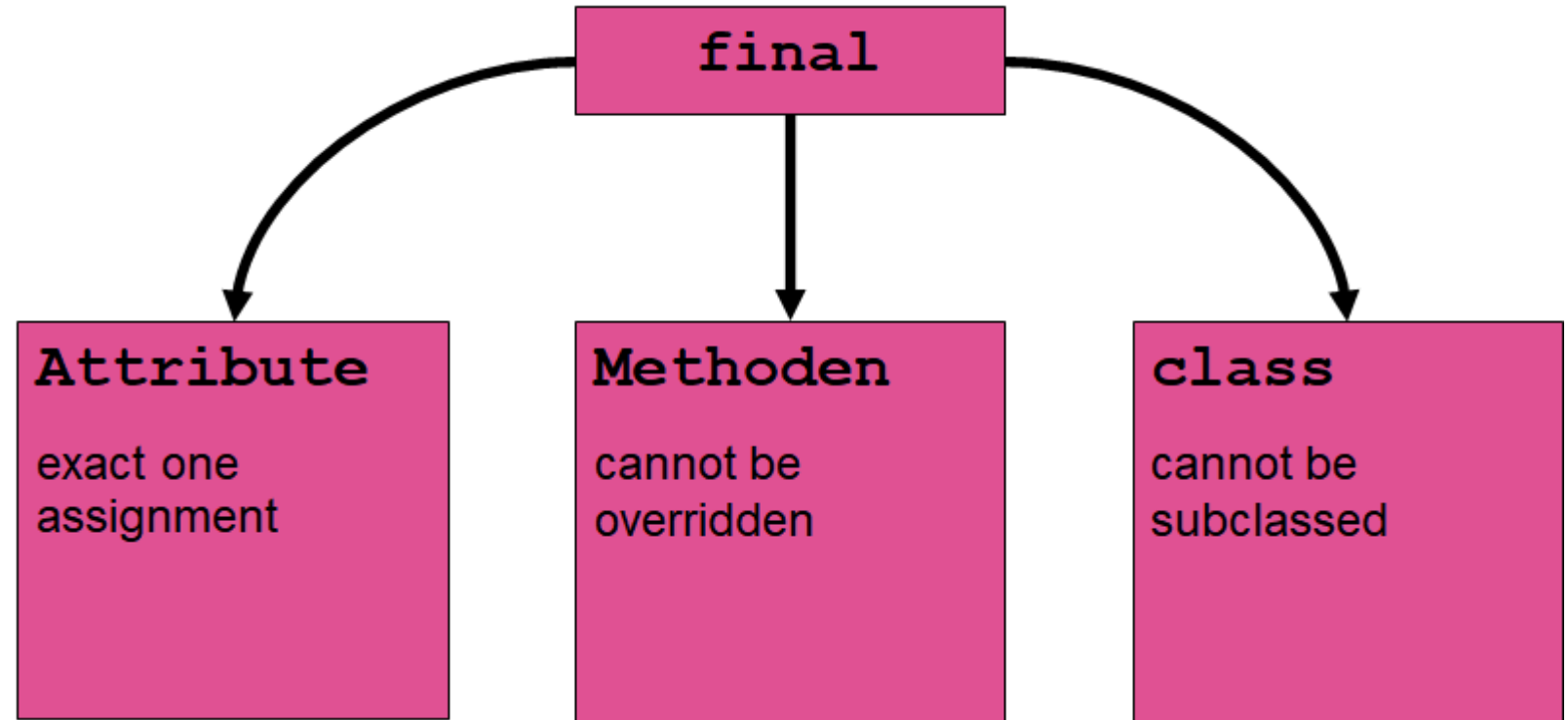


Final

- A final attribute or variable must be assigned exact one value
- Final methods cannot be overridden
- Final classes cannot be subclassed



Final





Polymorphism and Overriding Methods



Inheritance

Syntax:

```
visibility class Subclass extends Superclass{  
    //  
}
```

Example:

```
public class Worker extends Person {  
    private String company;  
  
    public String getCompany() {  
        return this.company;  
    }  
}
```



Visibility and Inheritance

- `private`
 - same class only
- `<default>`
 - all classes inside the same package
- `protected`
 - all classes inside the same package and all subclasses
- `public`
 - all classes



