



**Fachhochschule  
Bonn-Rhein-Sieg**  
*University of Applied Sciences*

**Fachbereich Informatik**  
*Department of Computer Science*

**Master of Autonomous Systems 2007/2008**

**JAVA Tutorial:**  
**A tutorial on JAVA Object Orientation**

Sebastian Blumenthal, Geovanny Giorgana

Subject:  
Agile Software Team Techniques

## Table of Contents

1 Introduction.....	1
2 Class/objects.....	2
2.1 Concepts.....	2
2.1.1 Overloading.....	6
2.2 Summary.....	7
3 Inheritance.....	8
3.1 Concept.....	8
3.2 Overriding.....	11
3.3 Abstract classes and methods .....	13
3.4 Constructors and inheritance.....	15
3.5 Class "Object" .....	15
4 Polymorphism .....	17
4.1 Concept .....	17
4.2 Polymorphism and abstract classes.....	18
5 Information hiding .....	19
5.1 Concept.....	19
5.2 Access modifiers.....	19
5.3 Access modifiers for methods and fields.....	20
6 Interfaces.....	22
6.1 Concept.....	22
6.2 Definition of Interfaces.....	23
6.3 Interfaces as types.....	23
6.4 Interfaces as specifications.....	24
6.5 Abstract class vs Interface.....	24
7 Threads .....	25
7.1 Concept.....	25
7.2 Handling.....	25
7.3 Synchronization.....	28
8 Appendix .....	29
8.1 Package.....	29
8.2 Primitive types supported by Java.....	30

## 1 Introduction

Java is high-level programming language which supports object oriented programming. It is similar to C++ but there is a significant difference: Java uses a so called **Virtual Machine** (VM). The Virtual Machine can be seen as a software layer between binary code and real hardware. Java compilers produce *byte code* for the Virtual Machine instead of compiling it to the hardware like compilers for other programming languages. The Virtual Machine behaves like a simulated computer. In fact whenever Java *byte code* is executed the Virtual Machine interprets the code and translates it into corresponding code for the actual hardware.

This concept has some interesting advantages: first of all the program is portable. That means a compiled Java program (*byte code*) can run on **every** Virtual Machine regardless of the underlying hardware or operating system. Only the Virtual Machine must be installed on the target system to run *byte code*. Another aspect of the Virtual Machine is that a programmer doesn't have to care about hardware issues like low-level input/output operations or memory allocation. Even cleaning up free memory space is done within Java ("Garbage Collector").

Unfortunately the concept of the Virtual Machine bears a drawback: performance. One can easily imagine that a code which has to be compiled on-the-fly while a program is running is slower than a precompiled machine code. But in practice the performance issue is today a minor problem due to considerable power in "standard" PCs, especially when a program primarily computes (non-CPU-intensive) I/O requests.

This tutorial is intended to serve as an introduction to the powerful world of object orientation. It addresses programmers with basic background about programming in Java, but little knowledge about object-oriented programming. In the following chapters basic items about object orientation like dealing with objects, information hiding, inheritance and polymorphism are discussed. Interfaces in Java and thread handling is also presented.

The main topics are explained by a brief description followed by an example and finalized by a deeper explanation.

## 2 Class/objects

Object-oriented programming (OOP) is used to model some parts of the world in a computer by using objects that appear in the problem domain. By making use of OOP we can divide the whole complex program into well-defined parts – the objects- that can be built and examined separately. This technique allows us to focus in only one part of the problem ignoring the details of the other parts. These two advantages are known as the principles of *modularization* and *abstraction*, respectively.

### 2.1 Concepts

The objects used in the model can be categorized, and a class describes, in an abstract way, all objects of a particular kind.

For example, if we wanted to model a traffic simulation, we would have to deal with cars. Is car a class or an object? To answer we may consider a few questions that help us to make a decision.

Which color has a car? How fast can it go? Where is it right now?

We cannot answer these questions unless we talk about one specific car. The reason is that the word “car” in this context refers to the *class* car.

Now, if we direct this same questions to one specific car, for example, “the old green car that is parked at home in my garage”, we can answer the questions above. That car is green, it doesn’t go very fast, and it is at home in my garage. Now we are taking about an *object*, about one particular example of a car.

You can have as many objects of one class as you want. All the objects of the *same* class will be described by using the same attributes, but the value of those attributes for each object may be different. For instance, every car will have its own color, maximal speed, current position, size, etc. In contrast, objects of a *different* class may have different fields. A railway, for example, would have the attribute “number of wagons”, while a bicycle would have the attribute “diameter of the wheels”.

The object attributes are referred to as *fields*. The number, types, and names of fields are defined in a class, not in an object. When an object of a class is created, the object will automatically have these fields. Although the fields are defined in a class, the values of

these fields are stored in the object.

