# Object Oriented Programming in Java

Hochschule Ravensburg-Weingarten
Prof. Dr. Martin Zeller

# Why (not) using OOP / Java?

Compared to C, C++, assembler

- Lower performance
- Higher demand for memory
- No low level access to hardware
- Less error prone
- Greater development productivity
- Better maintainability
- Higher degree of portability

# Availability of Java

- PC / Server / SoCs (e.g. RaspberryPI)
  Windows, Linux/Unix, MacOS, . . .
- Mainframes
  z/OS
- Embedded Systems (few)
- Handheld
  (Android with non-standard library, limited support on other smartphones)

# Lerning targets

You can:

- Explain and use oo-features of the Java language
- Explore and use the Java Standard Library

# Outline of the Course

- Procedural Programming – the base
- Classes and Objects
- Composition and inheritance
  – don't repeat yourself
- Encapsulation – controlling access
- Exception handling – catching runtime-errors
- The standard library: Java I/O, Container, . . .
- Design-Patterns: Building blocks
- JNI - Linking Java and C

# Bibliography

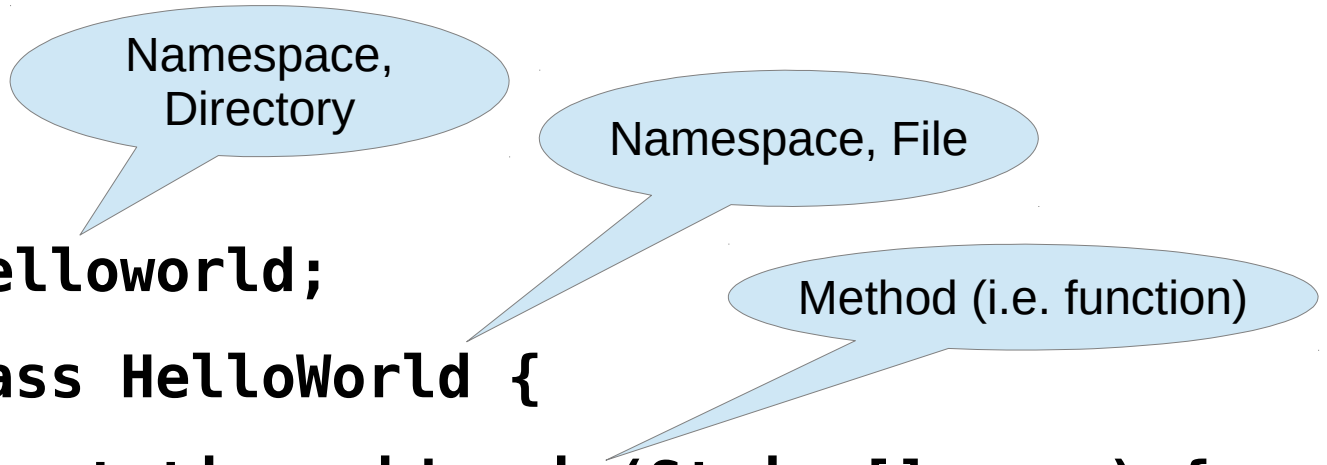

- Bruce Eckel: *Thinking in Java.* Prentice Hall
- Christian Ullenboom: *Java ist auch eine Insel.* Rheinwerk Verlag
- David J. Barnes and Michael Kölling: *Objects First with Java.* Prentice Hall
- Joshua Bloch: *Effective Java.* Addison Wesley
- K. Arnold, J. Gosling, D. Holmes: *The Java programming Language.* Prentice Hall
- Simon Kendal: *Object Oriented Programming using Java.* Ventus Publishing

# Development Environment

- NetBeans
- BlueJ
- Eclipse
- IntelliJ IDEA
- Quite a few more . . .

The assigments for the lab course come as NetBeans projects.

# Hello World in Java




```
package helloworld;
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(" Hello World ");
        System.out.println(" I am up an running ");
    }
}
```

# Compiling/Executing

Java source code → Java byte code → Execution

- JRE
  - Java virtual machine
  - Java runtime library
  - Run time support programs
- JDK
  - JRE
  - Compiler, packager . . .
  - Compile time support programs

# Variables and Types

- Primitive types
  - Boolean
  - Numbers
  - Character
- Classes
  - Similar to structures in C
  - Several extensions – details later

# Primitive Types

- boolean: {true, false}
- byte: $-2^7 \ldots 2^7-1$
- short: $-2^{15} \ldots 2^{15}-1$
- int: $-2^{31} \ldots 2^{31}-1$
- long: $-2^{63} \ldots 2^{63}-1$
- char: $0 \ldots 2^{16}-1$ (unicode character)
- void: no value
- float, double: 32 bits / 64 bits IEEE 754

No unsigend numbers

# Classes for Primitive Types

For each primitive type a class exist

- boolean: Boolean
- byte: Byte
- short: Short
- int: Integer
- long: Long
- char: Character
- float, double: Float, Double

Holds one value of type boolean

# Control-Statements

Loops:

- while
- for
- do
- break
- continue

Branches:

- if
- switch

Accepts constant strings as label

# Operators

Similar to C (but no sizeof operator)

- Assigment: = ++ -- += *= . . .
- Arithmetic: + - * / %
- Relational, logical: > == < ! && || . . .
- Bitwise: & | << >> >>> . . .
- Some more . . .

unsigend right shift

# Order of Evaluation

Evaluation of operators:

- Precedence
- Associativity

As in C

Evaluation of subexpressions:

- Left to right (in Java specified)

Unspecified in C

# Order of Evaluation

Example: `x = a() + b() + c() * d();`

```
av = a();
bv = b();
cv = c();
dv = d();
```

Java ensures this order.
In C any order is possible

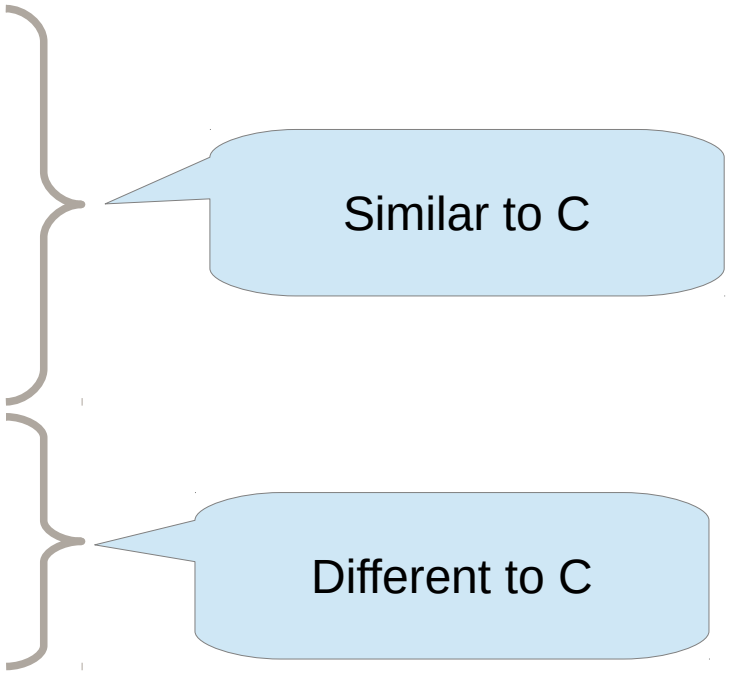
```
x =  av + bv + (cv * dv);
```

Order of execution is not specified.
Result is independent of the order.

# Methods i.e. Functions

```
System.out.println(" Hello World ");
```

- Name
- Return value
- Parameters
- Name need not be unique
- Bound to a class/object



# Multiplication Table Example

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.prinln("Enter number of columns:  ");
    int columns = input.nextInt();
    System.out.println("Enter number of rows:  ");
    int rows = input.nextInt();
    for (int i = 1; i <= rows; i++){
        for (int j = 1; j <= columns; j++){
            System.out.printf("  %3d", i*j);
        }
        System.out.println();
    }
}
```

# Import Classes

## Import a class of a library

```
import packageX.packageY.ClassName;
```

## Example:

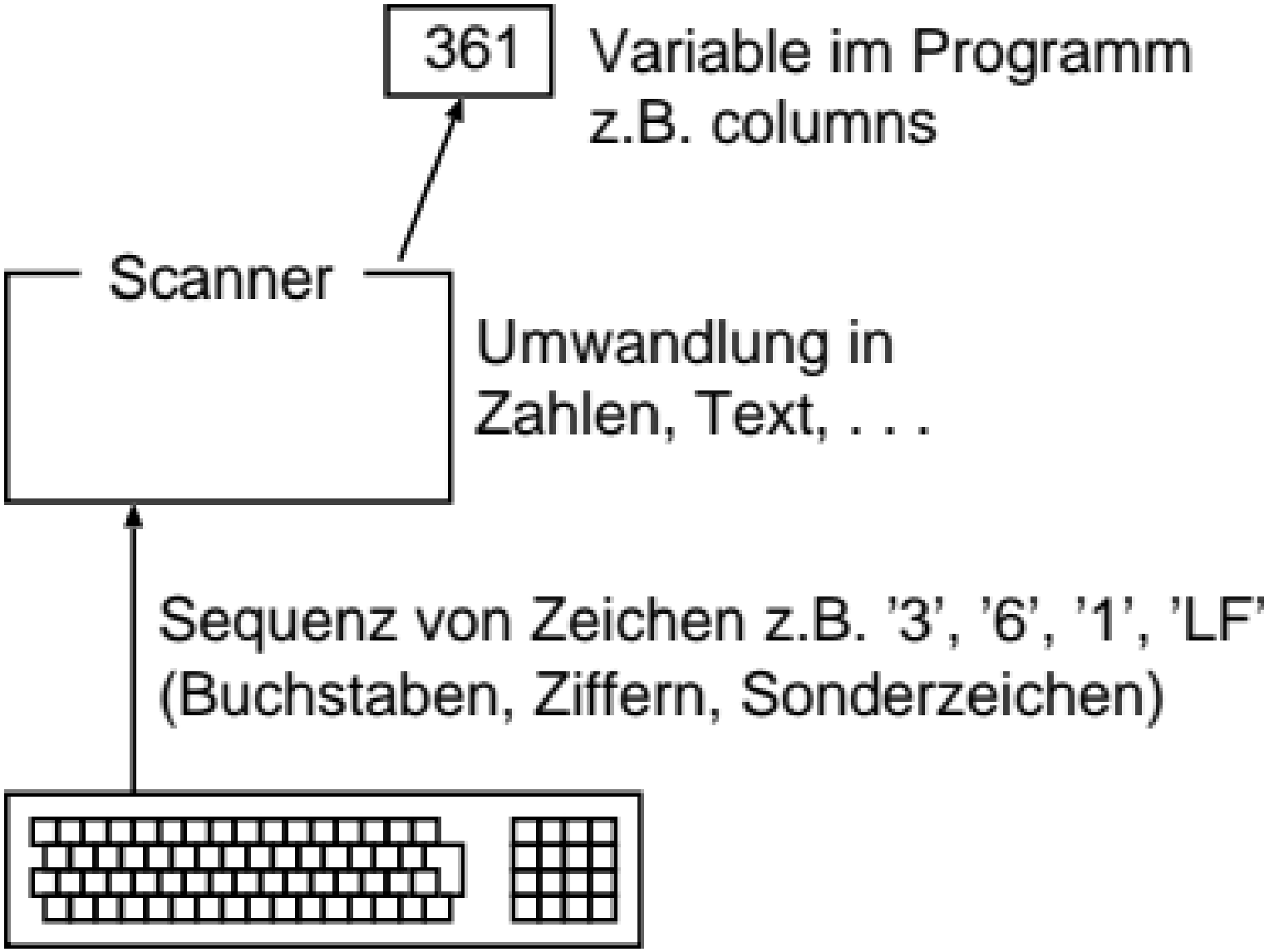
```
package multiplicationtab;


import java.util.Scanner;

public class MultiplicationTab {
    public static void main(String[] args) {
       Scanner keyboard = new Scanner(System.in);
                    :
    }
}
```

Import class Scanner

# Scanner – reading input



# Arrays in Java

- Definition of a reference to an array-object:

```
int[] numbers;      // no storage is
int numbers[];      // allocated
```

- Allocated on the heap:

```
numbers = new int[10];
```

- Access to one element:

```
numbers[i]
```

A new Array-Object is created

# Arrays in Java

```
int[] numbers = new int[10];
```

Access to an element is checked:

```
numbers[0]   // OK
numbers[3]   // OK
numbers[9]   // OK
numbers[10]  // Program-flow is interrupted
numbers[-4]  // Program-flow is interrupted
```

# Arrays in Java

The length of an array is stored in a member:

```
numbers.length
```

The length is automatically stored on creation.

A typical loop :

```
for (int i = 0; i < numbers.length; i++){
        :
    numbers[i]
        :
}
```

# Strings in Java

```
public static void main(String[] args) {
    String firstName = "Bob_";
    String lastName = null;
    System.out.println(firstName);
    System.out.println(lastName);

    firstName = "Frank";
    lastName = new String("furt");
    System.out.println(firstName + lastName);
}
```

null: No object

Same visible effect

May occupy more memory

# Strings in Java

- String is a class not `char[]`
- No end mark (zero-byte)
- Strings are immutable and memory efficient
- Mutable Strings:
  - StringBuilder (not thread safe)
  - StringBuffer (thread safe)

For changing Strings these alternatives are faster.

# Strings in Java

- Concatenation by operator '+'
- Characters are encoded in UTF-16
- Access one character:

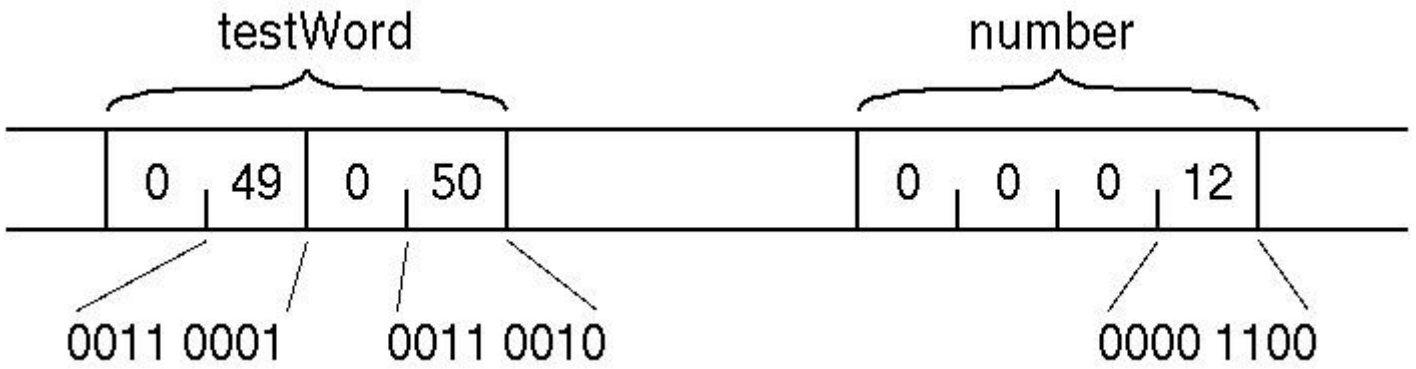
```
char aChar = firstName.charAt(4);
```

Fifth character

- Class String: more than 60 methods e.g.
  - length()
  - equals()
  - toLower()
  - substring(int beginIndex, int endIndex)

# Strings vs int

```
public static void main ( String [] args ) {
    String testWord = "12" ;
    int number = 12;
            :
}
```



# The Concept of Class and Object

- Class
  - Blueprint of objects
  - Defines members and methods


- Object (chunk of memory)
  - Identity (in Java: the reference)
  - State (values of members)
  - Behavior (methods)

# FirstObject

```
package firstobject;

public class FirstObject {

  int objectState;

  public FirstObject(int aState) {
      objectState = aState;
  }


  void printState(int num){
      int sum = num + objectState;
      System.out.println(sum);
  }

}
```

Member:
Variable bound to an object

Method i.e. function. Uses
- parameters
- locale variables
- members

# FirstObject

```
public class FirstMain {

    public static void main(String[] args) {

        FirstObject objectA = new FirstObject(1234);
        FirstObject objectB = new FirstObject(5432);


        objectA.printState(66);
        objectB.printState(-32);

    }

}
```

Creating objects

Method-call
object.method()

# Calling a Method

A method is bound to an object (or to a class).

- May use member values of the own object
- May use parameters (as in C)

```
objectA.printState(66);
objectB.printState(-32);
```

# Creating Objects 2

new <class-name>( . . . )

- allocates memory for the object
- initialises members (0, false, null)
- calls the appropriate constructor

```
public ExampleObject(int aState){
     objectState = aState;
 }

 public static void main(String[] args) {
     ExampleObject exObject1 = new ExampleObject(111);
     . . .
 }
```

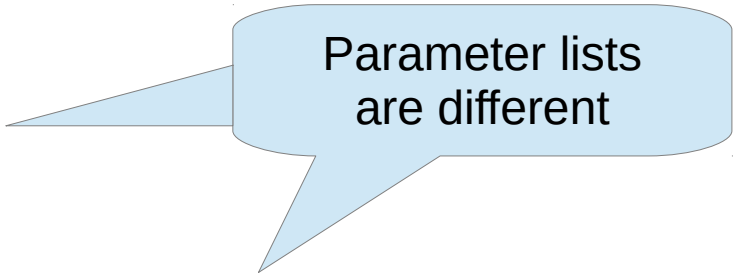
# Creating Objects 3

Constructor: a special method

- Same name as class
- No return value
- Automatically called upon `new . . .`
- Each class has at least one constructor
- More constructors a possible

# Method Overloading

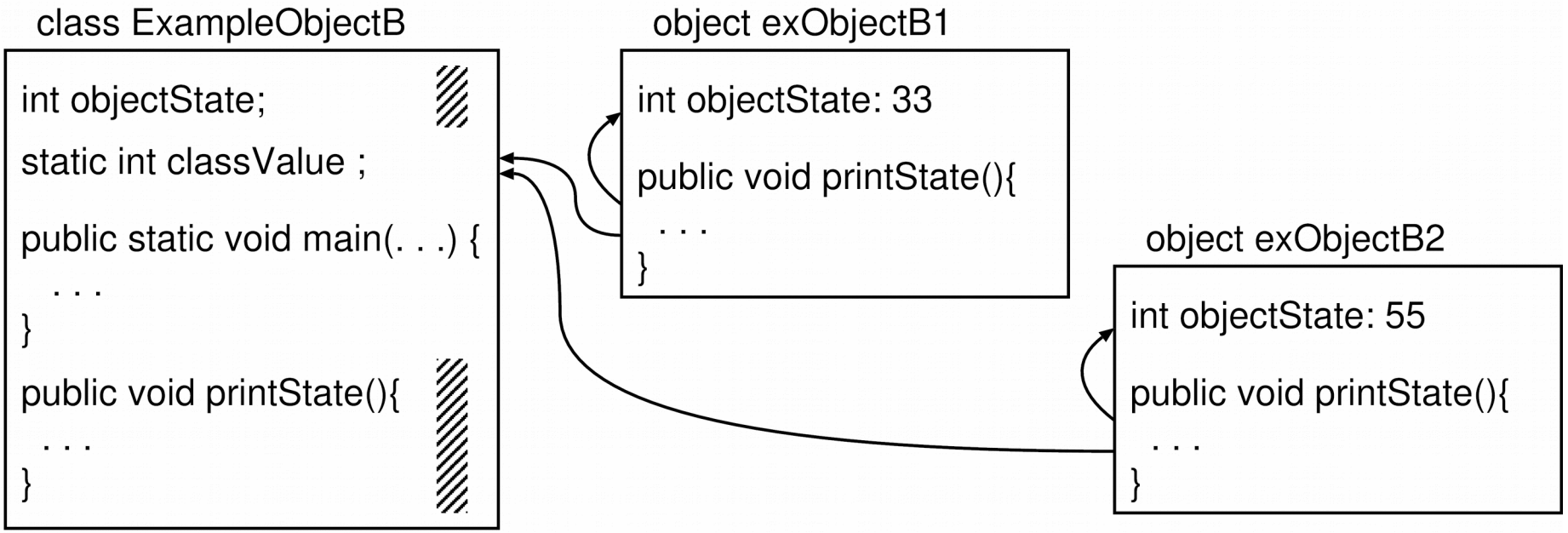
- More than one constructor for a class
- More than one method with the same name in a class




```
public ExampleObject(){
    objectState = 42;
}
public ExampleObject(int objectState_){
    objectState = objectState_;
}
public static void main(String[] args) {
    ExampleObject exObject1 = new ExampleObject();
    ExampleObject exObject2 = new ExampleObject(111);
    . . .
}
```

# The Concept of Class and Object

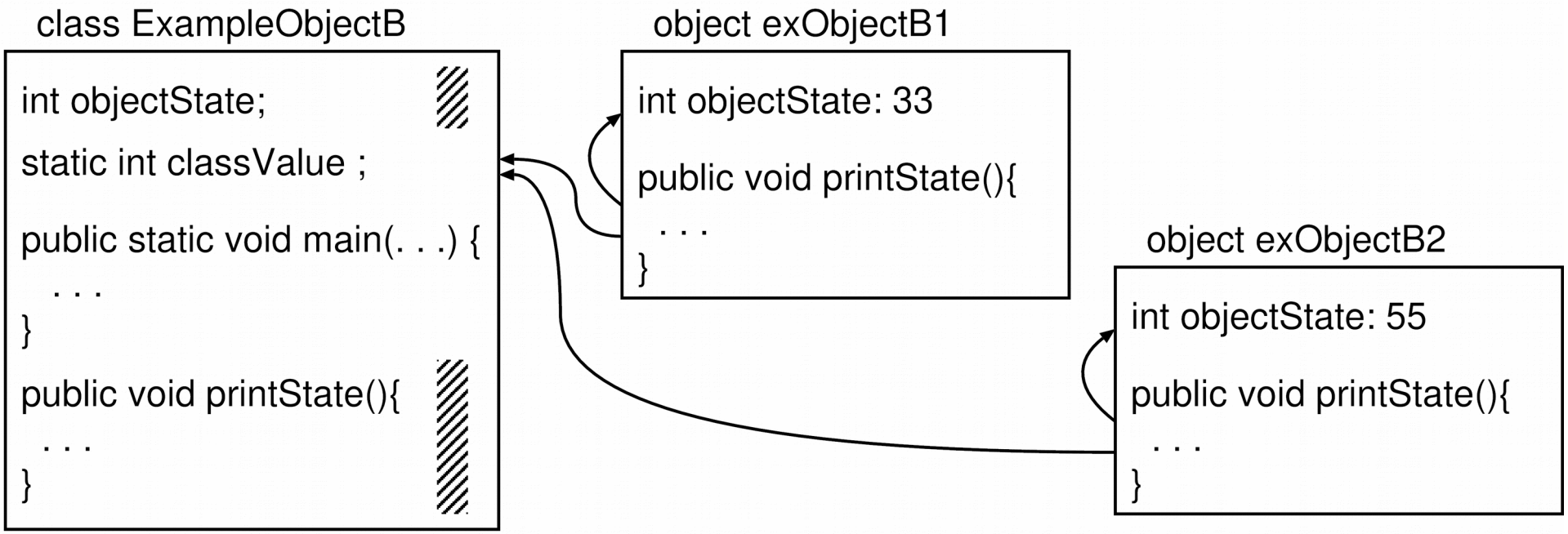
- Static members and methods accessable from static and non-static context
- Non-static members and methods accessable only from non-static context



# The Concept of Class and Object

A static member exists only in the class – regardless how many objects exist.

- Member **`classValue`** exists only once
- Each object has its own member **`objectState`**



# Class-Type Variables

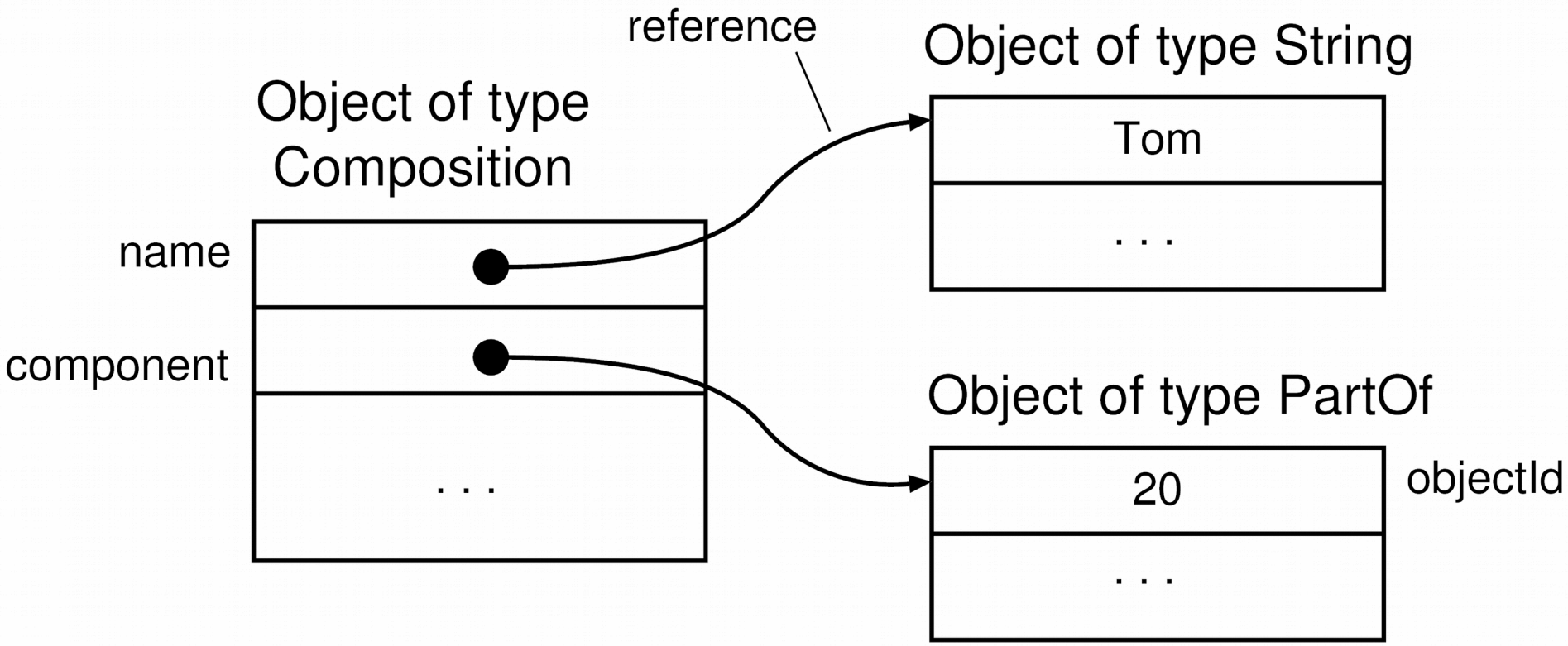
Variables of a class - type (e.g. objectA)

- Assignment:
```
objectA = new  . . .
objectB = null;
```
- Comparison (compares two references)
```
if (objectA == objectB) . . .
```
- Dereferencing `objectA.`
  The dot means: Go to the object
- No arithmetics on references
  e.g. increment or decrement

# Class-Type Variables: References