Inheritance and super

- Calling a superclass constructor
 - `super (parameterliste);`
 - Caution: Constructors are not inherited!
- Calling a superclass method
 - `super.method();`
- Caution: There is no such thing as a super-Hierarchy
 - `super.super.method();` not allowed



Polymorphism

- A subclass method overwrites a superclass method if it has the same signature
 - the methods signature is it's name plus parameter list
- Polymorphism
 - Overridden methods will be resolved using the inverse inheritance hierarchy.

```
Person p = new Person();  
Worker w = new Worker();  
Person p2 = new Worker(); //yes, that is possible!  
                           //a worker is a person  
p.sayHello(); // "Hello, I am a person"  
w.sayHello(); // "Hello, I am a worker"  
p2.sayHello(); // "Hello, I am a worker"
```



Static Elements



Static members

- Static members are part of the class, not the object
 - Better: Static members are part of the class object
- Syntax:

```
visibility static type name [ = value ];  
visibility static returntype methodname( parameter )  
{  
    // only static attributes are accessible here!  
    // no this!  
}
```
- static members are accessed using the class object



Static initializers

- Example:

```
// Attribute
private String lastname;
static private boolean a;
static private double b;

static {
    a = true;
    b = 1.0;
}
```



Packages



Packages and import

- Packages group classes
 - `java.lang`
 - `org.javacream.basics.oop`
- referencing classes:
 - fully qualified class names
 - `org.javacream.basics.oop.people.Person;`
 - import and shortened class names
 - import one class only
 - `import org.javacream.basics.oop.people.Person;`
 - package import
 - `import org.javacream.java.basics.oop.people.*;`



Visibility and packages

- `private`
 - same class only
- `<default>`
 - all classes inside the same package
- `protected`
 - all classes inside the same package and all subclasses
- `public`
 - all classes



Exception Handling



Exceptions

- An additional return value that signals.
 - Hardware errors
 - Programmers errors
 - Exceptional application conditions like constraint violations or illegal states
- An Exception is a return value of type `Exception`
- Exceptions hierarchy
 - Base class `Throwable`
 - Subclasses `Error` and `Exception`
 - Don't subclass or throw `Errors`!
 - `Exception` itself is the base class for
 - Checked Exceptions
 - Must be handled or declared
 - `RuntimeExceptionS`
 - May be handled or declared



Exception handling: example

```
readFile {  
    open file;  
    calculate file size;  
    allocate memory;  
    read file;  
    close file;  
}
```



"Old fashioned" Error handling

```
readFile {  
    //error code = 0;  
    //open file...  
    if (file is open) {  
        //caculate size;  
        if (size known) {  
            //allocate memory;  
            if(memory is allocated) {  
                //read file;  
                if(something is wrong)  
                    errorcode = -1;  
            } else  
                errorcode = -2;  
        } else  
            errorcode = -3;  
        close file;  
        error code = -5;  
        return error code;  
    }  
}
```



Exception keywords

- `try`
 - defines a block that may catch a checked exception
- `catch`
 - a block that handles a thrown exception
- `finally`
 - contains statements that should be executed anyway
- `throw`
 - throws an object of type `Exception`
- `throws Exception`
 - part of a method declaration listing all thrown exceptions
 - mandatory for all checked exceptions



Annotations



Meta Information

- Objects are often used inside a "container"
- The container adds some additional features
- Examples:
 - A test container creates reports (success and error)
 - A service container provides remote access
- The container needs additional information to fulfill its tasks
 - "What methods should be invoked during a test?"
 - "What kind of http-request should trigger the invocation of a Java method?"
-



Exkurs: Eclipse



Eclipse Workbench



Beispiele für Views und Editoren



Perspektiven



Views



Editoren



Die Standard-Perspektiven



Eclipse Workbench



Inhalt

- Eclipse Workbench
 - Aufbau
 - Perspektiven
 - Views
 - Editoren



Aufbau

- Aufbau der Workbench



Perspektiven



Perspektiven

- Was ist eine Perspektive?
 - Views
 - Editoren
 - Menüs
 - Toolbars
 - etc.

die zu einem Anwendungsgebiet gehören, z.B. Java, CVS o.ä.