We can communicate with objects and manipulate the fields of an object by invoking methods on them. Objects usually do something if we invoke a method. The methods, similarly to fields, are defined in the class of the object. As a result, all objects of a given class have the same methods, and due to methods are invoked on objects, to know which object to change when invoking a method is clear.

We shall now explain the way classes are defined and objects are created in Java by using an example. This example models a Book Storage System in a basic way. We define a class – *Book* – that allows us to create as many objects as we want to represent different Books of a Bookshop, for instance, and it is possible to store and to visualize the title and price of each book at the moment it is created or later on.

The source code for the implementation of the class *Book* is shown below and explained further after presenting it.

#### Book Class:

```
package ClassObjectsExample;

/**
 * The class Book represents a Book object. Information
 * about the book is stored and can be retrieved.
 *
 * @author Geovanny Giorgana and Sebastian Blumenthal
 * @version 2008-01-13
 */

public class Book {

    /* field declaration */
    private String Title;
    private double Price;

    /* Initialize and object with default values */
    public Book() {
        Title = "Field_empty";
        Price = 0;
    }

    /* Initialize and object with predefined values */
    public Book(String theTitle, double thePrice){
        Title = theTitle;
        Price = thePrice;
    }

    /* Accessor method to return the Title of the book */
}
```

```
public void getTitle()
{
    System.out.println("Title: "+Title);
}

/* Accessor method to return the Price of the book */
public void getPrice()
{
    System.out.println("Price: "+Price+ " Euro");
}

/* Mutator method to set the Title of the book */
public void setTitle(String Title)
{
    this.Title= Title;
}

/* Mutator method to change the Price of the book */
public void setPrice(double Price)
{
    this.Price= Price;
}

/* Method to print the Characteristics of the book */
public void displaydata()
{
    System.out.println("Book: "+Title);
    System.out.println("Price: "+Price+ " Euro\n");
}
}
```

The first line of code corresponds to the declaration of the package where the class *Book* corresponds (the reader can see at the appendix (chapter 8) to know more details about packages). Then, as mentioned above, the class *Book* is created to allow us to define Book objects. Each one of those Book objects will have two attributes -*Title* and *Price*- . We have two ways to construct or initialize the objects, one is using default values, signaling that the field *Title* is still empty and assigning a zero to the field *Price*, and the second way consists in assigning values defined by the user. The first way can be used simply to reserve data in memory for a new book whose attributes will be specified in the future. In addition, we have four methods to access the fields of the objects, whether to change or to retrieve the current value of their fields. The methods used to change the state of an object are known as “mutator methods” and the methods to retrieve information about the state of an object are known as “accesor methods”. Finally, we use the method *displaydata* to print the value of the fields of an object to the text terminal.

What follows is to show the way how distinct objects from the class *Book* are created and how to mutate and retrieve the state of those objects. The main class containing the main function that is used to realize the just mentioned looks as follows.

## Main Class:

```
package ClassObjectsExample;

/**
 * The class Bookshop has the main method where two objects of
 * the class Book are created, SoftEng and DistSyst. A change
 * of price is realized by using its mutator method. The results
 * are displayed on screen.
 *
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */

public class Bookshop {

    public static void main(String[] args) {

        /* The objects SoftEng and DistSyst are created, the former
         * does not have a detailed initialization while the latter
         * is detailed from the beginning.
         */
        Book SoftEng = new Book();
        Book DistSyst = new Book("Distributed Systems", 50);

        /* Display the data of both books */
        SoftEng.displaydata();
        DistSyst.displaydata();

        /* Set the correct values for the first object */
        SoftEng.setTitle("Software Engineering");
        SoftEng.setPrice(45.00);

        /* Display again the details of the first book to see
         * the change in the values of its attributes.
         */
        SoftEng.getTitle();
        SoftEng.getPrice();
    }
}
```

First, two objects are created -*SoftEng* and *DistSyst*-, the object *SoftEng* is created but initialized with default values because no parameters are passed to its constructor. The other object is initialized with values that are passed at the moment it is created, so the constructor of this object knows what values to assign to the fields Title and the Price of this book. Then the state of the object is printed to the text terminal by using the method *displaydata* of each object. The text seen in the text terminal looks as follows

```
Book: Field empty
Price: 0.0 Euro
```

```
Book: Distributed Systems
Price: 50.0 Euro
```

We can see that the state of each order is printed in the same order as it is required in the main function, and for the object `SoftEng` we note that the default values were assigned to its fields and for the object `DistSyst` the values passed through the parameters of its constructor were assigned. Now, suppose information about the first book is given to the user, surely he would want to change the attributes of this object and he can do it by using the mutator methods `setTitle` and `setPrice` provided in the definition of the class `Book` (see code below).

```
/* Set the correct values for the first object */
SoftEng.setTitle("Softwate Engineering");
SoftEng.setPrice(45.00);
```