```
public class ExampleObject {
  int objectState;
  public ExampleObject(int objectState_){
      objectState = objectState_;
  }
  public static void main(String[] args) {
      ExampleObject exObject1 = new ExampleObject(111);
      ExampleObject exObject3 = exObject1;
      exObject3.objectState = 222;
      exObject1.printState();
  }
  public void printState(){
      System.out.println(objectState);
  }
}
```

# Linking Objects



# Linking Objects

```
public class Composition {
  String name;      // reference to an object
  PartOf component; // reference to an object

  Composition(String name_, int compId) {
    name = name_;                    // copy reference
    component = new PartOf(compId); // create object
  }
  public static void main(String[] args) {
    Composition compObject = new Composition("Bob", 20);
    compObject.printId();
  }
  void printId() { . . . }
}
```

# Linking Objects

```
public class PartOf {
  int objectId;

  PartOf(int objectId_) {
      objectId = objectId_;
  }

  void printId() {
      System.out.print("Class: PartOf, object-id: ");
      System.out.print(objectId);
  }
}
```

# Uniques of identifiers

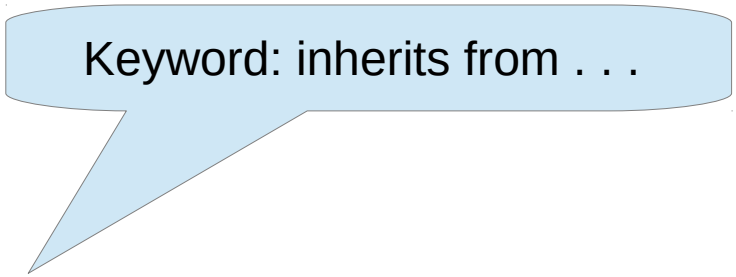
Method names need not be unique:

```
Composition.printId()
Partof.printId()
```

Different methods located in different classes.

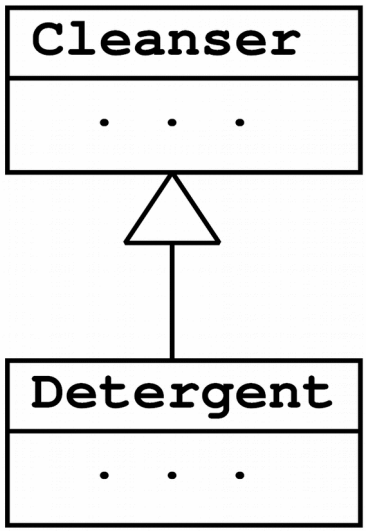
# Inheritance

```
public class Cleanser {
            :
            :
}
                . . .
 public class Detergent extends Cleanser {
            :
            :
}
```

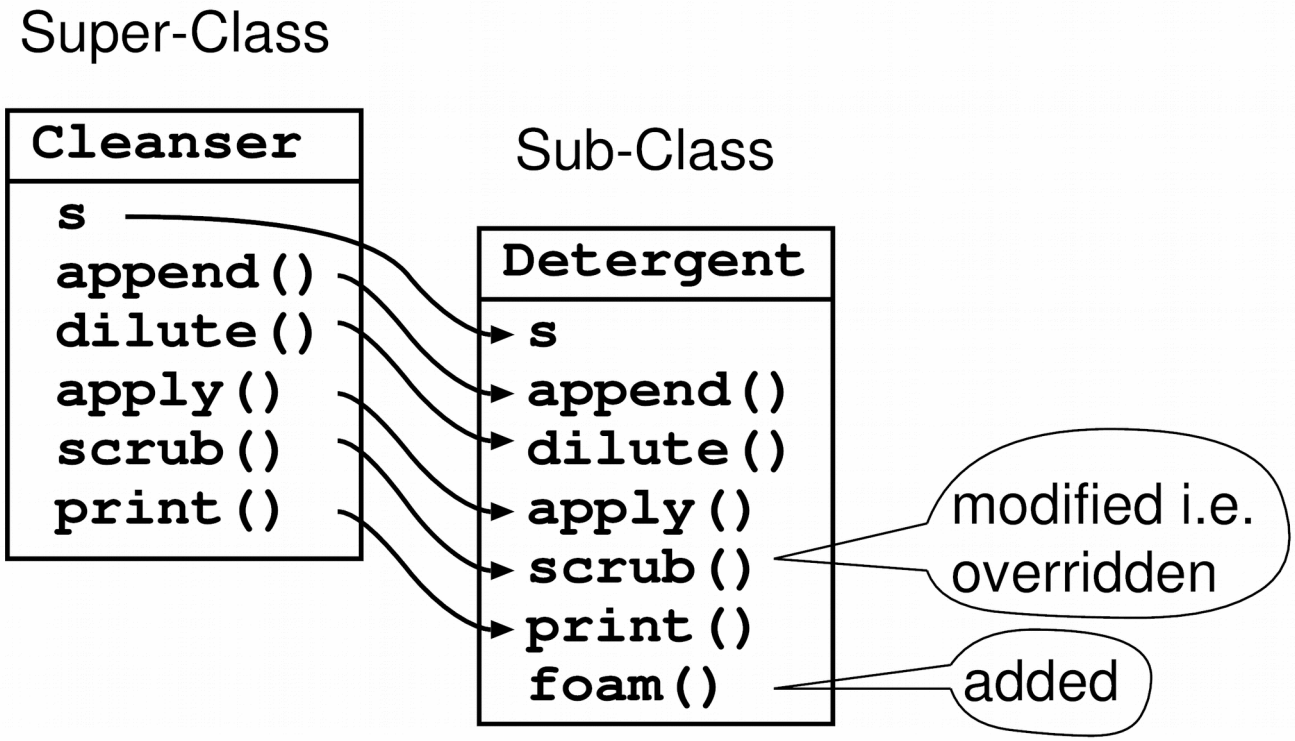


# Inheritance

## UML-Notation




## Effect



# Inheritance - Rules

- A class can inherit from at most one class.*
- The subclass contains all members and non-private methods of the super class.
- The  subclass may change (override) methods of the super class.
- The  subclass may add members and methods.

Removal of anything is not possible

* some OO programming languages allow for multiple inheritance e.g. C++.

# Inheritance - Overriding

```
public class Cleanser {
    private String s = new String("Cleanser");
    public void append(String a) {
        s += a;
    }
    public void scrub() {
        append(" Cleanser.scrub()");
    }
                . . .
 public class Detergent extends Cleanser {
    public void scrub() {
        append(" Detergent.scrub()");
    }
                        . . .
```

# Uniques of identifiers

Inherited methods can be changed:

```
Cleanser.scrub()
Detergent.scrub()
```

are different methods.

Detergent “overrides” the inherited method scrub().

The referred object is the context of the method.

# Inheritance - Overriding

```
Cleanser cl = new Cleanser();
cl.scrub();

Detergent dt = new Detergent();
dt.scrub();
```

The run time system picks the first available method `scrub()`

1) Object
2) Class of the object
3) Super class of the object (and so on)

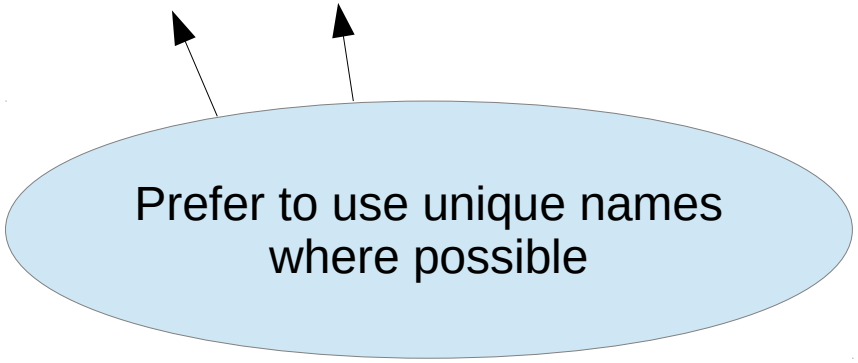
# Inheritance - **super**

```
public class Cleanser {
    private String s = new String("Cleanser");
    public void append(String a) {
        s += a;
    }
    public void scrub() {
        append(" scrub()");
    }
                . . .
 public class Detergent extends Cleanser {
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub(); // call base-class version
    }
```

Refers to the super class of Detergent

# Uniques of identifiers - **this**

```
public class Cleanser {
    String s = new String("Cleanser");
    public void append(String s) {
       this.s += s; // set the local member s
    }
```




**this** refers to the current object.

**this.s** is member **s**
just **s** is variable **s**

# Constructors and Inheritance

learn by heart

1) Base class initialization: steps (1) - (4) (i.e. recursively where applicable).
2) Members are set to default values
3) Member initializers are called in the order of declaration.
4) The constructor is called.

Example:

```
//: c07: Sandwich.java
// From 'Thinking in Java' by B. Eckel
```

# Constructors and Inheritance

Example:

```
public class Sandwich extends PortableLunch {
Bread b = new Bread();
Cheese c = new Cheese();
Lettuce l = new Lettuce();