Perspektiven

- Perspektive auswählen/öffnen



Resource Perspektive



Java Perspektive



Java Browsing Perspektive



Debug Perspektive



Views



Views

- Was ist eine View?
 - Anzeige einer Resource, z.B. Java Klasse, XML Schema, etc.
 - Wird oft als Strukturanzeige verwendet, z. B. Dateibaum, Klassenaufbau, Klassenhierarchie etc.
 - Meistens Baum- oder Tabellenansicht.
- Beispiele
 - Outline View zeigt z. B. den Strukturbaum einer Java-Klasse
 - Tasks View zeigt Fehler und noch zu erledigende Aufgaben
 - Navigator zeigt den Verzeichnisbaum der Festplatte



Navigator View



Package Navigator View



Editoren



Editoren

- Was ist ein Editor?
 - Auf eine Resource, z.B. Java-Datei, spezialisiertes Eingabeelement
 - Bietet spezielle, angepasste Editierfunktionalität, z.B. Syntax-Highlighting, Autoergänzung, Validierung etc.
- Beispiele
 - Java-Editor Editieren von Java Source-Dateien
 - plugin.xml Editieren der *plugin.xml* Datei
 - JSP-Editor Editieren von JSP-Dateien (extra Plugin!)
 - ...



Die Standard-Perspektiven



Standard Perspektiven

- Standard Perspektiven der Eclipse Umgebung
 - Resource allgemeine Projekte
 - Java Java Projekte
 - Java Browsing Browsen von Packages, Klassen und Interfaces
 - Java Type Hierarchy Anzeige der Java Typen Hierarchie
 - Debug Ausführung des Java Debuggers
 - CVS Repository Anbindung an CVS Versionsverwaltungen Exploring
 - Plugin Dev... Plugin Entwicklung
 - Team Synch... Abgleich mit Repository bei Teamarbeit



Resource Perspektive



Java Perspektive



Java Browsing Perspektive



Java Type Hierarchy

Perspektive



Debug Perspektive



CVS Repository

Exploring Perspektive



Plug-in Development Perspektive



Beispiele für Views und Editoren



Standard Views

- Views
 - Navigator
 - Outline
 - Tasks
 - Console
 - Package Explorer
 - Hierarchy
 - Ant
 - Search
 - ...



Navigator View

- Navigator



Outline View

- Outline
 - Strukturdarstellung
(z.B. Java Klasse oder
Interface)



Tasks View

- Tasks
 - Fehler, Warnungen, Infos und TODOs.
 - Doppelklick öffnet die entsprechende Resource im Editor.



Console View



Package Explorer View



Hierarchy View



Standard Editoren

Standard-Editoren:

- Text-Editor
- Java-Editor
- plugin.xml Editor
- ...



