Andre Strobel

Object Oriented Programming - Basics of JAVA

September 2018

Agenda

1. Expectations
2. Object Oriented Programming
3. Basics of JAVA
4. Summary

Expectations

- You know the basic concepts of computers and programming
- You will learn the basic concepts of an object oriented programming language – not necessarily get proficient at it.
- You participated successfully if you can distinguish the attribute “English” from the object “Englishman” and the method “Drink tea”.
- You will not be able to write graphical user interfaces in JAVA, but you know everything to easily learn how to do it.

If you get bored and rather read a book…

- Sams Teach Yourself Java in 21 Days

Object Oriented Programming
Von Neumann Computer

Central Processing Unit

Control Unit

Arithmetic/Logic Unit

Input Device

Output Device

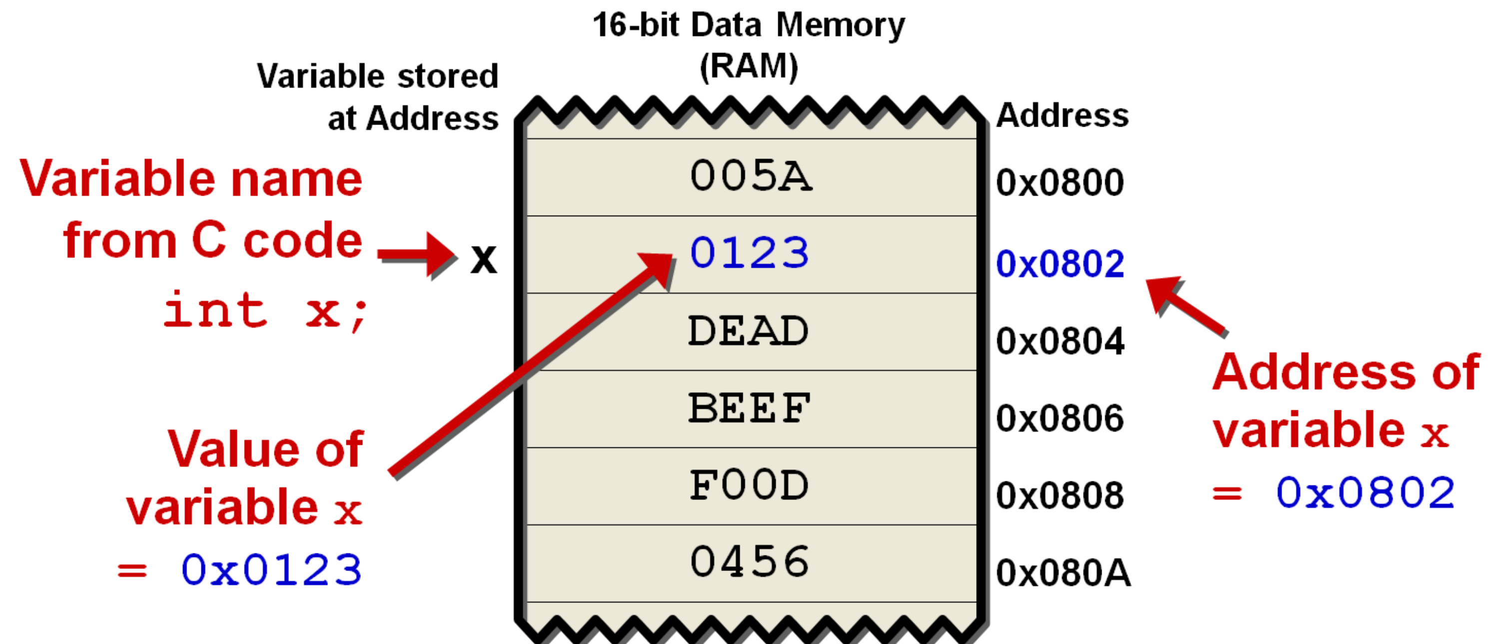
Memory Unit

Object Oriented Programming
Programming Languages

- Compiler versus Interpreter
  - A compiler translates the program into machine code before execution (pre-determined, predictable)
  - “Scripting” is mostly done via an interpreter language – the program is interpreted with existing machine code pieces at the time of execution
- Abstraction layers:
  - Machine Code: direct memory access (Assembler)
  - Spaghetti Code: one long code with simple variable types (original BASIC)
  - Structured: sub functions, structured variable types (PASCAL, C)
  - Objects: just wait a little … that’s what we will cover in the following (C++, JAVA)
  - Visual: signal flows (Simulink), state machines (Stateflow) etc.
  - Context?: kind of talking, not programming anymore (???)
  - … (when I retire or later)