After modifying the title and the price of the book the user could want to see in the text terminal that the attributes of this object were correctly updated. To see these changes, the accessor methods `getTitle` and `getPrice` can be used (see code below). We can use the method `displaydata` to see both attributes of the objects, however, the methods `getTitle` and `getPrice` make possible to see the state of only one field of the object without displaying the state of the other field.

```
/* Display again the details of the first book to see
 * the change in the values of its attributes.
 */
SoftEng.getTitle();
SoftEng.getPrice();
```

The text terminal shows the changes when both accessor methods are called.

```
Title: Softwate Engineering
Price: 45.0 Euro
```

### 2.1.1 Overloading

Our class `Book` contains two constructors `-Book()` and `Book(String theTitle, double thePrice)-`, but why?. This is because a class may contain more than one constructor, offering more than one way to create an object. For our example, the user can either reserve a place in memory for a new book without knowing the exact title and correct price of this book or create a new book in the system and assign the exact title and correct price at the moment he creates the book.



What distinguishes a constructor from others is its set of parameters (which is also called *signature*). Our first constructor does not receive parameters while the second receives the title and the price. Remember that the constructor of a class is called every time a new object of this class is created. This is known as *Overloading* a constructor or method.

## **2.2 Summary**

In this section we learned the basic principles of object-oriented programming, how classes are declared, objects are created, how methods can be used to access or mutate fields of objects and the concept of overloading a constructor or a method.

## 3 Inheritance

### 3.1 Concept

Programmers are lazy. They don't want to write new code when a previous module already serves the needed core elements. Copy and past is not an option so we need something else: *inheritance*. With inheritance we can use an existing class and extend it with new features like new methods or variables (attributes). This concept avoids code duplication which helps to maintain software. Consider a module that has duplicates. When a change in the software must be done this would apply to all modules. Identifying and changing the code in the correct places is very fragile to errors.

Example:

```
package InheritanceExample;

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class MusicPlayer {

    public int songNumber; //just an example

    public MusicPlayer() {
        //initialize e.g. display
    }

    public void play() {
        //do something meaningful...
    }

    public void stop() {
        //do something meaningful...
    }
}
```

MusicPlayer is the existing “father” or “super” class. The next two inheritate from “Music-Player”:

```
package InheritanceExample;

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class Walkman extends MusicPlayer {
```

```
    /* Object variables of MusicPlayer are implicit available
    * because of inheritance.
    */

    /* constructor */
    public Walkman() {
        //init Walkman...
    }

    public void ejectCassette() {
        //do something meaningful...
    }

    /* Methods from MusicPlayer are implicit available because of
    * inheritance.
    */
}
```

```
package InheritanceExample;
```

```
/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class IPod extends MusicPlayer {

    /* Object variables of MusicPlayer are implicit available
    * because of inheritance.
    */

    /* constructor */
    public IPod() {
        //init IPod...
    }

    public void searchSong(String query) {
        //do something meaningful...
    }

    /* Methods from MusicPlayer are implicit available because of
    * inheritance.
    */
}
```

### Main class:

```
package InheritanceExample;
```

```
/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class MusicListener {
```

```

/**
 * @param args
 */
public static void main(String[] args) {
    /* use music players: */

    /* generic music player */
    MusicPlayer myPlayer = new MusicPlayer();

    myPlayer.play();
    // enjoy music ;-)
    myPlayer.stop();

    /* use a Walkman */
    Walkman myWalkman = new Walkman();

    myWalkman.play(); //inherited from MusicPlayer
    // enjoy music ;-)
    myWalkman.stop(); //inherited from MusicPlayer
    myWalkman.ejectCassette();

    /* use an iPod */
    iPod myIPod = new iPod();

    myIPod.play(); //inherited from MusicPlayer
    // enjoy music ;-)
    myIPod.searchSong("Nothing else matters");
    // enjoy another song
    myIPod.stop(); //inherited from MusicPlayer
}
}

```

To enable inheritance the keyword **extends** must be used. It is followed by the class to be inherited. In Java only **one** super/father class is allowed (single inheritance). All methods and object variables from the father class are available to the “children” (as long as they are not **private** (see chapter 5)). Of course children could be also father classes to other classes – so a whole inheritance *hierarchy* can be created. In our example we can utilize an ULM class diagram<sup>1</sup> to make to the hierarchy more clear (compare Illustration 1).