Java Editor



Java Card Programmierung



Applets



Cryptography



Aufruf des Applets



Memory Management



Buildprozess



Transactions



Applets

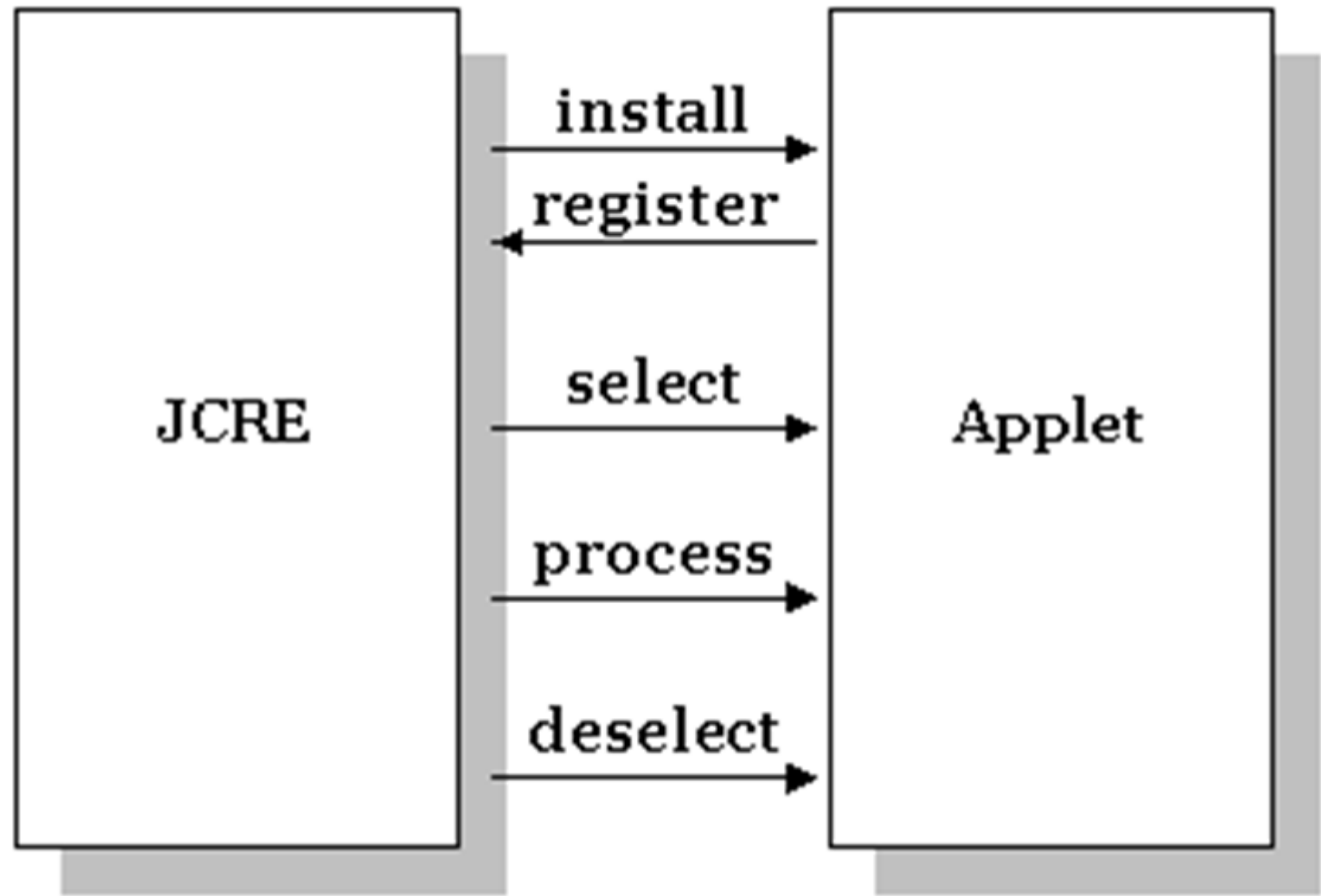


Was sind Applets?

- Ein Applet ist eine Instanz einer Klasse, die auf der Karte installiert wird
 - Programmiert wird eine Applet-Klasse
 - diese erbt von `javacard.framework.Applet`
 - Während des Installationsvorgangs wird eine Instanz dieser Klasse angelegt
 - sämtliche Attribute der Instanz liegen damit zwangsläufig im EEPROM
 - Vorsicht deshalb insbesondere bei schreibenden Zugriffen
- Nach der Installation wird das Applet durch Kontakt mit einem Kartenleser selektiert



Lebenszyklus des Applets (oracle.com)





Aufruf des Applets



Request-Response

- Die `process`-Funktion eines Applets wird vom Kartenleser aufgerufen
- Als Parameter wird eine APDU-Instanz mitgegeben
 - https://de.wikipedia.org/wiki/Application_Protocol_Data_Unit
- Diese Instanz
 - enthält sowohl die Daten, die der Chipleser übermittelt
 - und dient als Medium zum Schreiben des Ergebnisses
 - Dies entspricht einem klassischen Request-Response-Zyklus
- Der APDU-Buffer hält Daten ausschließlich im RAM der Karte



Memory Management



Transient und Persistent

- Das transiente RAM wird ergänzt durch das persistente Memory im EEPROM
 - Großer Unterschied zur Klassischen Programmierung!
 - Objekte im Speicher der Card müssen damit zumindest teilweise als dauerhaft angesehen werden
- Regeln:
 - Alle lokalen Variablen einer Methode sind transient
 - Alle Objekte werden persistent abgelegt
 - damit sind notwendigerweise auch alle Attribute persistent
- Als Spezialfall existieren Arrays mit transienten Elementen
 - Diese werden zurückgesetzt bei
 - Deselect (CLEAR_ON_DESELECT)
 - Reset bzw. Power on (CLEAR_ON_RESET)
 - Deselect ist hier natürlich ebenfalls eingeschlossen



Im Prinzip ganz einfach, aber...