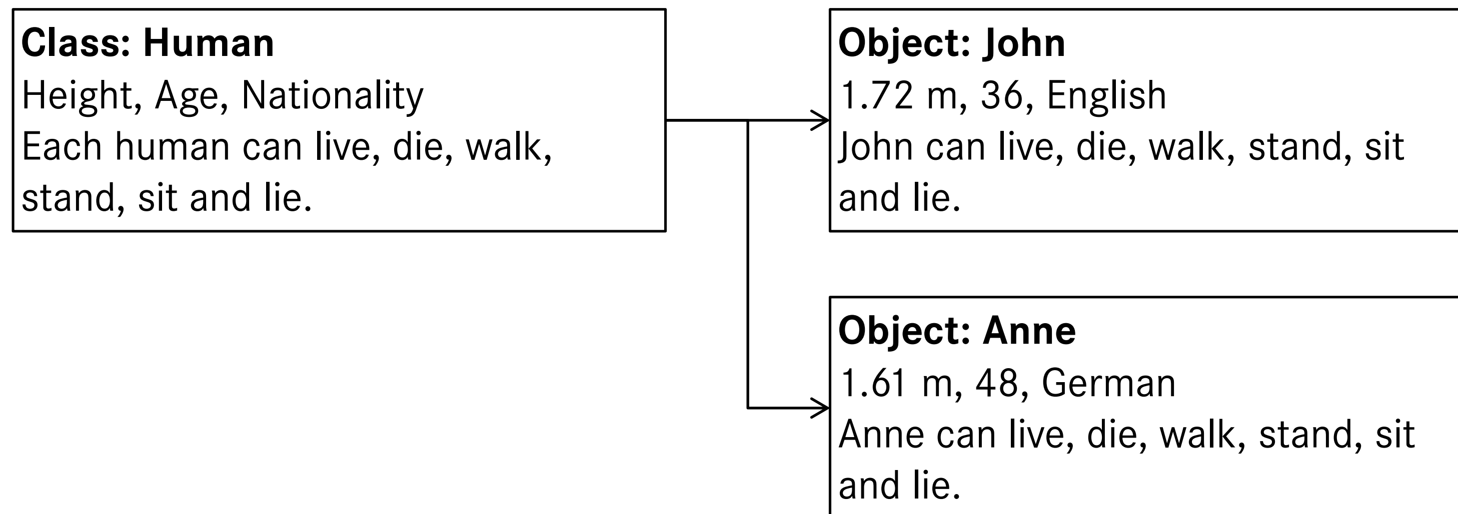
Object Oriented Programming
Variables in the Memory Unit

- Data Type: defines the memory structure of the variable
- Pointer: contains the address of a variable
- Instance: memory content for this variable



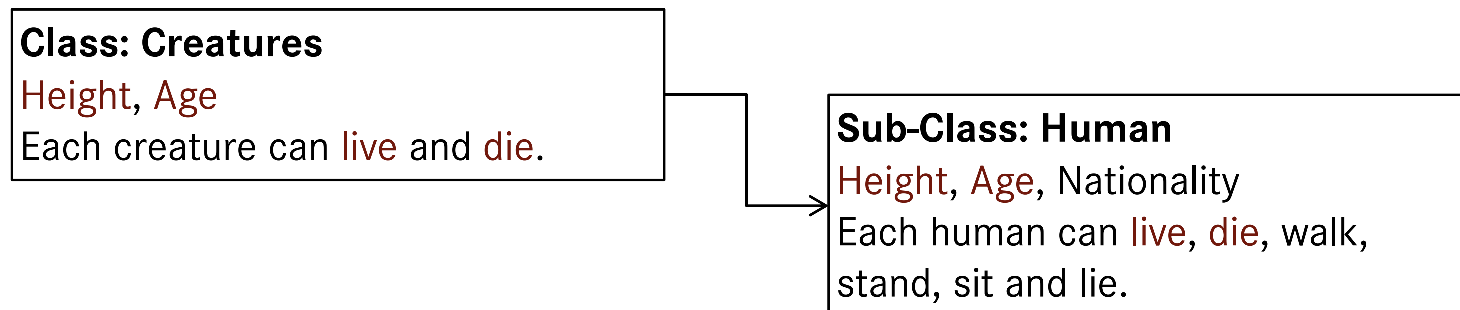
Object Oriented Programming
What is an object?

- In programming an object is a special data type
- An object consists of data describing the attributes of the object as well as methods describing the behavior of the object
- The architecture of an object is described as class. An object only exists if you create an instance of this class



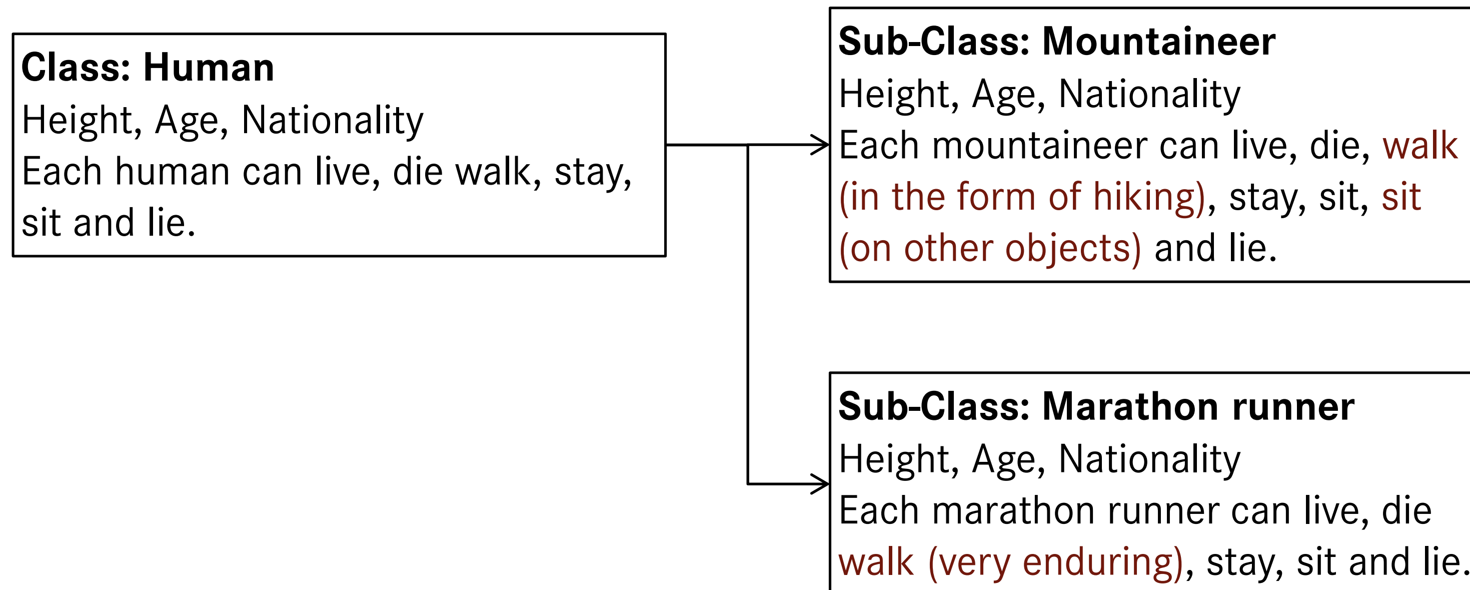
Object Oriented Programming Inheritance and Polymorphism/Reuse