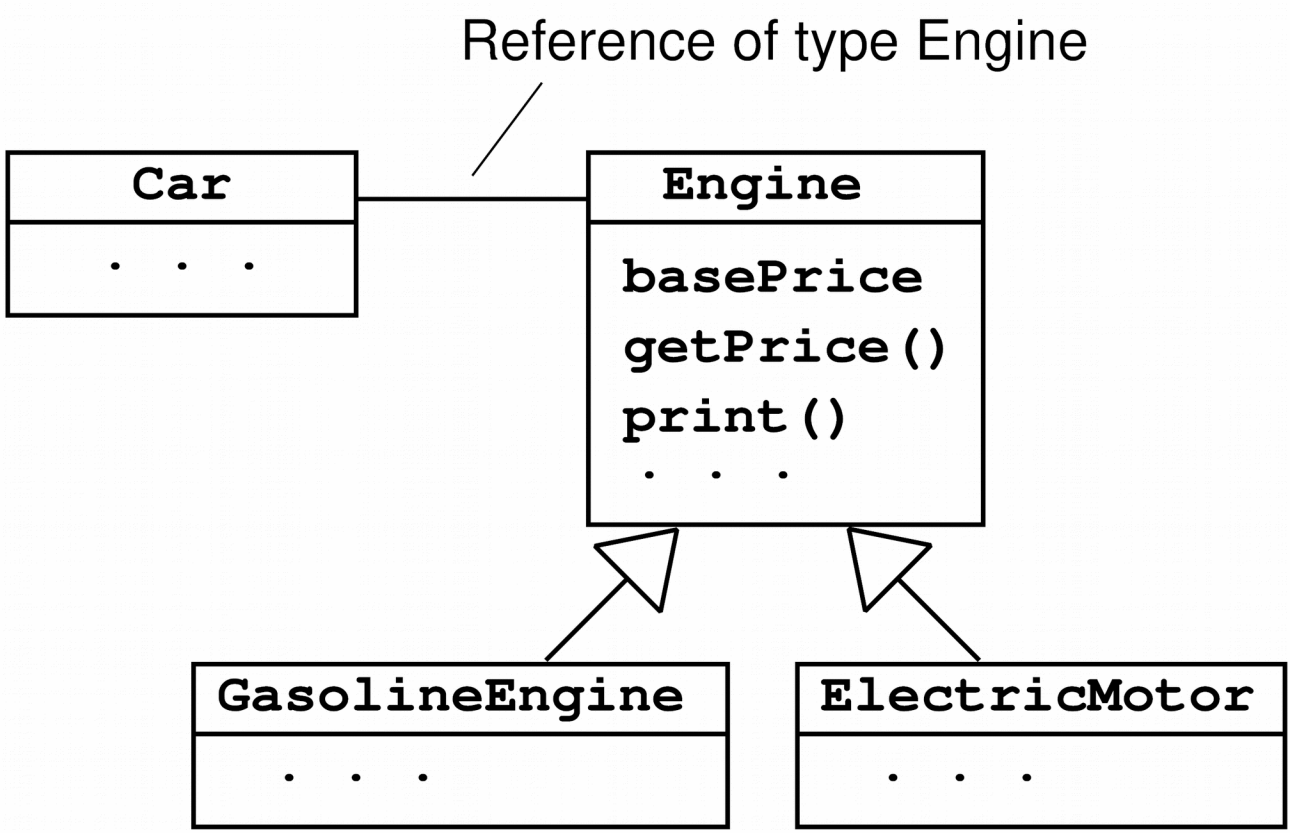
Sandwich() {
  System.out.println("Sandwich()");
}
```

**Implicit:**

```
Meal();
Lunch();
PortableLunch()
Bread b = new Bread();
Cheese c = new Cheese();
Lettuce l = new Lettuce();
```

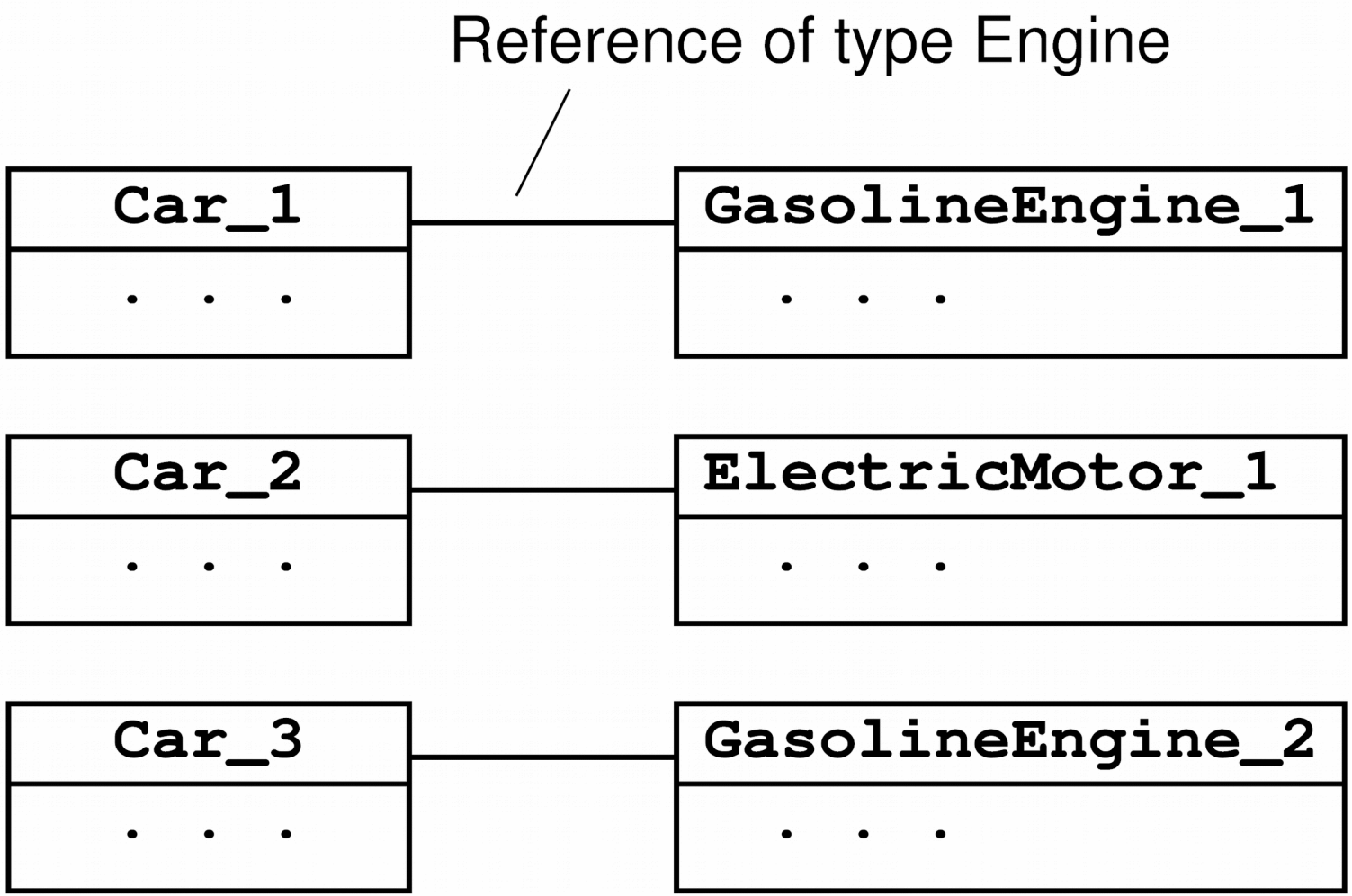
# Inheritance Example

## What we tell the compiler



# Inheritance Example

What happens at run-time



# Inheritance – Polymorphism (1)

Polymorphism:

- Method of super class-reference is called
- Method of subclass is executed

type: Engine
points to object of type GasolineEngine

- car1 calls **`theEngine.getPrice()`**
  - **`GasolineEngine.getPrice()`** is called
- car2 calls **`theEngine.getPrice()`**
  - **`ElectricMotor.getPrice()`** is called

# Inheritance – Polymorphism (2)

Datatype of the reference: Engine

Datatype of the object: GasolineEngine

```
Engine theEngine = new GasolineEngine(1000.0)
```

- Compile time check:
  Methods defined in Engine can be called.
  Based on type of the reference.
- At runtime:
  Method defined in GasolineEngine is executed.
  Based on type of the object.

# Inheritance – Polymorphism (3)

```
Engine theEngine = new GasolineEngine(1000.0)
```

- An object of a subclass can always replace an object of its super class.
- But not vice versa

```
GasolineEngine theEngine = new Engine(1000.0)
```

# Inheritance – Benefit?

Class `Car` without utilising inheritance:

- Separate reference for each type of Engine
- Separate code to access each type of Engine (get or set values, print etc.)
- Adding a new type of Engine means
  adding another reference to `Car`
  adding more code to access new type
- Check for consistency
  (how many engines may a car host?)

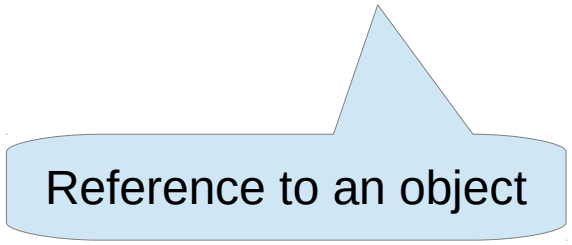
# Inheritance – method call (2)

Static method:

The class where the method is defined is the “context” of method call. Only static members can be used.

Non-static method:
The object is the “context” of method call.

```
theEngine.getPrice()
```



# Inheritance – method call (2)

Inherited method (no overriding):

- Method of base class is executed
- Current object is “context” of method call

Learn by heart

**`super.foo()`**

- Method of base class is executed
- Current object is “context” of method call

# Inheritance – Design Considerations

Inheritance vs. Composition

- Inheritance:
  Do I need to upcast?
  i.e. do I have references of the base class type?

- Combination of Compositions and Inheritance:
  Compositions allows for separate inheritance relationships (see car example or person/role).

# Inheritance – Design Considerations

## Inheritance vs. Composition

- Subclassing just `Car` (`DieselCar, ElectricCar ...`)
- Composition `Car` ↔ `Engine`
  Subclassing only `Engine`
- Test for “is a” vs. “has a”, “acts as” …

Inheritance

Composition

- A subclass must stand to the contract of the super class.

Support everything the super class supports

# Encapsulation

Access modifiers

- **`public`**
- **`private`**
- no qualifier (default)
- **`protected`**

- Which class can read/write which member
- Which class can call which method

# Encapsulation - public

## Access modifier

```
public int basePrice;
public int getBasePrice();
```

- Grants access to every class of the program
  - read/write the member **`basePrice`**
  - call the method **`getBasePrice()`**

# Encapsulation - private

Access modifier

```
private int basePrice;
private int getBasePrice();
```

- No other class of the program can
  - read/write the member **basePrice**
  - call the method **getBasePrice()**

There is an exception: An “inner class” still has access to private items.

# Encapsulation - <default>

No access modifier (package access)
in class Car:

```
double basePrice;
double getPrice();
```

- Grants access to every class within the same package
  - read/write the member **`basePrice`**
  - call the method **`getPrice()`**

# Encapsulation - protected

Access modifier in Engine

```
protected int basePrice;
protected int getBasePrice();
```

- Grants access to
  - every class within the same package
  - every sub-class within the program

  - read/write the member **`basePrice`**
  - call the method **`getBasePrice()`**

# Getter and Setter

## Access modifier