---

1 For more details take a glance at the UML tutorial

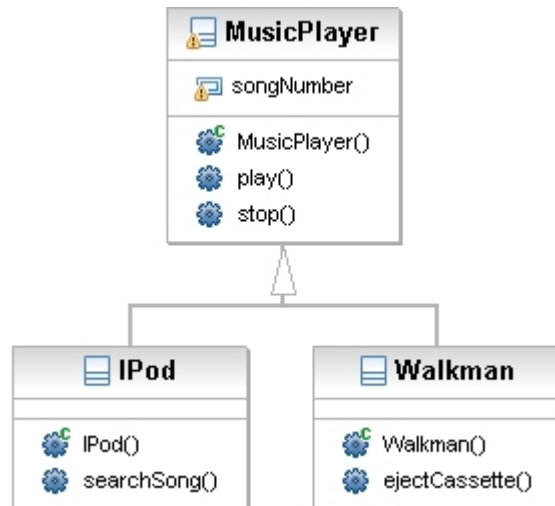


Illustration 1: Inheritance hierarchy (created by eclipse UML plugin)

## 3.2 Overriding

Sometimes inherited methods in a child class work/behave different so the super-class method must be overridden. For example the stop button for a MP3Player has a different behavior: when it is pressed longer than three seconds it turns off the whole device:

```

package OverridingExample;
import InheritanceExample.MusicPlayer; //reuse old stuff

/**
 * @author sblume2s
 */
public class MP3Player extends MusicPlayer {

    public int songNumber; //overrides variable from MusicPlayer

    /* constructor */
    public MP3Player() {
        //init MP3Player...

        /* accessing overridden variables */
        this.songNumber = 0; /* "this" addresses
                             * elements from this class
                             */

        super.songNumber = 0; /* "super" addresses
                             * elements from father/super
                             * class
                             */
    }

    /* override stop method from father class because
     * it works different (e.g. holding the stop button
    
```

```

        * longer than three seconds turns off the hole device)
        */
    public void stop() {
        /* do something different than in the father class
        * (of course "father method" stop can be accessed via
        * super.stop() if needed)
        */
        //measure how long button was pressed...
    }

    /* Remaining methods from MusicPlayer are implicit available
    * because of inheritance.
    */
}

```

### Main class:

```

package OverridingExample;

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class MusicListener2 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        /* use MP3Player players: */

        MP3Player myMp3Player = new MP3Player();

        myMp3Player.play();
        myMp3Player.stop(); // use overridden method
    }
}

```

Overriding a method or variables just done by writing a new method/variable in the children class with the same name. To access an overridden element use **this** and to access the original element use **super**. (Cascading calls like `super.super.x` are not allowed that means it is impossible to access a father's father (grandfather :-)) method/variable) Default is **this**.

When a method/variable is overridden in an inheritance hierarchy the hierarchy "tree" is scanned bottom up and the first found overridden method/variable is used.

### 3.3 Abstract classes and methods

In some cases it can be useful that a child class is forced to override a method. For example the father class models an operating system control. Every operating system shall have the ability to be turned off but how this turning off mechanism works is operating system dependent. So we could use an abstract class:

```
package AbstractClassExample;

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public abstract class OperatingSystemControl {

    /* constructor */
    public OperatingSystemControl() {
        // init this module
    }

    /* abstract method */
    abstract public void turnOff(); /* this method MUST
                                     be overridden in
                                     children classes */
}
```

```
package AbstractClassExample;

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class WindowsControl extends OperatingSystemControl {

    /* constructor */
    public WindowsControl() {
        //init module
    }

    /* override method from father class;
     * this is mandatory because turnOff is abstract within
     * OperatingSystemControl
     */
    public void turnOff() {
        /* turn off mechanism for a Windows system is
         * implemented here
         */
    }
}
```

```
package AbstractClassExample;

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class LinuxControl extends OperatingSystemControl {

    /* constructor */
    public LinuxControl() {
        // init module
    }

    /* override method from father class;
     * this is mandatory because turnOff is abstract within
     * OperatingSystemControl
     */
    public void turnOff() {
        /* turn off mechanism for a Linux system is
         * implemented here
         */
    }

}
```

### Main class:

```
package AbstractClassExample;

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class OperatingSystemControlTest {

    public static void main(String[] args) {
        //OperatingSystemControl myOS = new OperatingSystemControl();
        //abstract classes can not be initialized!
        LinuxControl myLinux = new LinuxControl();
        WindowsControl myWindos = new WindowsControl();

        myLinux.turnOff();
        myWindos.turnOff();

    }

}
```

`OperatingSystemControl` is an abstract class therefore the keyword **abstract** is put in front of **class**. **abstract public void turnOff();** creates an abstract method that must be overridden. Note that this statement ends with `;` instead of some kind of block indicated by brackets: `{}`.

In this example the father class doesn't know about internal workings about the turn off



mechanism. It is implemented in the child classes. In fact they are in a certain way substitutable with the help of polymorphism (refer to chapter 4).

As shown in `OperatingSystemControlTest` an abstract class can **never** be initialized.