- Inheritance is based on classes and not objects
- Each class can inherit the architecture and data from other classes (Inheritance)
- Each sub-class automatically owns the same attributes and behaviors of the parent classes (Polymorphism/Reuse)
- In case an object has several parents you must specify which parent class is dominant in inheriting attributes and behaviors



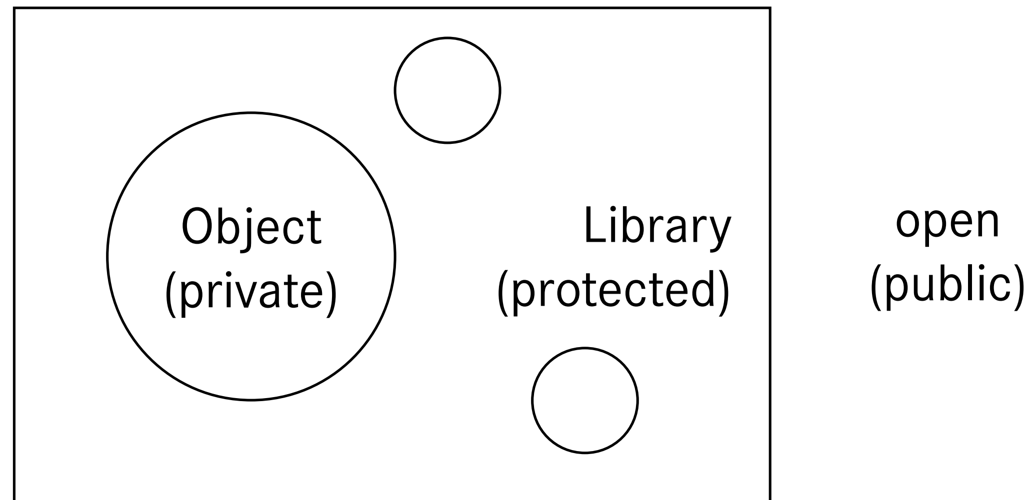
Object Oriented Programming
Overriding and Overloading

- Each sub-class can change the inherited behaviors (Overriding). The behavior itself remains, but it can be extended, changed or limited.
- Each behavior can depend on different input parameters (Overloading)



Object Oriented Programming
Encapsulation and Data Hiding

- Each class defines who can access its data and methods
- The access can be limited to the object itself (private), the object and its sub-classes as well as related objects in a library (protected) or be completely open (public)

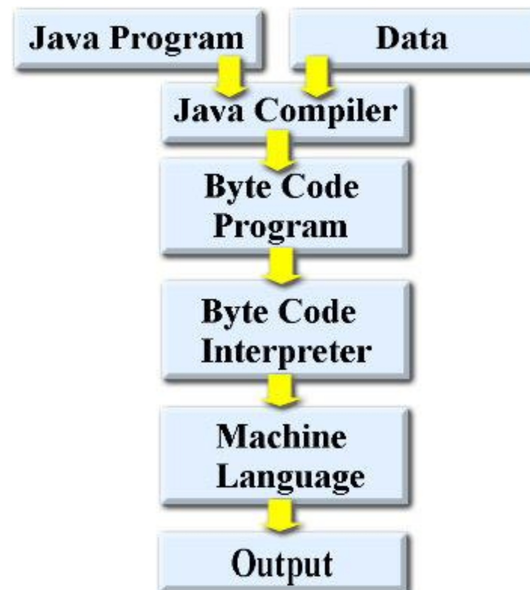


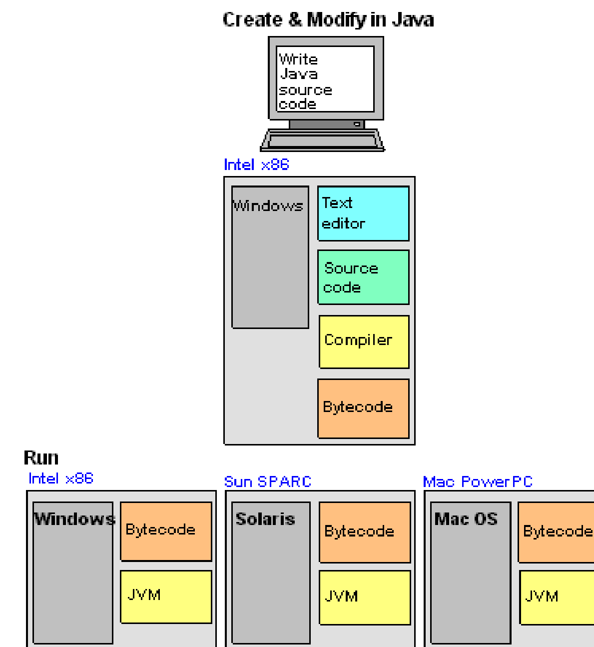
Basics of JAVA
Motivation

- Completely object oriented programming language
- Platform independent (Windows, Linux etc.)
- Memory management to avoid memory access conflicts
- Comprehensive supply of standard classes