```
private int basePrice;
public int getBasePrice();
public void setBasePrice(int);
```

- No other class of the program can read/write the member **`basePrice`** directly.
- All other classes of the program can use the methods **`getBasePrice()`** and **`setBasePrice(int)`.**

➔ Preferred way of member definition

# Design Considerations

- Access should be as restrictive as possible
- Members should be private
- Less restrictive (default/protected/public)
  - Setter-Method: `setXY()`
  - Getter-Method: `getXY()`
- Grant read- and write- access separately
- Allows for checking and/or rejecting values

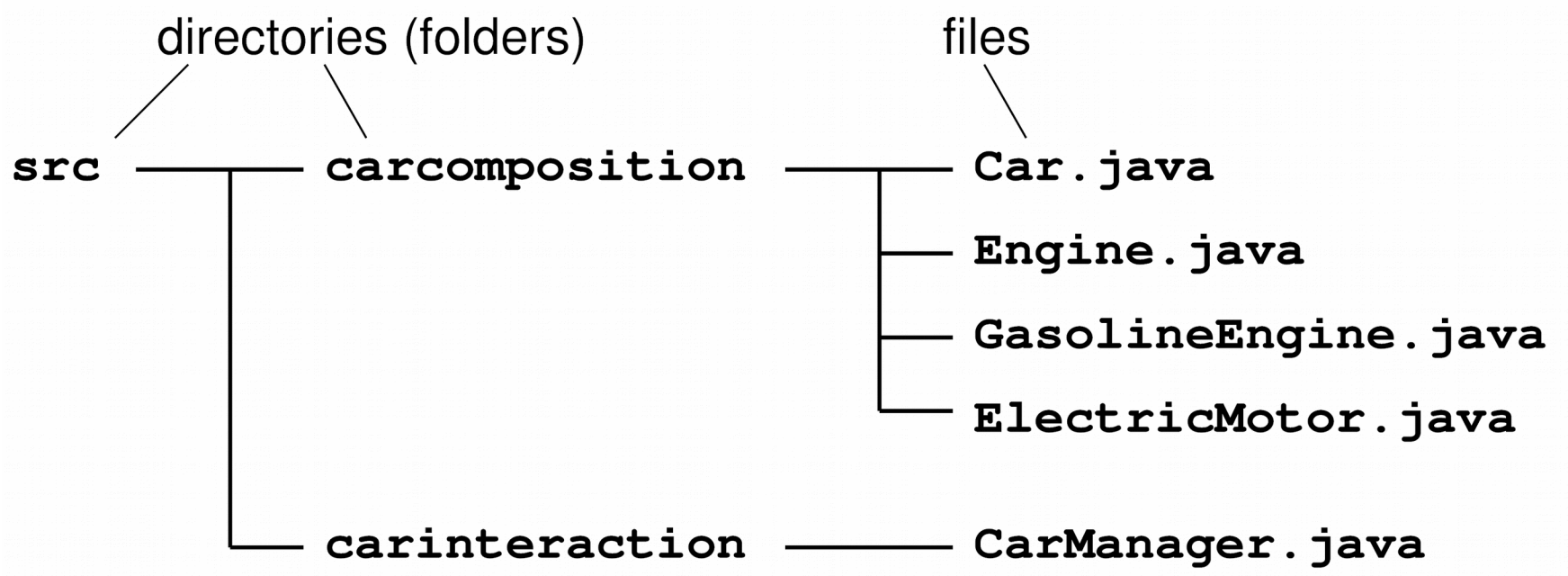
# Design Considerations

Restrictive access to members and methods:

- Minimize dependencies
- If I change a member or a method, which parts of the program are affected?
- If a member contains an incorrect value, where might it come from?

# Source Code Directories

- One directory mandatory for each package
- Packages **`carcomposition, carinteraction`**



# Design Considerations

- A package should group colaborating classes
  - Calling methods of each other
  - (accessing members of each other)
- A package forms a name space
  - Class names whithin a package must be unique
  - Each public class resides in its own file
  - Class name must be file name

# Interfaces

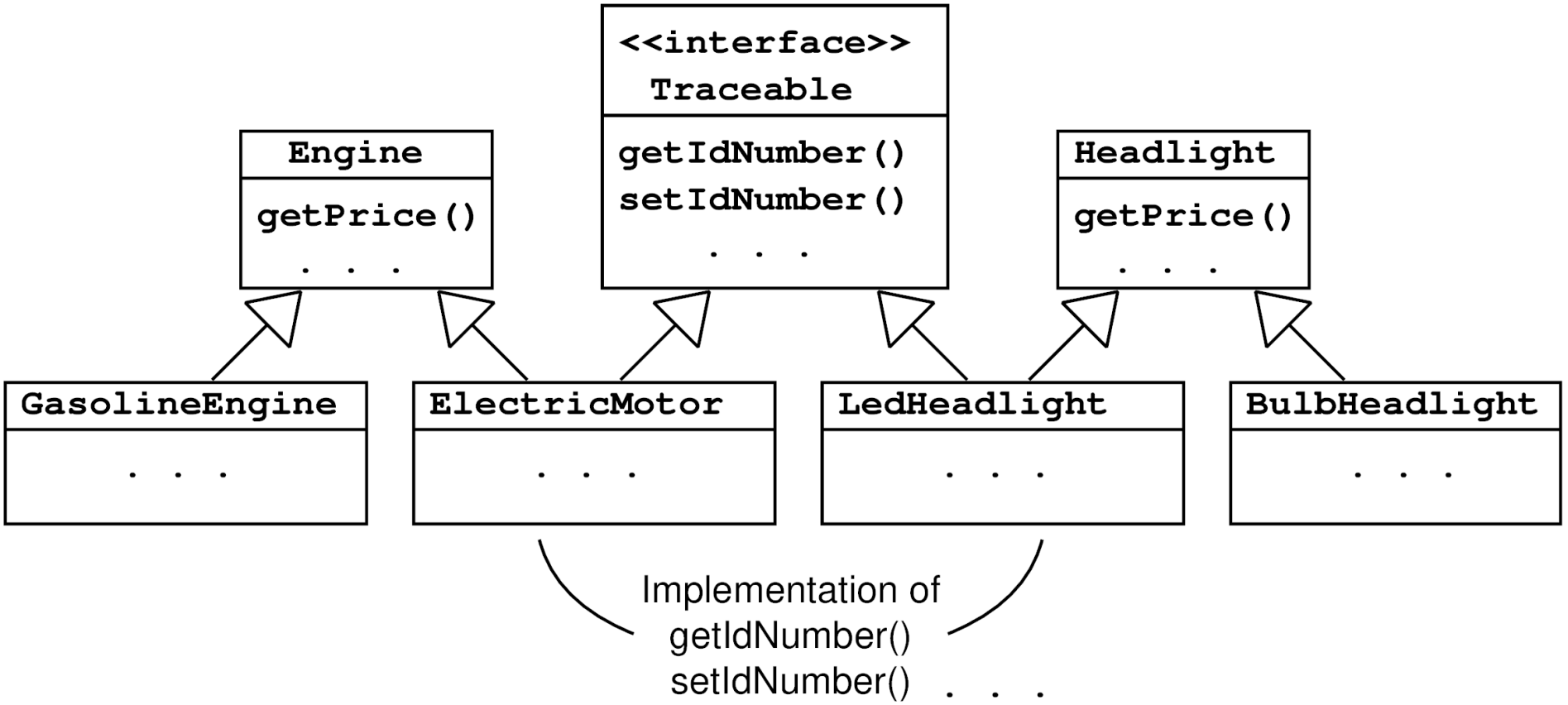
- Interface: Class without non-static members
  - Abstract (no constructor/no object)
  - Methods (implicitly public)
  - Members (implicitly public static final)

```
public interface Traceable {
    public void setIdNumber(int idNumber);
    public int getIdNumber();
}
```

- Allows for multiple appearance of an Object
- Allows objects of different types look the same

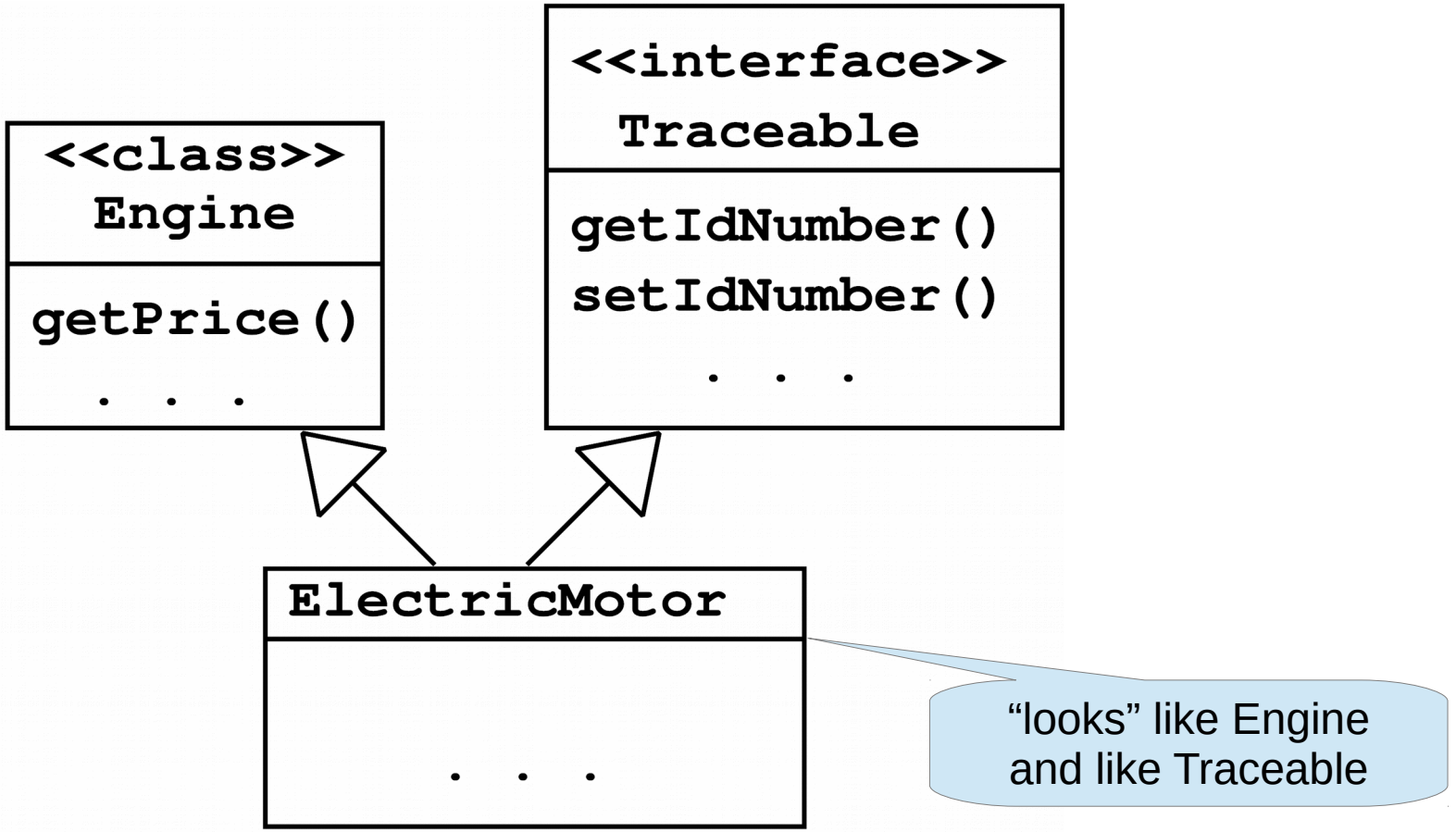
# Interfaces - Example

- A class can inherit from at most one class
- A class can inherit from many interfaces



# Interfaces - Example

- ElectricMotor inherits from class Engine and from interface Traceable



# Interfaces - Example

- Traceable is a valid type for a reference

```
Traceable[] myItems = new Traceable[10];
```

- The array may contain references to objects of type `ElectricMotor` or `LedHeadlight`.
- Only the methods defined in `Traceable` can be called on these references.

Changing the implementation of a method does not require changes in the depending code (at least not syntactically).

# Exceptions

## Signalling a runtime error:

```
public class Composition {

    public static void main(String[] args) {
            :
        compObject = null;
        compObject.printId();
            :
```

```
Exception in thread "main"
java.lang.NullPointerException
   at composition.Composition.main(Composition.java:20)
```

# Throwing an Exception

## Signalling a runtime error:

declare exception

```
public class SimpleExceptionDemoB {

  public void foo(int i) throws SimpleException {
    if (i > 1) {
      System.out.println("Throwing . . . f()");
      throw new SimpleException("error in f()");
    }
    System.out.println("foo() continues, i: " + i);
    return;
  }
```

throw exception

# Catching an Exception

## Catching run time errors:

```
try {
      sed.foo(1);
      sed.bar(1);
      sed.foo(2);
      sed.bar(2);

} catch (SimpleException ex) {
      System.err.println("Caught it!");
      ex.printStackTrace();
}
```

throws an exception

not executed due to exception

If an exception is thrown the program continues in the next matching catch-block.

# Exceptions

Two categories:

- RuntimeException (“unchecked”)
  - Predefined in the Java-Language
  - Can be ignored in the program (shows up eventually anyway)
- “checked” Exceptions
  - Subclasses of Exception (or Throwable)
  - Must be declared (enforced by compiler)
  - Must be caught (enforced by compiler)

# Exceptions

Examples of RuntimeExceptions:

- Array index out of bound
- Integer division by zero
- Dereferencing a null reference
- Attempt to do an illegal cast

RuntimeExceptions are used to indicate a programming error.

# Exceptions

- Throwing an exception should not be the normal flow of the program.
- Throwing an Exception means:
  Handle the situation later (upper).
- Where do we have enough information to handle the exceptional case?

# Exceptions

Alternatives:
Throw an exception or check for a condition in advance:

- Array: `if ((i >= 0) && (i < myArray.length))`
- Division: `if (divisor != 0)`
- Object-reference: `if (myObject != null)`

# Exceptions

## Catching different exceptions:

```
try {
    int num = 10 / x;
    System.out.println("num: " + num);
    sed.foo(2);
} catch (SimpleException2 ex) {
         System.err.println("Caught it!");
} catch (Exception ex) {
         System.err.println("General Exception");
}
```

exception if x is 0

catches any exception

## The program continues at the first matching catch.

# Exceptions

Catching different exceptions:

catches only exceptions of type SimpleException and subclasses

```
} catch (SimpleException ex) {
          System.err.println("Caught it!");
} catch (Exception ex) {
          System.err.println("General Exception");
}
```

catches any exception

An exception matches if it is of the specified type or a subtype.

→ Catch most derived exception first.

# Exceptions

## Do a clean up anyway:

```
try {
      sw.on();
      // Code that can throw exceptions...
      OnOffSwitch.f();
    } catch(OnOffException1 e) {
      System.err.println("OnOffException1");
    } catch(OnOffException2 e) {
      System.err.println("OnOffException2");
    } finally {
         sw.off();
    }
}
```

Finally clause is executed even if an exception occurs that is not caught or if a “return” is executed.

# Characters in Java (and elsewhere)

Characters are mapped to whole numbers:

- ASCII- Code:
  - Range 0 – 127
  - e.g. A⇔65, B⇔66, . . . a⇔97, b⇔98 . . .
- Unicode
  - Range 0 – $2^{32}$ (not exhausted)
  - Mapping 0 – 127 same as ASCII-Code
  - Examples: Ä⇔0x00c4, ü⇔0x00fc

# Characters in Java (and elsewhere)

Unicode character examples (unicode in hex)

0643: ش Arabic

0915: क Hindi/Devanagari

042f: Я Cyrillic

0ba3: ண Tamil

2749: ❉ Dingbats

20ac: €

# Characters in Java (and elsewhere)

Different ways to encode Unicode-Values

- UTF-32 (plain): 4 bytes per character (waste of storage, not commonly used)
- UTF-16: 2 bytes per character, 4 bytes for rarely used characters
- UTF-8: Uses 1 byte for ASCII characters and 2, 3 or 4 bytes for rarely used characters

Java uses internally UTF-16.

Most OS today prefer UTF-8.

# Unicode and UTF-8

UTF-8: One way to encode Unicode values

- Values 0-127 are encoded in one byte
- Greater values are encoded in 2 - 4 bytes

| Unicode Value | UTF-8 Encoding |
|---|---|
| 0000 0000 – 0000 007F | 0xxxxxxx |
| 0000 0080 – 0000 07FF | 110xxxxx 10xxxxxx |
| 0000 0800 – 0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx |
| 0001 0000 – 0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx |

Same as ASCII

subsequent byte

start of 3-byte value

# Handling Unicode in Java

java.io.Reader

java.io.Writer

- Read and write unicode characters to/from streams
- Support different encoding
  UTF-8, UTF-16, UTF-32, ISO-8859 -1, . . .

# File-I/O in Java

Classes of the Java Run-Time-Library for handling files (among others)

- java.io.File (represents a path)
- java.nio.file.Files
  (contains many static methods)

Basic classes for reading from/writing to a File

- java.io.FileInputStream
- java.io.FileOutputStream

# Base Classes for File-I/O

Reading/writing binary data from/to a stream

- java.io.DataInputStream
- java.io.DataOutputStream

Reading/writing unicode characters i.e. text from/to a stream

- java.io.InputStreamReader
- java.io.OutputStreamWriter

# I/O of Primitive Datatypes

java.io.DataInputStream

java.io.DataOutputStream

- Read and write primitive datatypes to/from streams (byte, short, float, . . . )
- Identical format on all platforms i.e. an int value written on a PC is readable on a main frame.

# File-I/O in Java I

Basic streams are usually “decorated”, i.e. wrapped in an other stream class:

```
FileOutputStream fos = new FileOutputStream("testFile.txt");

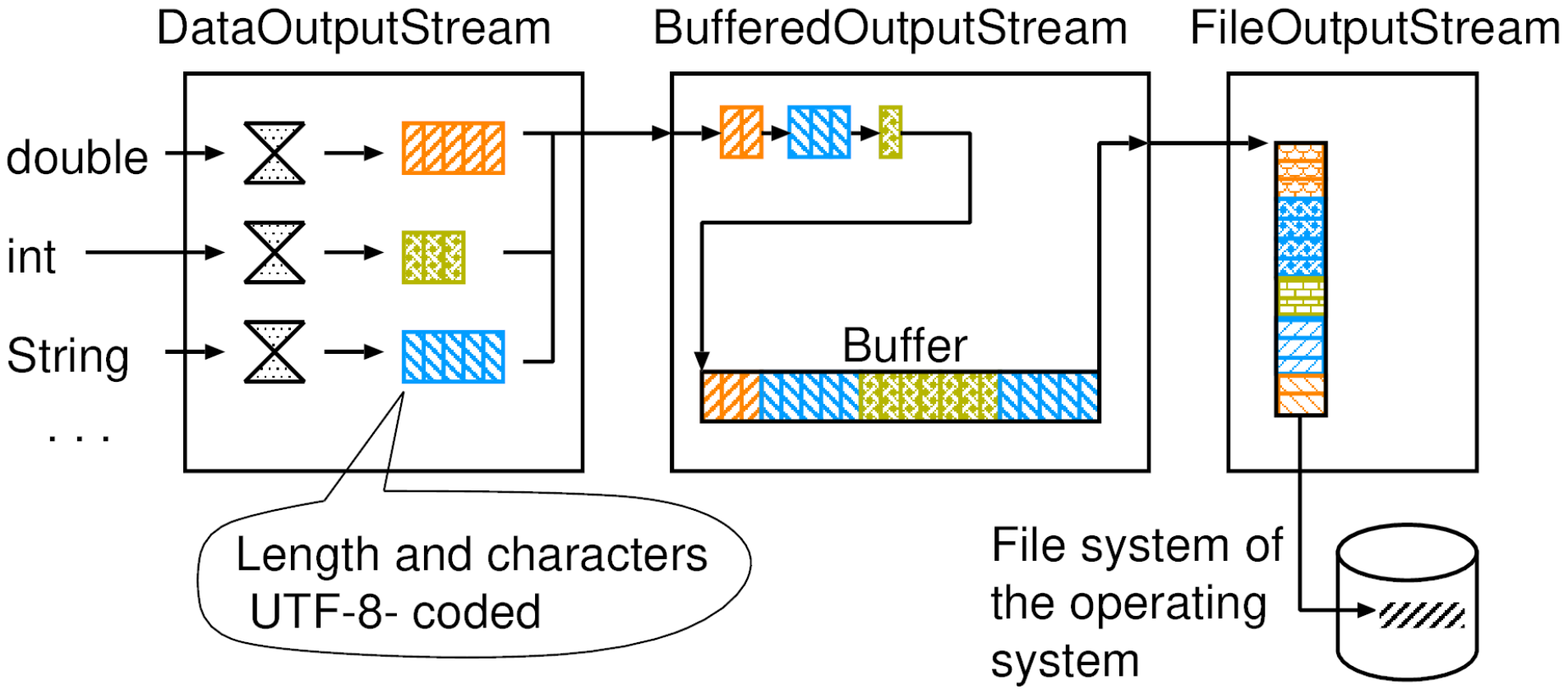
BufferedOutputStream bos = new BufferedOutputStream(fos);

DataOutputStream dos = new DataOutputStream(bos);
                   :
dos.writeDouble(3.1415); // writes 3.1415 into the file
```

Buffering is used to increase performance.

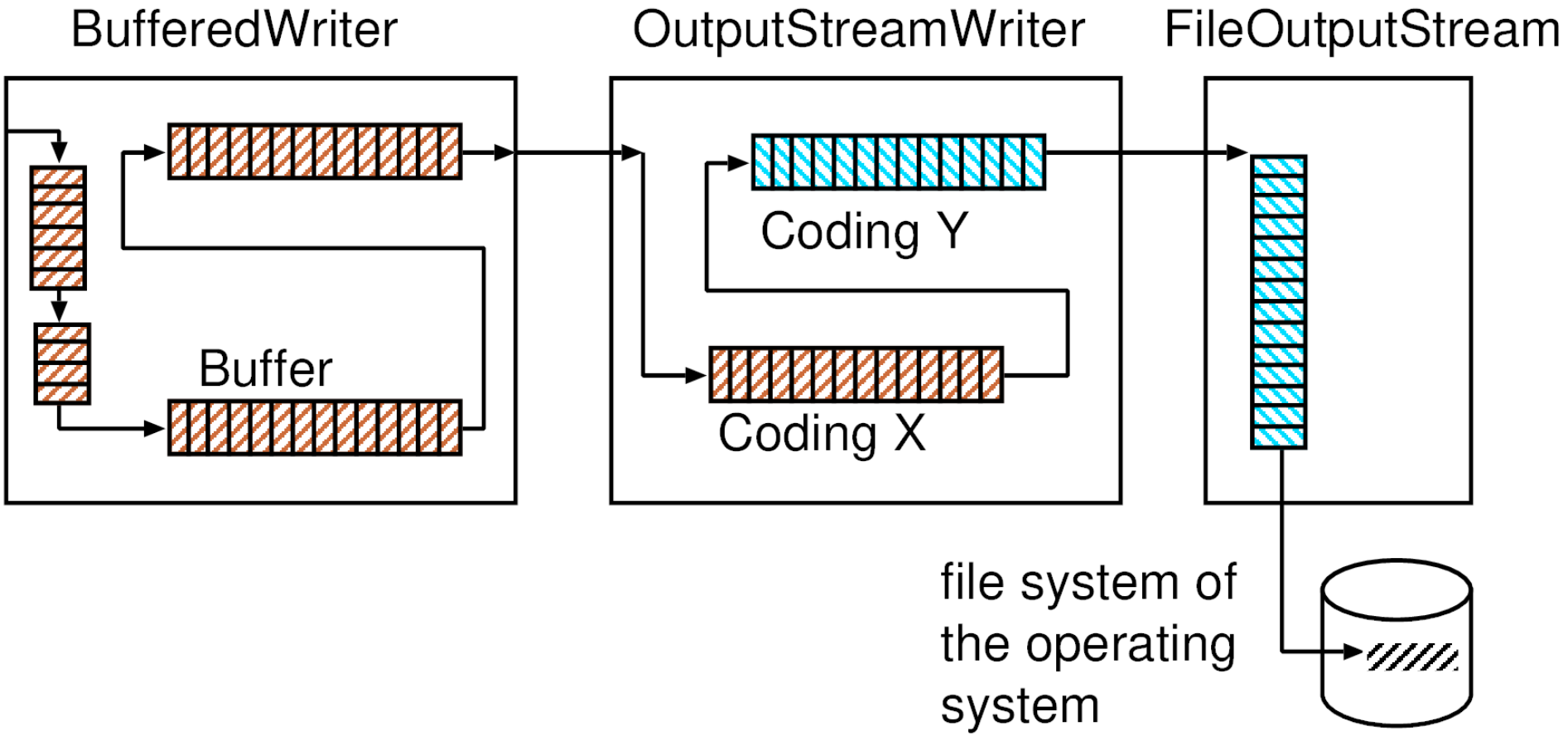
# File-I/O in Java II

Each class performs a single task.
Combination allows for flexible solutions.



# File-I/O in Java III

Basic streams are usually “decorated”, i.e. wrapped in an other stream class



# File-I/O in Java IV

Classes for writing data e.g.:

java.io.PrintWriter.print()

- Write primitive values to stream (here to file)
- File is readable by humans

```
FileOutputStream fos = new FileOutputStream("carFile.dat");
BufferedOutputStream bos = new BufferedOutputStream(fos);
PrintWriter prWriter = new PrintWriter(bos);
                  :
prWriter.print(basePrice);  // printf(); is also supported
prWriter.print(" ");
double ePrice = theEngine.getPrice();
prWriter.print(ePrice);
prWriter.print(" ")
```

# File-I/O in Java V

Classes for reading data out of text e.g.:

java.util.Scanner.nextXXX()

- Read primitive values from a (text) stream
- Create object from these values