- kann komplexer werden:

```
public class TransientPersistent {  
    short s = 42;  
    short[] ss = new short[8];  
    byte[] bs = { 0x19, (byte) 0x84 };  
    byte[] tss = JCSystem.makeTransientByteArray((short) 8,  
        JCSystem.CLEAR_ON_RESET);  
  
    public void foo() {  
        short[] localSs = { 1, 2 };  
    }  
}
```



Was ist was?

- **s**
 - Ein Attribut einer Klasse, damit ein persistentes Value
- **ss**
 - Ein Attribut einer Klasse, damit eine persistente Referenz
 - auf ein Array mit persistenten Elementen
- **bs**
 - ebenso
- **tss**
 - Ein Attribut einer Klasse, damit eine persistente Referenz
 - auf ein Array mit transienten Elementen
 - Innerhalb der Methode sind damit alle Elemente auf binär 0
- **localS**
 - Eine lokale Variable und damit eine transiente Referenz
 - auf ein persistentes(!) Array
 - { 1, 2 } ist eine Abkürzung für `new short[] { 1, 2 }`
 - damit erzeugt diese Anweisung auf Karten ohne Garbage Collector ein Memory Leak



Buildprozess



Code

- Normale Entwicklung eines Java-Programms unter Benutzung der JavaCard-Bibliotheken
 - Theoretisch könnte auch ein Standard JDK benutzt werden
 - dann müssten jedoch durch Richtlinien beispielsweise die Benutzung nicht zulässiger Datentypen verboten werden
 - eher akademisch interessant...



Compile

- Der normale Java-Compiler `javac` wandelt den Quellcode in Bytecode um
- Zusätzlich werden Export-Files erzeugt, die die vom Quellcode benutzten externen Klassen enthält
 - Diese bereiten die Konvertierung in ein CAP-File vor
 - Im Endeffekt wird dadurch das „Late Binding“ einer Standard Java Virtual Machine zu einem statischen „Early Binding“ umgewandelt



Verify

- Der vom Compiler erzeugte Bytecode kann
 - valide sein oder
 - aus einem „normalen“ Java-Programm erzeugt worden sein
 - und damit unzulässige Anweisungen enthalten
 - von einem nicht-zertifizierten Compiler erstellt worden sein
 - und damit ebenfalls unzulässige Anweisungen enthalten
 - nachträglich durch direkte Bytecode-Manipulation verändert worden sein
- Damit ist eine Validierung unbedingt notwendig
 - Diese macht eine Java Virtual Machine während der Laufzeit
 - Der Bytecode-Verifier als Bestandteil der JVM
 - Bei JavaCard ist eine offline Phase vorzusehen, da die Java Card Virtual Machine diese Validierung keinesfalls durchführen kann