### 3.4 Constructors and inheritance

Within Java it is possible to address a constructor method of a father class:

```
package ConstructorInheritanceExample;

import InheritanceExample.MusicPlayer;

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class IPod2 extends MusicPlayer {

    /* constructor of child */
    public IPod2() {
        super(); //explicit call of father class constructor
        /* this must be done as first statement;
        * here it is optionally because super without
        * any parameters is fit in automatically by the
        * compiler
        */
    }

    //...
}
```

Whenever an object is created (**new** operator) a constructor is called. If inheritance is used the constructor of the class the object created of is called (like normal classes). But implicitly or directly the constructor of the father class is also called. Too do so directly use **super()**; as first statement in the constructor. If the constructor of the father class has parameters they can be passed e.g. by **super(int example1, boolean example2)**; If **super** is not used the compiler fills it in automatically. (As a consequence the constructor with no parameters is invoked of the super class).

### 3.5 Class "Object"

If no inheritance is used a class implicitly inherits from the class "*Object*". Either a class inherits from *Object* or it belongs to a self programmed inheritance hierarchy. But the "root" node of such a hierarchy also inherits from *Object*. As a consequence every class directly

or indirectly inherits from *Object*. It something like a “master” class. The class *Object* delivers some interesting features like cloning objects or get a textual representation. For more information refer to the Java documentation.

## 4 Polymorphism

### 4.1 Concept

Polymorphism is about the ability that inheritance can be used to easily substitute classes with children classes.

Main Class:

```
package PolymorphismExample;

import InheritanceExample.*; //reuse old stuff

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class MusicListerner3 {

    public static void main(String[] args) {
        /* create players */
        Walkman myWalkman = new Walkman(); //normal variant
        MusicPlayer myPolyWalkman = (MusicPlayer) new Walkman();
        //polymorph variant
        /* because of inheritance "Walkman" has all the needed
         * methods (and variables) to behave like ("simulate")
         * MusicPlayer...
         */

        IPod myIPod = new IPod(); //normal variant
        MusicPlayer myPolyIPod = new IPod(); //polymorph variant

        /* use Walkman */
        myWalkman.play();
        myWalkman.stop();
        myWalkman.ejectCassette();

        /* use second Walkman object created with the help of
         * polymorphism
         */
        myPolyWalkman.play();
        myPolyWalkman.stop();
        //myPolyWalkman.ejectCassette();
        /* doesn't work because MusicPlayer has no
         * ejectCassette() method;
         * although a child is created only the methods of
         * the used type (father class MusicPlayer) can be used;
         */

        /* use IPod */
        myIPod.play();
        myIPod.searchSong("Nothing else matters");
        myIPod.stop();
    }
}
```

```

        /* use second IPod object created with the help of
        * polymorphism
        */
        myPolyIPod.play();
        //myPolyIPod.searchSong("Nothing else matters");
        /* doesn't work because MusicPlayer has no
        * searchSong(String name) method;
        */
        myPolyIPod.stop();
    }
}

```

As shown in the previous example, classes can have instances from their sub/child classes: `MusicPlayer myPolyWalkman = new Walkman();` `Walkman` is a child class of `MusicPlayer` and can serve all the methods/variables of `MusicPlayer`. That is why this statement is not an error. Only methods/variables from `MusicPlayer` can be used. It can be seen as a typecast from a `Walkman` to a `MusicPlayer` object. In fact it is a typecast because all classes in Java are types! The mentioned statement can be written as an explicit typecast: `MusicPlayer myPolyWalkman = (MusicPlayer) new Walkman();`

The instance of a type can be a child class or an arbitrary descendant class from an inheritance hierarchy.

## 4.2 Polymorphism and abstract classes

The reader may now wonder: why do we use polymorphism? Wouldn't it have been easier in the example of section 4.1 to use `MusicPlayer myPlayer = new MusicPlayer();` instead of `MusicPlayer myPolyWalkman = new Walkman();` because we can not use the functions of the child class. The answer is yes. It was just an example to show how it basically works. But now consider a little modification: the `MusicPlayer` is rewritten into an abstract class (see section 3.3). As a result `MusicPlayer myPlayer = new MusicPlayer();` wouldn't work because it is impossible to create an object of an abstract class. The only solution to create an object of type `MusicPlayer` is to utilize polymorphism: `MusicPlayer myPolyWalkman = new Walkman()` or `MusicPlayer myPolyWalkman = new IPod();`

## 5 Information hiding

### 5.1 Concept

Information hiding is a principle that states that internal details of a class's implementation should be hidden from other classes. It ensures better modularization of an application.

In many object-oriented programming languages the internal of a class -its implementation- are hidden from other classes. There are two main reasons: first, a programmer should *not need to know* the internals; second, a class should *not be allowed to know* the internals of another class.

The first principle *-need to know-* is related with abstraction and modularization. If it were necessary to know of all classes we need to use, we would never finish implementing large systems.