```
FileInputStream fis = new FileInputStream("car....");
BufferedInputStream bis = new BufferedInputStream(fis);
inScanner = new Scanner(bis);
          :
double bPrice = inScanner.nextDouble();
double ePrice = inScanner.nextDouble();
int    ePower = inScanner.nextInt();
Engine anEngine = createRandomTypeEngine(ePrice, ePower);
```

# File-I/O in Java VI

Class for saving objects:

java.io.ObjectOutputStream

- Saves a complete object to a stream
- Object and all components must implement java.io.Serializable
  (or must be marked transient)

```
FileOutputStream fos = new FileOutputStream("car....");
BufferedOutputStream bos = new BufferedOutputStream(fos);
ObjectOutputStream cos = new ObjectOutputStream(bos);
       :
cos.writeObject(myCars[i])
```

Writes an object and all referenced objects to the stream

# File-I/O in Java VII

Class for reading objects:

java.io.ObjectInputStream

- Reads a complete object from a stream

```
FileInputStream fis = new FileInputStream("car....");
BufferedInputStream bis = new BufferedInputStream(fis);
ObjectInputStream cis = new ObjectInputStream(bis);
                    :
myCars[i] = (Car) cis.readObject();
```

Cast reference to proper type

# Java Serialisation I

- Maps a Java object to a sequence of bytes
  - Type of the object (type identifier)
  - Primitive members
  - Object references
    (Serialisation of referenced objects)
- Recursive process
  - Handles circular references

# Java Serialisation II

- Prerequisites for serializing an object
  - Implements java.io.Serializable
  - All (non transient) members are serializable:
    Primitive data type or
    reference to a serializable object

- Keyword **`transient`**
  excludes a member from serialization

# Java Serialisation III

Non serializable classes:

Classes with counterparts outside the JVM e.g.

- JFrame (class for a GUI window)
- Socket (network connection interface)
- Stream (interface to a file, socket, . . .)
- Thread
- . . . .

# File-I/O in Java VI

Streams should be closed:

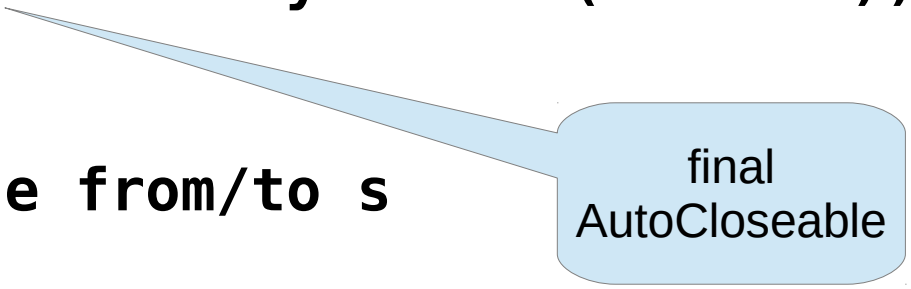
- Release resources of the runtime-system
- Output streams only:
  Flush data to the file-system/data-sink

```
FileInputStream fis = new FileInputStream("car....");
BufferedInputStream bis = new BufferedInputStream(fis);
ObjectInputStream cis = new ObjectInputStream(bis);
                    :
                    :
cis.close();
```

closes cis and nested streams

# try with resource

```
try(XyzStream s = new XyzStream( . . . )){
   while(){
      Read/write from/to s
   }
   // s.close() not required
} catch (IOException ioex){
}
```




Stream s is automatically closed when the program leaves the try-block.

# Java I/O Design Considerations

Where are the data stored?

- I/O direct to file: java.io.* . . . java.nio.*
- I/O to network: java.net.*
- I/O to data base: JDBC
  Java Data Base Connectivity
- Persistence managed by a “container” JEE

# Java I/O Design Considerations

File I/O data format:

- Text (PrintWriter / Scanner)
- Binary data (DataStream)
- Objects in binary form (ObjectStream, Serialization)
- Objects/Data in JSON-format (text)
- Mapping to data base

# Java I/O Design Considerations

Versioning is always an issue:

- adding a member / data field
- taking off a member
- changing a member (data type)

Always provide a version indicator in a file or message.

# Java Container Classes

Classes and Interfaces for storing objects

Some examples from java.util:

| Interface | Class | Class |
|---|---|---|
| Set | HashSet | TreeSet |
| List | ArrayList | LinkedList |
| Deque | ArrayDeque | LinkedList |
| Map | TreeMap | HashMap |

java.util contains quite a few more – pick what is appropriate for your application.

# Java Container Classes - List

- Access objects sequential or by an index
- Implements **`Iterable<V>`**
- Some methods of List:
  - **`public boolean add(V value)`**
  - **`public V get(int index)`**
  - **`public V remove(int index)`**
  - **`public boolean isEmpty()`**
  - **`public Iterator<V> iterator()`**

# Java Container Classes - Map

- Access Objects by a key
- Iterable via method `keySet()` or `values()`
- Some methods of Map:
  - `public V put(K key, V value)`
  - `public V get(Object key)`
  - `public V remove(Object key)`
  - `public boolean isEmpty()`

# Java Container Classes - Set

- Test if object is present
- Implements **`Iterable<E>`**
- Some methods of Set:
  - **`public boolean add(E e)`**
  - **`public boolean remove(Object o)`**
  - **`public contains(Object o)`**
  - **`public boolean isEmpty( )`**
  - **`public Iterator<V> iterator()`**

# Java Container Classes

If your code depends only on an interface the implementing class can be replaced without further changes.

| Interface | Class | Class |
|---|---|---|
| Set | HashSet | TreeSet |
| List | ArrayList | LinkedList |
| Deque | ArrayDeque | LinkedList |
| Map | TreeMap | HashMap |

# Generic Classes

Collection Classes are generic.

Definition of class ArrayList:

```
public class ArrayList<E> . . .

    public boolean add(E e) {
        . . . . .
        elementData[size++] = e;
        return true;
    }

    public E get(int index) {
        rangeCheck(index);
        return (E) elementData[index];
    }
```

Type variable: defines the type of hosted objects

slightly simplified

# Generic Classes

Collection Classes are generic:

```
ArrayList<Car> myCarList = new ArrayList<Car>();
                           new ArrayList<>();
```

Type of objects to host

Type can be omitted here

The variable **`myCarList`** is a reference to an ArrayList.

The ArrayList can hold references to objects of type **`Car`** (and subclasses of **`Car`**).

The methods **`get()`** and **`add()`** work only on objects of type **`Car`** (and subclasses).

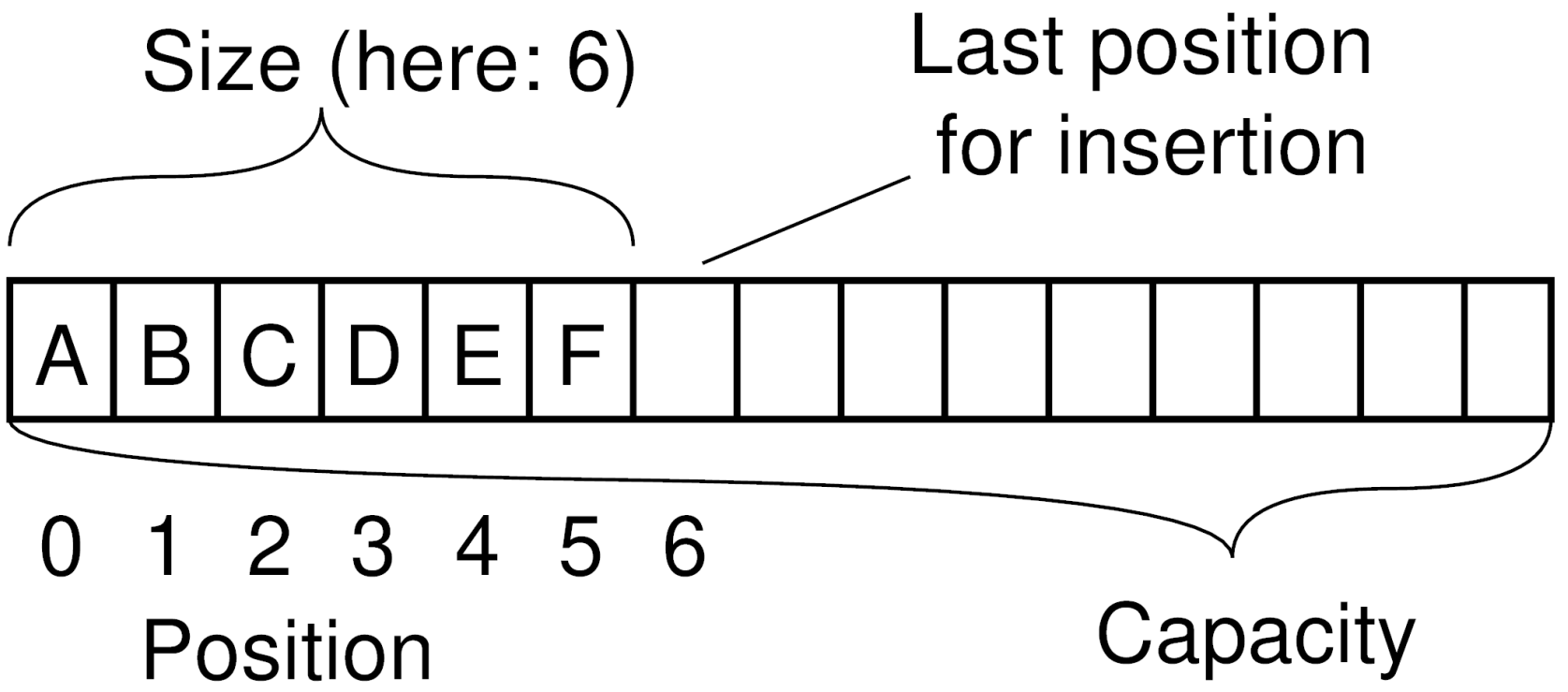
# ArrayList

- Holds objects in a specified order –
  - position may move
- ```
  ArrayList<DataObject> dataStore;
  dataStore  = new ArrayList<>();
  ```
- Methods
  - add(element), add(index, element)
  - get(index)
  - remove(index), remove(element)
  - . . . many more methods . . .
- Grows and shrinks silently as required
  ```

# ArrayList

Position for inserting Objects:

0 <= position <= size



# For-Loop for Iterable

## For-Loop for iterable Classes

```
Class XYZ implements Iterable
```

## Allows for:

```
XYZ myCollection = new XYZ();

for (XYZ anObject: myCollection){
    anObject...  // traverses all elements
}
```

Iterator still required for removing Elements.
Sets and Lists are iterable (among others).