Basics of JAVA
Platform Independency

- A compiler translates JAVA source code into Bytecode
- Bytecode can be run with a platform specific interpreter





Basics of JAVA
Variables and Basic Data Types

- Variables must be defined with a data type
  - Example: `VariableType myVariable;`
  - The following basic data types are allowed. The standard initialization value is mentioned in brackets. Numbers as `byte`, `short`, `int`, `long`, `float` and `double` (`0`), characters as `char` (`'\0'`), boolean values as `boolean` (`false`) as well as objects (`null`).
- Strings are defined using the class `String`
- Constants are marked with the command `final`
  - Example: `final float SHORTPI = 3.14;`
- Numerical values are interpreted as `int` by default. By adding `'L'` the value is interpreted as `long`. Octal and hexadecimal numbers are marked by a leading `'O'` and `'Ox'` respectively.
  - Example: `long myLongNumber = OxFFL;`

Basics of JAVA
Basic Syntax (1)

- Comments are started with `/*` and finished with `*/`. The remainder of a row can be a comment by starting the comment with `//`.
- The following basic arithmetic operators from other programming languages are available: `+`, `-`, `*`, `/` and `%` (modulus). You can increment or decrement a value by `++` and `--` respectively. If this operator is placed in front of a variable it will be adjusted before the value is used, otherwise afterwards.
  - Example:

```
int x = 0, y = 0, z = 0;
y = x++ + 1;
z = ++x + 1;
```

- Assignments can also be concatenated with the operator =
  - Example:

```
int x, y, z;
x = y = z = 1;
```

Basics of JAVA
Basic Syntax (2)

- You can increase/decrease a variable by the value on the right side of the operator using =+, =-
- The following basic comparison operators from other programming languages are available: ==, !=, <, >, <= and >=
- The following basic logical operators from other programming languages are available: &, |. Using && and || respectively prevents that the right side of the operator is being executed in case the result is already determined by the left side.
- Operators are executed in the known sequence and priority. Brackets have the highest priority.
- Strings can be concatenated using +
  - Example:

```
String myFirstName = "Michael";
String myName = "Smith";
String myFullName = myFirstName + " " + myName;
```

Basics of JAVA
Casting of Variables and Objects

- Casting is the exchange of data between variables with different data types
  - Example:

```
int myIntValue = 5;
float myFloatValue;
myFloatValue = (float) myIntValue;
```

- Casting can only be applied to objects that are related to each other. Values of exactly the same attributes will be transferred.
  - Example:

```
MySuperClass mySuperObject = new MySuperClass();
MySubClass mySubObject = new MySubClass();
mySuperObject = (MySuperClass) mySubObject;
```

Basics of JAVA
Statements

- In JAVA statements can be combined into blocks which are enclosed in braces
  - Example: `void myBlock() {}`
- Conditional statements are defined by `if` or `switch` blocks
  - Example 1:

```
if (args.length <= 0) {}
else if (args.length <= 1) {}
else {}
```

  - Example 2:

```
switch (args.length)
{
    case 0: break;
    case 1: break;
    default:
}
```

Basics of JAVA
Loops

- A `for` loop is used for a known number of iterations
  - Example:
    ```
    for (int vCount1 = 0; vCount1 < 10; vCount1++) {}
    ```
- A `while` loop (example 1) or `do` loop (example 2) is used for an unknown number of iterations
  - Example 1:
    ```
    int vCount1 = 0;
    while (vCount1 < 10) {vCount1++;}
    ```
  - Example 2:
    ```
    int vCount1 = 0;
    do {vCount1++;} while (vCount1 < (10 + 1));
    ```
- A loop can be left immediately with the `break` command
- A loop can immediately continue with the next iteration with the `continue` command

Basics of JAVA
Memory Management (1)

- In JAVA you cannot directly access memory addresses to avoid memory access conflicts. This restriction favors quality before execution speed.
- Memory is requested and will be occupied with the `new` command
  - Example: `MyClass myObject = new MyClass();`
  - A with `new` created object only contains a reference to the according location in the memory
- In other programming languages you must explicitly release the occupied memory after use. JAVA does this automatically. In order to do so the `garbage collection` function is regularly executed in the background.
- By deleting all references to an object you force the `garbage collection` function to free up the memory that the object occupied.
  - Example: `myObject = null;`

Basics of JAVA
Memory Management (2)

- Assigning an object to another object only copies its reference. Example 1 and example 2 do exactly the same. The memory that was occupied by `myObject2` in example 1 will automatically be released by the `garbage collection`.
  - Example 1:
    ```
    MyClass myObject1 = new MyClass();
    MyClass myObject2 = new MyClass();
    myObject2 = myObject1;
    ```
  - Example 2:
    ```
    MyClass myObject1 = new MyClass();
    MyClass myObject2;
    myObject2 = myObject1;
    ```

Basics of JAVA
Lists

- All variables can be defined as arrays. Example 1 and example 2 produce the same result.
  - Example 1 (not complete):

```
String[] weekDays = new String[7];
weekDays[0] = "Monday";
weekDays[6] = "Sunday";
```