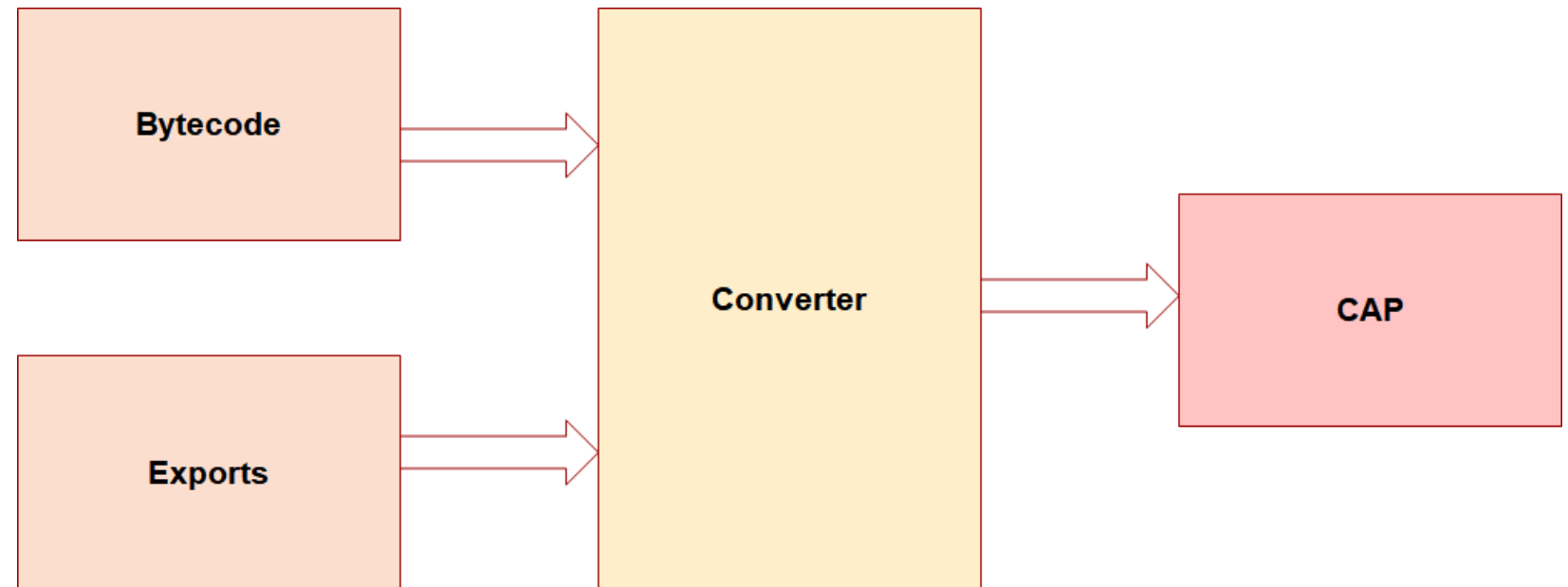


Aufgaben des Bytecode-Verifiers

- type correctness
- No stack underflow or overflow
- Objects are initialized
- Not forged objects references on the bytecode
- Illegal casting of object references between classes
- Only allowed JC API calls
- Jumping in the middle of legal JC API method
- Jumping to data as it were code



Convert zum Cap-File





Exkurs: Ein Angriffs-Szenarium

- Zwischen der Verifikation des Bytecodes und dem Erstellen des CAP-Files existiert ein Zeitfenster, in dem ein Angreifer den Bytecode nochmals modifizieren kann
 - Ebenso kann natürlich auch das CAP-File manipuliert werden
- Damit ist ein logischer Angriff auf eine Karte durch „Malformed Bytecode“ durchaus möglich
 - Beispielsweise kann dadurch ein Zahlenwert als Referenz = Speicheradresse umgedeutet werden, was einem illegalen Zugriff auf RAM oder EEPROM entspricht
- Was ist der Sinn eines solchen Angriffs oder „Warum hacke ich meine eigene Karte???“
 - Auf einer Java Card können mehrere Applets parallel installiert werden
 - Hat ein Angreifer während der Installationsphase den physikalischen Zugriff auf die Karte so kann zusätzlich ein Angriffs-Applet installiert werden
 - Mit solchen Applets können Details der interne Implementierung einer Karte und deren JCVM und damit potenziell Schwachstellen erkannt werden
 - und damit der eigentliche Angriff vorbereitet werden



Global Platform

- Die GlobalPlatform-Initiative fokussiert auf diese Problematik
 - Signaturen garantieren die binäre Unversehrtheit von geprüften CAP-Files
 - Bei der Installation eines Applets muss ein Karten-spezifischer Schlüssel bekannt sein



Transactions



Motivation

- Auf Grund der potenziell unzuverlässigen Stromversorgung sind eigentlich unkritische Sequenzen gefährlich
 - und führen zu potenziellen Inkonsistenzen
- Deshalb wird ein Transaktionsmechanismus eingeführt, in dem mehrere Attribut- und damit Speicherzugriffe gruppiert werden können
 - `JCSystem.beginTransaction()`
 - `JCSystem.commitTransaction()`
 - `JCSystem.abortTransaction()`
- Transaktionen können theoretisch verschachtelt werden
 - `JCSystem.getTransactionDepth()`
 - Aktuell: 1
- Die Größe der Datenänderungen innerhalb einer Transaktion ist beschränkt
 - `JCSystem.getMaxCommitCapacity()`
 - `JCSystem.getUnusedCommitCapacity()`