# Map – HashMap<K,V>

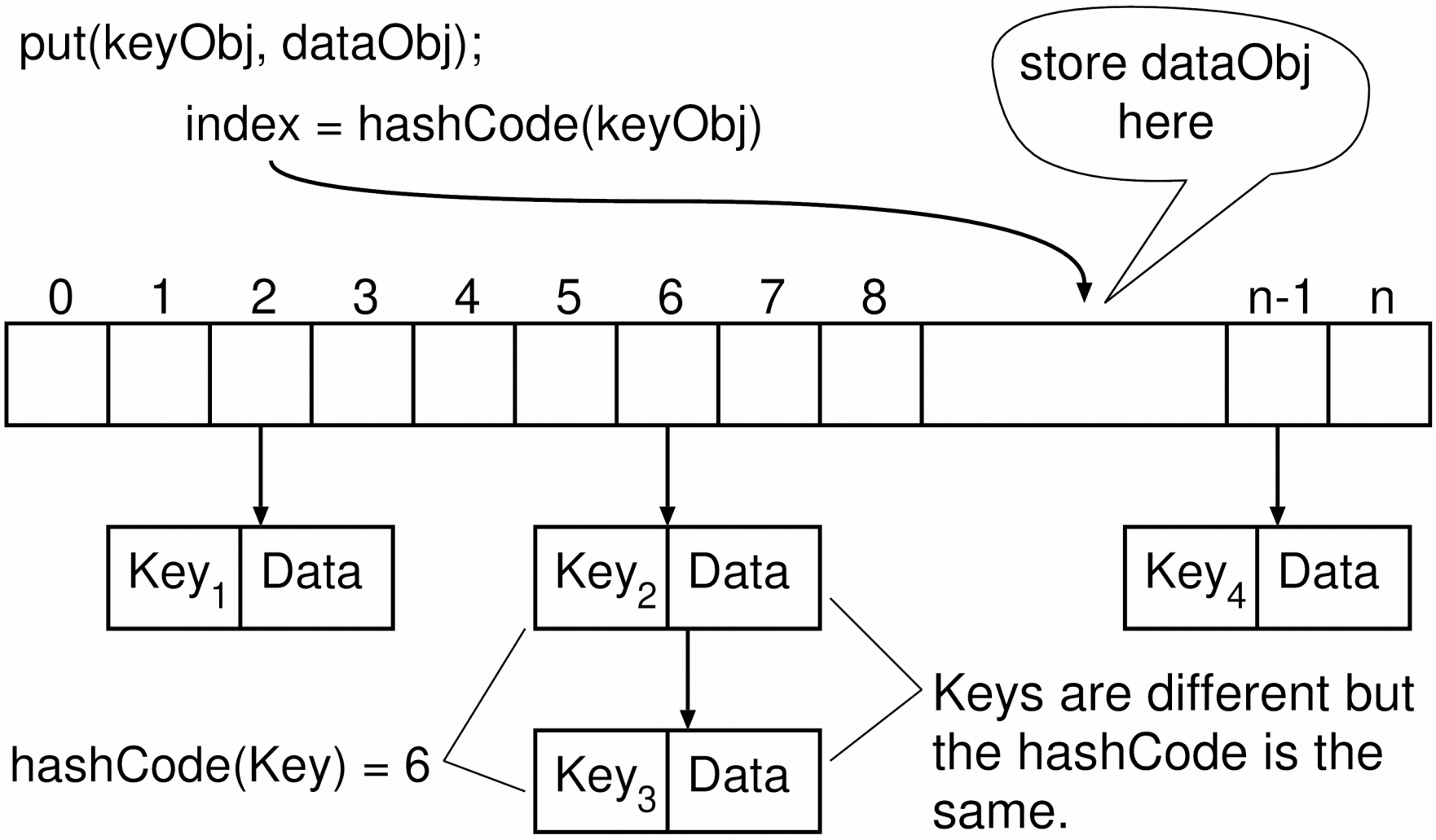
- Holds pairs <key, object>, keys must be unique
- **HashMap<Integer, DataObject> cache;**
  **cache = new HashMap<>();**
- Methods
  - **put(Object key, Object value)**, fast storage
  - **get(Object key)**, fast random access
  - **remove(Object key)**
  - **values()**, returns a collection of all objects
  - . . . many more methods . . .
- Grows and shrinks silently as required

# Map - HashMap

Hash-Function:

- Object ⟶ {0, … n}
- Maps values of a large range to values of a smaller range, suitable as indices for an array.
- Different objects should be mapped to different values (collisions may/will occur).
- In Java: Each object has method **`hashCode()`**; result can be trimmed using **`hashCode()%max`**

# Map - HashMap


put(keyObj, dataObj);
index = hashCode(keyObj)
store dataObj here
0 1 2 3 4 5 6 7 8 n-1 n
Key1 Data
Key2 Data
Key4 Data
Key3 Data
hashCode(Key) = 6
Keys are different but the hashCode is the same.

# Code Documentation - Javadoc

Start of Javadoc comment

```
/**
 * <h1>Base class for different types. . . </h1>
 * This class holds common attributes of . . .
 * <p>
 * <b>Note:</b> Once created only . . .
 *
 * @author martin zeller
 * @version 0.7
 */
```

- Tags in the source comments: @. . .
- javadoc generates HTML pages

# Javadoc

Some key-words for javadoc:

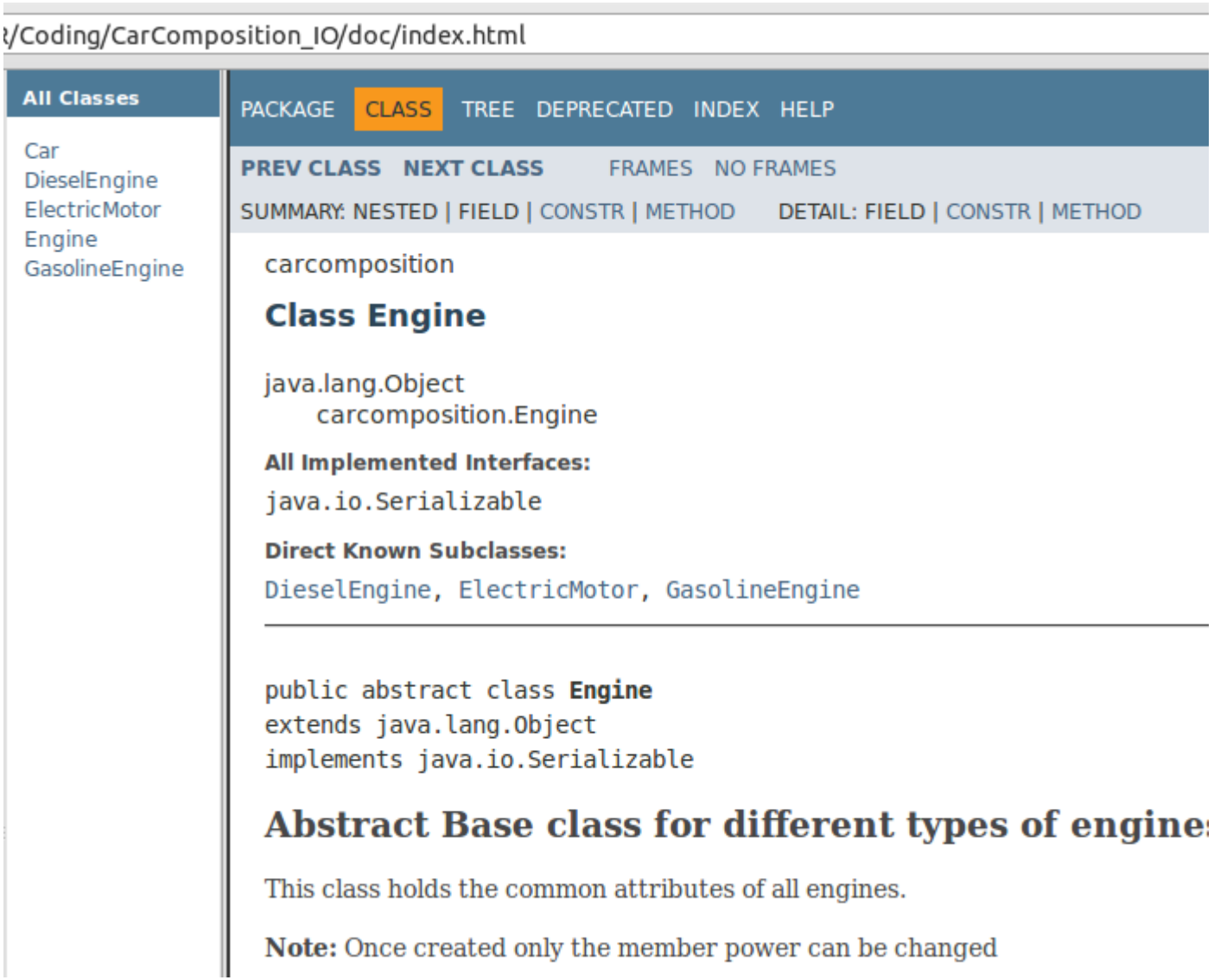
- **`@author`** (classes and interfaces only)
- **`@version`** (classes and interfaces only)
- **`@param`** (methods and constructors only)
- **`@return`** (methods only)
- HTML-Tags are possible: **`<p> <b> ... </b>`**

```
javadoc -sourcepath src -d doc carcomposition
```

Create HTML from comments

# Javadoc

Result:



# JNI - calling C-functions

(1) Write Java Class that uses C Code

(2) Create C/C++ header file

(3) Implement functions defined in header file

(4) Compile C file into library

(5) Run java program using the library

# JNI - calling C-functions

Executed when the class is loaded (usually at program start)

```
package jnitest;

public class SensorProxy {

   static {System.loadLibrary("sensors");}

   public static native
          int getSensorValueJni(byte sensorId);

   public static native
          int setActorValueJni(double throttle);

  . . .

}
```

Compile the Java-Class  ==> SensorProxy.class

# JNI – Create C Header File

```
javah jnitest.SensorProxy
                ==> jnitest_SensorProxy.h
-----------------------------------------

JNIEXPORT jint JNICALL
   Java_jnitest_SensorProxy_getSensorValueJni
                          (JNIEnv *, jclass, jbyte);

JNIEXPORT jint JNICALL
   Java_jnitest_SensorProxy_setActorValueJni
                         (JNIEnv *, jclass, jdouble);
```

Name of the C-function

Parameters

# JNI – Implement C Functions

```
jnitest_SensorProxy.c

----------------------------------------
#include "jnitest_SensorProxy.h"
          . . .
JNIEXPORT jint JNICALL
   Java_jnitest_SensorProxy_setActorValueJni
          (JNIEnv *jniEnvPtr,
           jclass jClass,
           jdouble actorValue){

      printf( "This is the C function  . . .  );
      printf( "   the value of parameter . . .);
      return (int) (actorValue * 100);
}
          . . .
```

# JNI – Implement C Functions

Depends on the c compiler. Another compiler may require different options.

```
gcc -I $JAVA_HOME/include \
    -I $JAVA_HOME/include/linux \
    -shared -fPIC \
    -Wl,-soname,sensors.so \
    -o libsensors.so jnitest_SensorProxy.c

   ==> libsensors.so
```

Path to libsensors.so

**Run program:**

```
java -Djava.library.path=/opt/Java/JniLib
     jnitest.JniTest
```

# Design Patterns

Example Singleton: “At most one instance”

Only private constructors

```
public class MediaAdmin {
   private static MediaAdmin singleInstance;
    . . .
   static MediaAdmin getMediaAdmin() {
        if (singleInstance == null) {
            singleInstance = new MediaAdmin();
        }
        return singleInstance;
    }
```

# Design Patterns - Singleton

Global access to one instance.

- Private constructors
- Private static member contains reference to instance
- Public method to get the instance
- The instance is created by the first call to the get-method

Variant:  A limited amount of objects may exist.

# Design Patterns

Example Observer: “Notify other objects”

Interfaces:

```
Observable
    register()
    deregister()
    notify()
```

Server

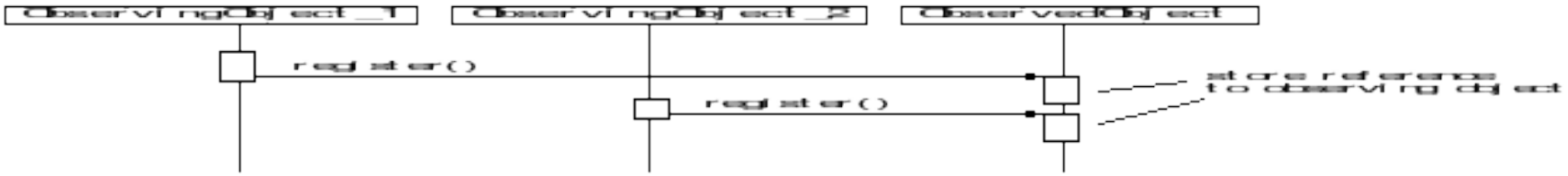
```
Observer
    notifyEvent()
```

Client

# Design Patterns

Example Observer:

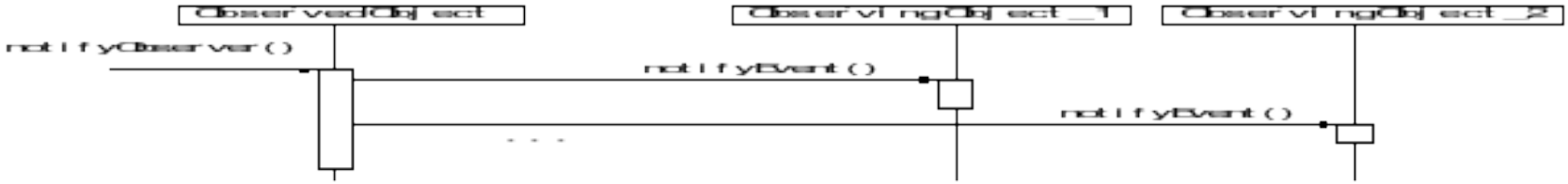
(1) register observing objects



# Design Patterns

Example Observer:

(2) Notify registered objects



# Lerning targets revisited

You can explain and use the oo features of Java

- Definition of classes
- Object creation
- Inheritance and composition
- Polymorphism, overriding, overloading
- Exception handling
- Generics (usage)
- Serialisation

# Lerning targets revisited

You can use and explore the Java RT-library

- File I/O, ArrayList, HashMap
- Java Native Interface

You can apply some design patterns

- Singleton
- Observer