  - Example 2: `String[] weekDays = {"Monday"; "Tuesday"; "Wednesday"; "Thursday"; "Friday"; "Saturday"; "Sunday"};`

- Analog you can create multidimensional arrays
  - Example (not complete):

```
int[][] myMatrix = new int[2][2];
myMatrix[0][0] = 1;
myMatrix[1][1] = 4;
```

Basics of JAVA
Defining Classes and Objects (1)

- A class can be defined with the `class` command
  - Example: `class MyClass {}`
- An object can be defined by instantiation of a class with the `new` command
  - Example: `MyClass myObject = new MyClass();`
- You can define data and methods inside a class
  - Example:

```
class MyClass
{
    int zahl = 0;
    void MyClass() {}
    void myMethod(MyOtherClass inOtherObject) {}
}
```

Basics of JAVA
Defining Classes and Objects (2)

  - The method `MyClass()` which has the same name as the object itself is called `constructor` and will automatically be called when a new object of this class is being created
  - You can also define the method `finalize()` which will automatically be called before the object is removed from the memory

- You can define other classes inside of a class
  - Example:

```
class MyClass
{
    class MyInnerClass {}
}
```

Basics of JAVA
Access Management (1)

- Access to variables, methods and classes is defined via the key words `private`, `protected` and `public` (see Encapsulation and Data Hiding)
  - There is only one difference between the standard access and the `protected` access. A sub-class that is not defined in the same package as the parent class gains access to `protected` variables, methods and classes of the parent class.
  - It's good practice that an object holds back most of its data using `private` variables, methods and classes. Access to the data is then granted via accessor methods. This allows to change how the data is implemented without changing the interfaces to the outside of the class (Reuse).

Basics of JAVA
Access Management (2)

- Example:

```
class MyClass
{
   private int myInteger;

   public int getMyInteger()
   {
      return this.myInteger;
   }

   public void setMyInteger(int inInteger)
   {
      this.myInteger = inInteger;
   }
}
```

Basics of JAVA
Access to Variables and Methods

- Access to variables and methods of an object follows the schema `myObject.myVariable` and `myObject.myMethod()` respectively
  - You can also access variables and methods of classes if they have been defined as `static`
- The current object is always named `this` within the same object. The parent object can be accessed with `super`.
- In principle you can access variables and methods within an object without using `this`, but this can lead to issues if another variable is defined with the same name within the object (not the class) which is allowed in principle.
- The own `constructor` can be called via `this()`. The `constructor` of the parent object can be called via `super()`.

Basics of JAVA
Defining and Calling Methods (1)

- Each method is defined following the schema `MyReturnClass myMethod(MyClass1 inObject1, MyClass2 inObject2).`
  - If the method does not return a value `MyReturnClass` is replaced with `void`
  - Somewhere within the method you need to define the return value following the schema `return myReturnObject`
  - Calling the method follows the schema `myMethod(inObject1, inObject2)`, i.e. without specifying the data types

Basics of JAVA
Defining and Calling Methods (2)

- The same method can be defined with different input parameters (Overloading). These methods must be defined with different number of parameters or data types of parameters. JAVA automatically selects the method that fits to how the method is being called.
  - Example:

```
int myAddition(int x, int y)
{
   return x+y;
}
int myAddition(int x, int y, int z)
{
   return myAddition(x+y, z);
}
```

Basics of JAVA
Call-by-Reference and Call-by-Value

- In principle JAVA only passes on references to variables and objects when methods are called with parameters. This means that the method operates on the original data and can change it (Call-by-Reference).
  — Example: `myMethod(inReference);`
- In case a method should not be able to change the original data you must create a copy of this data when you call the method (Call-by-Value). In the following example the class `MyClass` has a `constructor` that generates a copy of the object that is passed into this `constructor` and then returns a reference to this copy. This reference is then passed to the method `myMethod()`.
  — Example: `myMethod(MyClass inValue = new MyClass(inReference));`

Basics of JAVA
Converting Data Types

- The is a class for each basic data type (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Char` and `Boolean`)
  - In newer versions JAVA executes automatic Casting between these classes and the related basic data types
- Converting a `String` object to the variable of the basic data type `int` is done with a `static` method of the class `Integer`
  - Example: `int myInteger = Integer.parseInt("65000");`
- Converting a variable of the basic data type `int` to a `String` object is also done with a `static` method of the class `Integer`
  - Example: `String myString = Integer.toString(65000);`

Basics of JAVA
"Hello world!"

- A class that can be called as a program must have the method `public static void main(String args[])`
  - Example:

```
class MyProgram
{
   public static void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

  - The class `System` has the sub-class `out` and its method `println()` sends a `String` object to the output console

Basics of JAVA
Inheritance, Abstract Classes and Interfaces (1)

- A sub-class can be defined using the identifier `extends` (Inheritance)
  - A sub-class inherits all variables and methods of the parent class
  - Each class has at least the class `Object` as parent class (which is really confusing: class/object)
  - It is not allowed in JAVA to have several parent classes. JAVA uses Interfaces for this (covered later).
  - If a class is marked as `final` you cannot define a sub-class based on it
  - Example:

    ```
    class Human extends Creature {}
    ```
- Methods that a sub-class inherits from its parent class can be changed (Overriding)
  - If a method is marked as `final` you cannot change the inherited version (no Overriding)

Basics of JAVA
Inheritance, Abstract Classes and Interfaces (2)

— Example:

```
class Parent
{
    int speed = 0;
    void move() {this.speed = 5;}
}

class Child extends Parent
{
    void move() {this.speed = 10;}
}
```

Basics of JAVA
Inheritance, Abstract Classes and Interfaces (3)

- Abstract classes define the architecture that a class must have at least
  - You can define an abstract class by using the `abstract` identifier
  - You cannot create an object based on an abstract class
  - Abstract classes can contain abstract methods
  - The implementation of the architecture is done in sub-classes (which are not abstract)

Basics of JAVA
Inheritance, Abstract Classes and Interfaces (4)

- Example:

```
public abstract class Anything
{
   final int MYFIXEDNUMBER = 5;
   abstract int getMyFixedNumber();
}

public class Something extends Anything
{
   int getMyFixedNumber()
   {
      return this.MYFIXEDNUMBER;
   }
}
```

Basics of JAVA
Inheritance, Abstract Classes and Interfaces (5)

- An Interface is a collection of attributes and behaviors
  - —A class can be based on several Interfaces, but all of them must define completely different attributes and behaviors
  - —Example:

```
public interface Doctor
{
    void heal();
}

public interface Player
{
    void gamble();
}
```

Basics of JAVA
Inheritance, Abstract Classes and Interfaces (6)

— Example (cont.):

```
class Person implements Doctor, Player
{
   void heal() {}
   void gamble() {}
}
```

- Interfaces can be extended based on other Interfaces

— Example:

```
interface SuperDoctor extends Doctor
{
   void cut();
}
```

Basics of JAVA
Using and Creating Packages (1)

- Packages are the libraries of JAVA
- The basic name of a package is based on an Internet address
  - `http://www.daimler.com/` leads to `com.daimler`
- The files of a Package are located in a sub-folder in the folder that is known to JAVA as `Classpath`. The name of this sub-folder is defined by the Package name.
  - `com.daimler` leads to `com/daimler`
- A Package can be linked to the code using the command `import`
  - You can either link all Packages of a folder (*) or just a specific Package (not recursive)
  - If you try to link two Packages with the same name from different folders, JAVA issues an error
  - Example:

```
import java.util.*;
import com.daimler.dtna.dpwt.makeeverythingeasy;
```

Basics of JAVA
Using and Creating Packages (2)

- A Package is a collection of classes that are linked to this Package
  - The classes of a Package must be defined inside the folder of the Package
  - The Package must be declared at the top of each related JAVA source code file that contains the definition of related classes
  - Example:

```
package com.daimler.dtna.dpwt.makeeverythingeasy;
```

Basics of JAVA
Compiling and Executing JAVA

- JAVA source code is stored in a `*.java` file. The location of the file is defined by the related Package.
- A JAVA source code file must be compiled with `javac`
  - The `Classpath` variable must be defined correctly so that all necessary Packages can be found
  - The result of the compilation is JAVA Bytecode for a class that is stored in a `*.class` file
  - Example: `> javac MyClass`
- After this you can execute the JAVA Bytecode
  - Example: `> java MyClass`

Basics of JAVA
Data Structures (1)

- The Package `java.util.*` provides special classes in order to store more complex data structures with a variable number of elements within this data structure
- The class `BitSet` stores a defined number of Boolean values
  - You can access the Boolean values with the methods `set()`, `clear()` and `get()`
  - When you create an instance of the class you can specify the number of Boolean values it will contain. The method `size()` returns the number of currently stored Boolean values within this object.
  - Example (not complete):

```
BitSet myBitSet = new BitSet(1);
myBitSet.set(1);
myBitSet.clear(1);
```

Basics of JAVA
Data Structures (2)

- A varying number of equal objects can be stored in a list using the `Vector` class
  - You can access the elements of this list with the methods `add()`, `set()`, `remove()`, `removeElement()`, `get()`, `contains()` and `indexOf()`
  - When you create an instance of the class you can specify the number of elements and optionally the increment for adding additional elements in case you go beyond the initial number of elements. The method `size()` returns the number of currently stored elements within this list. Further available methods are `setSize()`, `trimToSize()` and `capacity()`.
  - During the definition of the list you can specify the data type of the stored objects in the form `Vector<String> stringVector = new Vector<String>()` where the data type is defined within <>. This allows JAVA to check the syntax of the source code better and sometimes this is even necessary.

Basics of JAVA
Data Structures (3)

— Example:

```
Vector<String> stringVector = new Vector<String>(5, 5);
stringVector.add("Michael");
stringVector.add("Anne");
stringVector.add("Mary");
stringVector.remove(2);
stringVector.removeElement("Michael");
stringVector.trimToSize();
String myString = (String) stringVector.get(0);
myString = (String)
   stringVector.get(stringVector.indexOf("Anne"));
```

Basics of JAVA
Data Structures (4)

- The Interface `Iterator` allows simple navigation through a list
  - The Interface `Iterator` forces the methods `hasNext()` and `next()`
  - The class `Iterator` implements the Interface `Iterator`
  - Example:

```
for (Iterator vCount1 = stringVector.iterator();
    vCount1.hasNext(); )
{
    String currentString = (String) vCount1.next();
    System.out.println(currentString);
}
```

- Additionally JAVA offers the data structures Stacks (class `Stack`) with `push()` and `pop()` as well as Hash Tables (class `Hashtable`) which allow access to the individual elements via keys and not only element numbers.

Basics of JAVA
Exceptions (1)

- Exceptions are used to catch unavoidable errors during program execution
  - Each class is required to react to each possible Exception. Otherwise, JAVA issues an error during the compilation of the source code. The class can either pass the Exception on to its parent class or it can deal with it itself.
  - Each Exception is a sub-class of the class Exception
  - Example: `class NoFileException extends Exception {}`
- Throwing of Exceptions
  - Exceptions are thrown within a method with the command `throw`, e.g. if the method realizes that the desired file is not even existing
  - Example:

```
NoFileException nfe = new NoFileException();
throw nfe;
```

Basics of JAVA
Exceptions (2)

- Passing on Exceptions
  - A class can pass on Exceptions by listing them during the class definition
  - Example: `class FileList throws NoFileException {}`
- Dealing with Exceptions
  - A class can deal with an Exception by catching the throwing object or the throwing of an Exception within a `try-catch` block
  - The `try` block contains the source code that can generate the Exception
  - The `catch` block contains the source code which is being executed in case the Exception occurs
  - The `finally` block contains all the statements which are executed in any case, e.g. closing a file that was opened inside the `try` block
  - You can throw the same or other Exceptions again within the `catch` block

Basics of JAVA
Exceptions (3)

— Example:

```
try
{
   FileList.getFile();
}
catch (NoFileException nfe)
{
   // do something intelligent
   System.out.println("Error: "+nfe.getMessage());
   throw nfe;
}
finally
{
}
```

Basics of JAVA
Assertions

- Assertions check values during the execution whether they make sense and if they are correct
  - Assertions must be turned on during the execution of the JAVA Bytecode
  - Example: `> java -ea MyClass`
- An Assertion can be defined anywhere in the source code using the command `assert`
  - In case the to be tested statement results in a false value an `AssertionError` Exception is thrown
  - You can either deal with this Exception or pass it on all the way to the JAVA Interpreter
  - Example:

```
assert myValue > 0 : "Wert ist größer als Null.";
```

Basics of JAVA
Threads (1)

- Threads allow to execute pieces of the source code in parallel (Multitasking). This allows e.g. to use more than one CPU core for the execution of the program.
- A Thread in JAVA is a special object based on an object who's class implements the Interface `Runnable`
  - The object `MyObject` becomes a Thread by calling it with `Thread MyThread = new Thread(MyObject)`
  - The Interface `Runnable` defines several methods like `start()` and `run()`. The method `start()` automatically calls the method `run()` and is normally used in conjunction with a `constructor`.
  - It makes sense to add the variable `finished` which mentions whether a Thread is finished or not

Basics of JAVA
Threads (2)

— Example:

```
public class CountALot implements Runnable
{
    private long beginNumber, endNumber;
    private Thread runner;
    public boolean finished = false;

    CountALot(long inBeginNumber, long inEndNumber)
    {
        beginNumber = inBeginNumber;
        endNumber = inEndNumber;
        if (runner == null)
        {
            runner = new Thread(this);
            runner.start();
        }
```

Basics of JAVA
Threads (3)

— Example (cont.):

```
    }

    public void run()
    {
        for (long vCount1 = this.beginNumber;
            vCount1 <= this.endNumber; vCount1++)
            {System.out.println(
                Integer.toString(vCount1))}
        this.finished = true;
    }
}
```

Basics of JAVA
Threads (4)

- A Thread can be used in any program
  - The Thread normally starts when the `constructor` is called
  - The program waits until all Threads are finished
  - Example:

```
public class MyThreadedProgram
{
    public final long BEGIN = 0;
    public final long END = 999999;

    public static void main(String[] args)
    {
        boolean complete = false;
        CountALot counter = new CountALot(BEGIN, END);
```

Basics of JAVA
Threads (5)

— Example (cont.):

```
            while (!complete)
            {
                complete = true;
                if (!counter.finished)
                {
                    complete = false;
                }
                try {Thread.sleep(1000);}
                catch (InterruptedException ie)
                {
                    // do nothing
                }
            }
        }
    }
```

Basics of JAVA
Streams for Input and Output (1)

- JAVA uses Streams to exchange data with other media (file systems, networks)
  - In order to use Streams you must load the Package `java.io.*`
  - All Streams can be passed through several Filters
  - Streams typically throw the Exceptions `FileNotFound`, `EOFException` and `IOException`
- The abstract classes `InputStream` and `OutputStream` define the basic form of a Bytestream
  - The byte wise access to the stored data is managed with the classes `FileInputStream` and `FileOutputStream`

Basics of JAVA
Streams for Input and Output (2)

— Example:

```
FileInputStream fileInput = new
   FileInputStream("test.dat");
do
{
   int myInput = (Integer) fileInput.read();
   System.out.print(Integer.toString(myInput) + " ");
} while (myInput != -1);
System.out.println("");
fileInput.close();
```

Basics of JAVA
Streams for Input and Output (3)

- The data of Bytestreams can be changed and passed on using Filters
  - The classes `BufferedInputStream` and `BufferedOutputStream` are used for buffered access to a Stream
  - The classes `DataInputStream` and `DataOutputStream` are used for accessing a Bytestream with integer and floating point numbers

Basics of JAVA
Streams for Input and Output (4)

— Example:

```
final String MYFILENAME = "test.dat";
final double MYDOUBLENUMBER = 0;
FileOutputStream fileOutput = new
   FileOutputStream(MYFILENAME);
BufferedOutputStream bufferOutput = new
   BufferedOutputStream(fileOutput);
DataOutputStream dataOutput = new
   DataOutputStream(bufferOutput);
dataOutput.writeDouble(MYDOUBLENUMBER);
bufferOutput.flush();
dataOutput.close();
fileOutput.close();
```

Basics of JAVA
Streams for Input and Output (5)

- The classes `FileReader` and `FileWriter` are used for accessing a Stream that contains text
  - Example:

```
FileReader fileText = new FileReader("test.txt");
do
{
   int myInput = fileText.read();
   System.out.print((char) myInput);
} while (myInput != -1);
System.out.println("");
fileInput.close();
```

- The data of a Stream that contains text can be changed and passed on using Filters

Basics of JAVA
Streams for Input and Output (6)

— Example:

```
final String MYFILENAME = "test.txt";
FileReader fileText = new FileReader(MYFILENAME);
BufferedReader bufferText = new BufferedReader(fileText);
do
{
   String myLine = bufferText.readLine();
   if (myLine != null)
   {
      System.out.println(myLine);
   }
} while (myLine != null);
bufferText.close();
fileText.close();
```

Basics of JAVA
File Access

- The class `File` in the Package `java.io.*` is used to access a file or a folder
  - The class `File` throws the Exceptions `IOException` and `SecurityException`
  - The platform independent file separator is `File.separator`
  - The method `exists()` returns a Boolean value for whether or not the file exists. After that you can ask for the size of the file in bytes using the method `length()`.
  - The method `delete()` deletes a file
  - The methods `getName()` and `getPath()` return the name and path of the file
  - The method `mkdir()` creates a sub-folder. The method `isDirectory()` returns a Boolean value for whether or not a file is actually a folder. The method `listFiles()` lists the content of a folder.
  - Example: `File myFile = new File("test.txt");`

Basics of JAVA
Serializing Objects (1)

- Objects can be directly written into a file using Streams (Serializing). After that they can also be read from the same file using a Stream.
  - The class of the related object must implement the Interface `Serializable`
  - If a variable is marked `transient` within a class it is excluded from Serializing (e.g. passwords)
  - Example:

```
class MyObject implements Serializable {
   private transient String passWord = "";
   private int myNumber = 0;
   public void setPassWord(String inPassWord) {
      this.passWord = inPassWord;
   }
   public int getMyNumber() {return myNumber;}
}
```

Basics of JAVA
Serializing Objects (2)

— Example (cont.):

```
public class ObjectWriter {
   public static void main(String[] args) {
      MyObject myObject = new MyObject();
      try
      {
         FileOutputStream fileOutput = new
            FileOutputStream("MyObject.obj");
         ObjectOutputStream objectOutput = new
            ObjectOutputStream(fileOutput);
         objectOutput.writeObject(myObject);
         objectOutput.close();
      } catch (IOException ioe) {
         System.out.println("Error: " + ioe.toString());
      }
   }
}
```

Basics of JAVA
Serializing Objects (3)

— Example (cont.):

```
public class ObjectReader {
   public static void main(String[] args) {
      try
      {
         FileInputStream fileInput = new
            FileInputStream("MyObject.obj");
         ObjectInputStream objectInput = new
            ObjectInputStream(fileInput);
         MyObject myObject = (MyObject)
            objectInput.readObject();
         objectInput.close();
      } catch (IOException ioe) {
         System.out.println("Error");
      } catch (StreamCorruptionException sce) {
         System.out.println("Error");
      }}}
```

Basics of JAVA
Graphical User Interfaces with Swing

- The Packages `java.awt.*` and `javax.swing.*` help to create extremely powerful graphical user interfaces in a very simple way
  - —The description of how to create such graphical user interfaces is beyond the scope of this training

Basics of JAVA
Reaction to Events (1)

- Objects can react to user inputs using Event Listeners that are defined as Interfaces
  - ActionListener, AdjustmentListener, FocusListener, ItemListener, KeyListener, MouseListener, MouseMotionListener, TextListener, WindowListener
  - Example:

```
public class MyWindow extends JFrame implements
    ActionListener, MouseListener {}
```

- In case a class implements an Event Listener the Event Listener is typically activated inside the constructor
  - Example:

```
JButton myButton = new JButton("O.K.");
myButton.addMouseListener(this);
```

Basics of JAVA
Reaction to Events (2)

- The reactions are defined within the methods that are defined by the Event Listener
  - Example:

```
public void mouseClicked(MouseEvent inMouseEvent)
{
    // do something as expected
}
```

Summary

- Now you know everything to become a JAVA expert
- But if you want to use JAVA successfully, you still need to practice, practice and practice
- I highly recommend to intensify your knowledge of JAVA by reading the book Sams Teach Yourself Java in 21 Days from Roger Cadenhead and Laura Lemay
  - It describes all aspects of JAVA as well as the creation of graphical user interfaces
  - As most of the examples contain mistakes, your highest attention while reading this book is ensured
  - A corrected version can be borrowed from André Strobel ;-)
- So all you need is 21 days to read a book and practice ;-)

Amendments (1)

- The statements for importing Packages must be defined at the beginning of the JAVA source code file before the first class definition
  - Example:

```
import java.util.*
class MyClass {}
```

- Interfaces can contain variables, but they are automatically `public`, `static` and `final` and therefore openly available constants. It's optional to mention `public`, `static` and `final` for these variables.

Amendments (2)

- The period operator can be used in at least three cases
  - —Defining the hierarchy in order to call methods of sub-classes and sub-packages etc.
  - —Selecting a variable or method of a specific class or object
  - —Nesting of the two cases before in case the return value of a method is again an object who’s method should be called
- Overloading and Overriding is often referred to as Polymorphism