The second principle *-not being allowed to know-* has to do with modularization, but in a different context. The private section of one class is not visible to any other class. This ensures that one class doesn't depend on the implementation of another class. It allows us to make improvements or fix bugs in one class without making changes in other classes as well. This issue is known as *coupling*: if changes in one part of a program do not make it necessary to also make changes in another part of the program, this is known as weak coupling or loose coupling, which is good.

### 5.2 Access modifiers

Access modifiers are the key words *public*, *private* and *protected* at the beginning of field declarations, method and constructor signatures. For example:

```
//field declaration
private String Title;
private double Price;

//Constructors

public Book() {
    ...
}

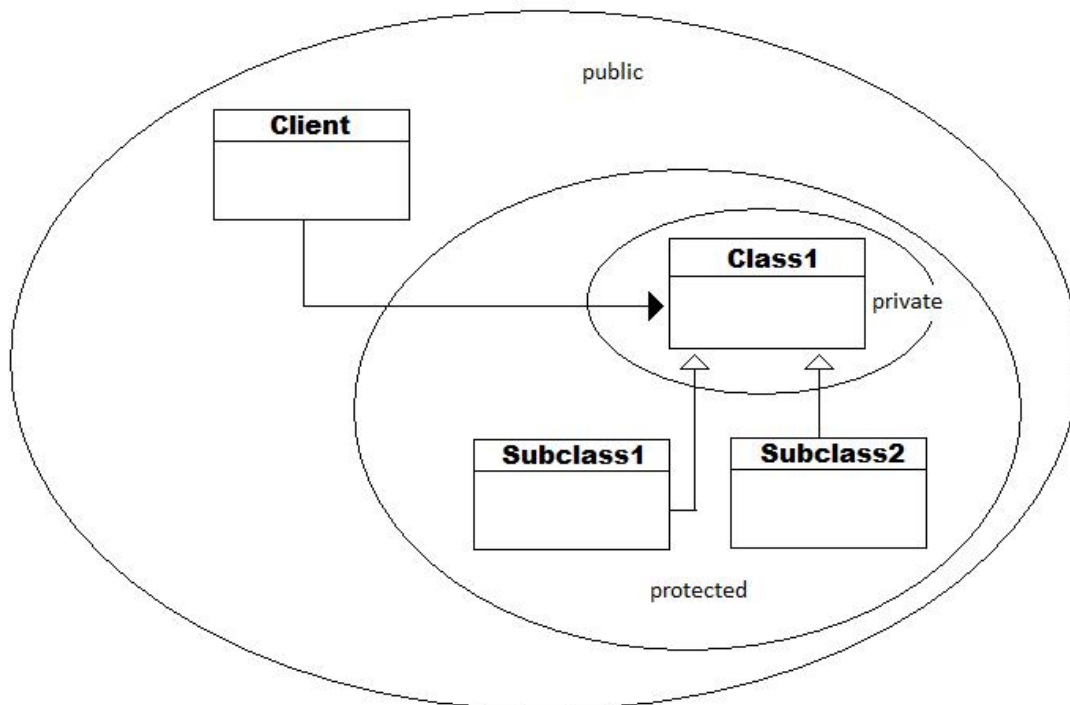
// methods
public void getTitle()
{
```

```

        ...
    }

```

Access modifiers define the visibility of a field, constructor or method. If an element (field, method or constructor) is *public*, it can be accessed from inside the same class and from any other classes. *Private* elements can be accessed only from within the class in which they are declared. They are not visible to other classes. A *protected* element can be accessed only from inside the class in which they are declared and from direct or indirect subclasses. Compare Illustration 2.



*Illustration 2: Access levels: public, private and protected*

### 5.3 Access modifiers for methods and fields

Fields, constructors and methods can all be either public or private, although so far we have seen mostly private fields and public constructors and methods.

A public method, as we will see mostly, are used to provide operations to users of a class (to access or mutate a field). A private method is used to break up a larger task into several smaller ones to make the large task easier to handle. These sub-tasks are not intended

to be invoked directly from outside the class, but are placed in separate methods purely to make the implementation of a class easier to read.

Another reason for having a private method is for a task that is needed in several of a class's methods. Instead of writing the code multiple times, we can write it once in a single private method and then call this method from several different places.

In Java, fields can also be declared private or public, but using public fields breaks the information-hiding principle. It makes a class that depends on that information vulnerable to incorrect operation if the implementation changes.

In short: fields should always be private.

## 6 Interfaces

### 6.1 Concept

Multiple inheritance exists in cases where one class inherits from more than one superclass. The subclass then has all the features of both superclasses, and those defined in the subclass itself.

Multiple inheritance can lead to significant complications in the implementation of a programming language. Some languages allow the inheritance of multiple superclasses, other not. Java lies somewhere in the middle: it does not allow multiple inheritance of classes, but provides another construct, called 'interfaces', that allows a limited form of multiple inheritance.