Beispiel Fehlerhaft ohne Transaktionen

```
public class Transactional {  
    short lastSuccessfulTransactionId;  
    short account;  
    public void updateWallet(short id, short delta) {  
        lastSuccessfulTransactionId = id;  
        account += delta;  
    }  
}
```



Beispiel Fehlerhaft ohne Transaktionen

EEPROM	
lastSuccessfulTransactionId	41
amount	100

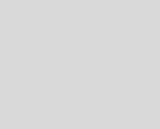


Cryptography



Features

- symmetric encryption and decryption algorithms
- asymmetric encryption and decryption algorithms
- key interfaces
- signature generation and verification
- message digests
- random data generation
- PIN management







Artikel von Victor Gayoso Martinez

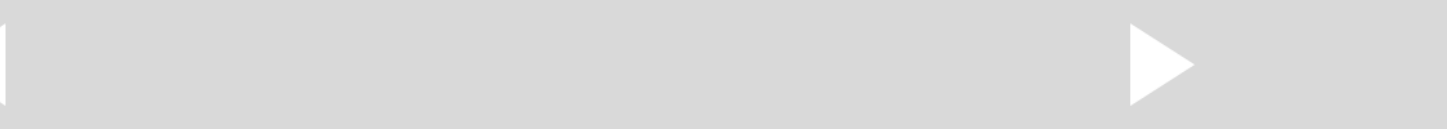




Best Practices



-  Card Limitations
-  Design
-  Optimization
-  Angriffe



Card Limitations



Card Limits

- Card resources are extremely constrained
 - EEPROM size typically is 32-64Kb
 - RAM size is usually around 4Kb, and must accommodate for application stack, transient data and various runtime buffers
- CPU power is very limited.
 - Java Card applets are much slower than native code, so you don't want to perform unnecessary operations
- Portability
 - Even though your applet runs fine on your platform, you still want to save as much resources as possible, so that it will also run on more limited platforms
 - If the applet must be interoperable, do NOT use any proprietary APIs



Design



Java Card is not Java

- class / interface overhead in the CAP file
- virtual method overhead at run time



High-level design

- • class proliferation
 - keep the number of classes / abstract classes / interfaces under control
- utility classes / interfaces
 - instead of defining classes or interfaces containing only static methods, move those methods to classes which are actually instantiated
- deep class hierarchies
 - the necessity of inheritance must be crystal-clear: if not, it has to go
- design patterns
 - patterns look great in specs, but their implementation means lots of abstract classes, interfaces and deep class trees
- systematic get()/set() methods
 - ncreasing member visibility will allow you to remove them, hence reducing code size and improving execution speed
- package proliferation: there's really no need to break an application into packages unless one of the packages is going to be used by another applet
- Use Java exceptions for Error Handling



Data storage

- writing in RAM is fast
- writing in EEPROM is awfully slow
- allocating objects is even worse
- garbage collection is not part of Java Card 2.1.1



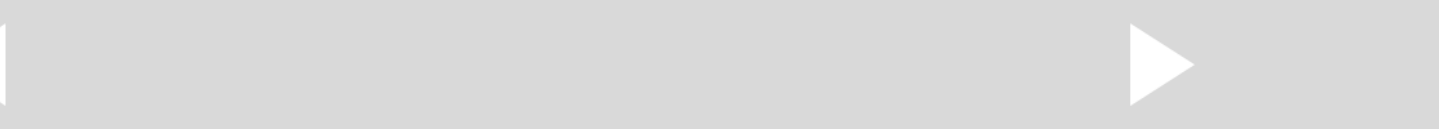
Memory allocation

- The key rule is to create all instance data at installation time
 - in the applet constructor called by `javacard.framework.Applet.install()` object construction using `new`, transient array allocation, all explicit initializations
 - This way, there's no risk to face a memory shortage during the life of the applet
 - Furthermore, applet commands won't be slowed down by memory allocation
- Memory writes
 - Objects are stored in EEPROM.
 - This means that every time you're doing "`myObject.myByte = (byte)5`", you are writing in EEPROM
 - If you need to store persistent data, there are many cases when you only need to store temporary data (intermediate result, data which is sent back to the terminal, etc)
 - This is where the APDU buffer comes in handy