Interfaces are similar to classes, with the most obvious difference being that their method definitions do not include method bodies.

Due to the fact that in one interface we will find the definition of the methods, and no bodies for those methods, we can say that interfaces are similar to abstract classes in which all methods are abstract.

Java interfaces have a number of significant features:

- The key word *interface* is used instead of *class* in the header of the declaration.
- All methods in an interface are abstract; no method bodies are permitted. The *abstract* key word is not needed.
- Interfaces do not contain any constructors.
- All method signatures in an interface have public visibility. The visibility does not need to be declared (i.e. The *public* key word is not needed for each method).
- Only constant fields (public, static and final) are allowed in an interface. The key words may be omitted, but all fields are still treated as public, static and final.

Any class implementing one or more than one interface is forced to implement all the methods of the interfaces.



## 6.2 Definition of Interfaces

A class can implement an interface, which is very similar to overriding abstract methods from an abstract class. Java uses a key word – *implements* - for implementing interfaces.

A class is said to *implement* an interface if it includes an *implements clause* in its class header. For instance:

```
//Interface and Class inheritance
public class Circles extends Shapes implements Drawable
{
    ...
}
```

In this case, the class Circles extends a class and implements an interface, then the extends clause must be written first in the class header.

As mentioned above, Java allows any class to implement any number of interfaces (in addition to possibly extending one class). Thus, if a class implements more than one interface:

```
//Multiple inheritance of interfaces
public class Circles implements Drawable, Paintable, Visualizer{
    ...
}
```

## 6.3 Interfaces as types

If a class that implements an interface does not inherit any implementation from it, what do we actually gain by implementing interfaces?

Inheritance provides two great benefits:

- Avoiding code duplication and reusing of existing code.
- The subclass becomes a subtype of the superclass. This allows polymorphic variables and method calls.

Interfaces do not provide the first benefit (since they do not implement any method), but they provide the second. An interface defines a type just as a class does, allowing polymorphism because the variables can be declared to be of interface types, even though no objects of that type can exist (only subtypes).

Interfaces can have no direct instances, but they serve as supertypes for instances of other classes.

### ***6.4 Interfaces as specifications***

We have mentioned that using interfaces is useful to simulate multiple inheritance in Java and that it allows polymorphism, however the most important characteristic of interfaces is that they completely separate the definition of the functionality from its implementation. It means that two or more classes implementing an interface usually will have different characteristics, but the same intrinsic behavior or functionality. This fact combined with polymorphism (interfaces as types) allows us to use the interface as a variable until we really need to create a new object of some of the child classes, making our application work independently of the specific type of object we are currently using.

### ***6.5 Abstract class vs Interface***

Sometimes the decision between using an abstract class or an interface is easy: we need to use an abstract class when we want a superclass which already implements some methods. In other cases either abstract classes or interfaces can do the job. However, interfaces are usually preferable because we can simulate multiple inheritance and we still have the opportunity to inherit from a class.

## 7 Threads

### 7.1 Concept

Java provides the possibility to parallelize programs. Processes are not available but Java focuses on their closely relatives: threads. Threads are a build-in feature so not special thread library is necessary. The nature of parallel programming is very different to “normal” sequential program flow. Different threads run completely uncoupled to each other without special functionality you can not predict the internal stat of a threat relatively seen from another thread or process. In a certain manner the behavior is nondeterministic. To get more information about this the reader is advised to study chapter 3: Processes in *Tanenbaum, Andrew S.; van Steen, Maarten: Distributed Systems, Principles and Paradigms, Prentice-Hall Inc., New Jersey, 2002.*

### 7.2 Handling

In order to use a thread first a thread class must be defined which contains the code for the thread. Then the thread must be instantiated and started.

#### Example Class:

```
package ThreadExample;

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class NonsenseSpeaker extends Thread {
    /* In order to use threads this class must inherit from
     * "Thread". capabilities are inherited...
     */

    /* shared variable */
    static int globalPersonNumber = 0; //to distinguish printouts

    /* this is the the "body" of the thread: it
     * must be named "public void run()"
     * (in fact it is overridden)
     */
    public void run () {
        globalPersonNumber++;
        int personNumber = globalPersonNumber;

        /* just print some words */
        for (int i=0; i < 100; i++) {
            System.out.println("Talker " + personNumber + ":
blueb");
        }
    }
}
```

```
    }

}
```

### Main Class:

```
package ThreadExample;

/**
 * Here a talk show is modeled. The moderator tries to speak
 * something but he/she is interrupted by his audience. The
 * audience is represented by threads. This example shall
 * make clear the concurrent behavior.
 *
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */

public class TalkShow {

    public static void main(String[] args) {
        NonsenseSpeaker speaker1 = new NonsenseSpeaker();
        NonsenseSpeaker speaker2 = new NonsenseSpeaker();

        /* let people start speaking */
        speaker1.start(); /* start thread (implicitly "run"
                           is called) */

        speaker2.start();
        talkNonsenseAsModerator();
    }

    public static void talkNonsenseAsModerator() {
        /* just print some words */
        for (int i=0; i < 100; i++) {
            System.out.println("Maderator: bla");
        }
    }
}
```

The first class `NonsenseSpeaker` contains the code for the thread. This class must inherit from `Thread`. By doing so all the needed abilities to handle threads are inherited. The actual code is embodied in the **`public void run ()`** method. In fact this overrides the `run()` function of the father class.