Security

- If the applet is submitted to a security evaluation (Common Criteria, etc.), some specific steps may be taken
 - isolate all security functions (key/PIN handling, etc.) in separate class
 - This way, only these classes need to be documented and evaluated
 - use the Java Card API as much as you can (PIN, keys, etc)
 - If the platform offers the security services you need, it's useless and unsafe for the applet to implement them
- Fixed-size PIN
 - Use fixed-size PINs, i.e. store the PIN length in the PIN itself and pad the remaining bytes
 - This helps prevent timing attacks and also simplifies PIN handling
- Anti-DFA RSA signatures
 - Verify all RSA signatures before sending the result to the terminal
 - This will slow down the signature operation, but it will prevent DFA attacks
- Keep an eye on platform bugs



Optimization



Reducing code size

- Avoid complex object-oriented designs
 - Keep the number of classes/interfaces to a minimum
 - Combine “similar” classes and discriminate them at instantiation time using constructor parameters
 - Keep the number of methods to a minimum
 - In particular, get rid all get()/set() methods, increase the visibility of data members (e.g. from private to package visible) to reduce the number of get()/set() methods
 - Since you save method calls, this will also improve performance



Remove dead code

- Hunt down unused variables and code
- Also, try to keep the initialization / personalization as short as possible
- Hunt down all redundant code and factor it
 - If this means breaking down classes and replacing them with private/static methods, do it.
 - Tradeoff: possible slowdown if the code is called very often. Limit number of parameters
- Try to use no more than 3 parameters for virtual methods and 4 for static methods
 - This way, the compiler will use a number of bytecode shortcuts, which help reduce code size
 - Tradeoff: noticeable slowdown if this means saving/reading data from object instances.



Reducing EEPROM consumption

- Recycle all objects
 - the Java Card Virtual Machine runs "forever", so if an object becomes unreachable, its memory is gone "forever" (or at least until you delete the applet)
 - Even if your platform provides proprietary garbage collection, you'd better reuse your objects
- The OS allocates memory in 32-bytes chunks, called clusters
 - A cluster cannot be shared between two objects, so any object will always eat at least one cluster, no matter how small it is
 - The Virtual Machine appends a header to all objects:
 - 6 bytes for "normal" objects
 - 8 bytes for primitive type arrays
 - 12 bytes for object arrays
 - This means that instead of creating several small arrays of the same type, you should combine them into a single array, accessing the latter using fixed offsets
 - Tradeoff: slight increase of code complexity



Reducing RAM consumption

- Reuse local variables
 - Instead of allocating a new local variable any time you need one, try to reuse one that has been previously declared
- Instead of creating several small transient arrays of the same type, combine them into a single array and access it using fixed offsets
- Try to use no more than 3 parameters for virtual methods and 4 for static methods
- Avoid deeply-nested method calls
 - Dangerous nested calls usually happen with deep class trees (base class method calls) and with recursion
- Beware of local variables in switch/case structures



Improving execution speed

- Switch on compiler optimization
 - If your compiler provides optimization flags, use them
- Avoid complex object-oriented designs
- Favor static, private, & final methods
- Use native APIs
 - Whenever the platform provides native code to perform an operation, use it
- Use transactions with care
- Since many native APIs already transaction write operations, make sure that any explicit use of transactions in the applet is mandatory
- Avoid unnecessary initializations
- Don't use exceptions for flow control
- Use RAMbuffers
 - Writing in EEPROM is about 1,000 times slower than writing in RAM, so the less you do it, the better
 - You can work with the APDU buffer or with transient arrays to store session data and temporary results
- Clean up your loops
 - Accessing an instance member is costlier than accessing a local variable



Angriffe



Präsentation von Sergi Casanova und Pol Matamoros



JAVA CARD SECURITY Introduction and attacks

October 21, 2014

Sergi Casanova scasanova@bitwise.cat
Pol Matamoros poi@bitwise.cat

bitwise.cat