To start the Thread just create an object of the previous defined thread class and run the `start()` method. Note that the start method is invoked and not `run()` because `start()` triggers the initialization of the thread (which can take some time) and afterward the `run()` function is called. After finishing code in the `run()` function the thread is stopped.

Due to the fact that Java only provides single inheritance there is an alternative to create threads. The thread class must implement the “Runnable” Interface and implement the `run()` method (in fact only `public class NonsenseSpeaker2 implements Runnable` changes):

```
package ThreadExample;

/**
 * @author Geovanny Giorgana & Sebastian Blumenthal
 * @version 2008-01-13
 */
public class NonsenseSpeaker2 implements Runnable {
    /* In order to use threads this class must inherit from
     * "Thread". capabilities are inherited...
     */

    /* shared variable */
    static int globalPersonNumber = 0; //to distinguish printouts

    /* this is the the "body" of the thread: it
     * must be named "public void run()"
     * (in fact it is overridden)
     */
    public void run () {
        globalPersonNumber++;
        int personNumber = globalPersonNumber;

        /* just print some words */
        for (int i=0; i < 100; i++) {
            System.out.println("Talker " + personNumber + ":
blub");
        }
    }
}
```

Starting the thread is quite similar with a little modification: an instance of `Thread` is created with a reference to an instance of the thread class as parameter. Then `start()` is invoked on the `Thread` object:

```
NonsenseSpeaker2 speaker1 = new NonsenseSpeaker2();
NonsenseSpeaker2 speaker2 = new NonsenseSpeaker2();

/* little helper */
Thread realSpeaker1 = new Thread(speaker1);
Thread realSpeaker2 = new Thread(speaker2);

/* let people start speaking */
realSpeaker1.start(); /* start thread (implicitly "run"
                      is called) */
realSpeaker2.start();
```

### 7.3 Synchronization

Whenever two or more threads want to invoke a method on the same object this could (depending on the code) lead to competing situations. To avoid this synchronization is needed. It ensures that only one thread can access the method at one time and other threads are blocked until the thread working on that method returns from its function call (the behavior is mutual exclusive). To enable synchronization just use the **synchronized** keyword:

```
public synchronized void myNiceMethod() {
```

Java provides some extended synchronization features like *conditional synchronization* (wait/notify). For more details look into the Java documentation.

## 8 Appendix

### 8.1 Package

A *package* is a group of classes. Java has default packages, which can be used by the user, however, the user can create his own packages.

To assign a class into a package call *pkgName*, for example, we need to write the following clause:

```
package pkgName;
```

This clause must be the first sentence from the source code Java file without counting comments and blank lines.

The name of the packages generally start with lowercase, to distinguish them from the classes, which start with uppercase. The name of a package can have several names, all of them gathered by dots ( the own Java's packages follow this rule, for instance *java.awt.event*).

The packages are used due to the following reasons:

1. To gather related classes.
2. To avoid name problems.
3. To help in the classes and members accessibility control.

To bring a package into a Java file you need to use the clause **import** *packname*. However, import a package do not make all the classes from this package to be accessible, only the public classes will be accessible. In addition, when a package is imported, the sub-packages will not be imported too. They must be imported explicitly, since they are different packages. For instance, when we import *java.awt* the sub-package *java.awt.event* is not imported.

The package *java.lang* is imported by default in Java.

## 8.2 Primitive types supported by Java

Java knows two kinds of type: primitive types and object types. Primitive types are stored in variables directly, and they have value semantics (values are copied when assigned to another variable). Object types are stored by storing references to the object (not the object itself). When assigned to another variable, only the reference is copied, not the object.

The following table lists all the primitive types of the Java language:

Type Name	Description	Example literals		
<i>Integer numbers</i>				
byte	Byte-sized (8-bit) integer	24	-2	
short	Short integer (16 bit)	137	-119	
int	Integer (32 bit)	5409	-2003	
long	Long integer (64 bit)	423266353	55L	
<i>Real numbers</i>				
float	Single-precision floating point	43.889F		
double	Double-precision floating point	45.63	2.4e5	
<i>Other types</i>				
char	A single character (16 bit)	'm'	'?'	'\u00F6'
boolean	A boolean value (true or false)	true	false	

Table 1: primitive data types