Java programming

Document number: 0883_12.fm

5 January 2023

ABIS Training & Consulting
Diestsevest 32 / 4b
B-3000 Leuven
Belgium

Document O:\Courses\Java\0883\0883_12.fm (5 January 2023)

Address comments concerning the contents of this publication to:

ABIS Training & Consulting
Diestsevest 32 / 4b, B-3000 Leuven, Belgium
Tel.: (+32)-16-245610
website: http://www.abis.be/
e-mail: training@abis.be

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

To cope with the complexity of application development, which has grown dramatically in the last decades, another way of thinking, analysing, designing and building applications has emerged: object oriented technology. In this course we will offer an insight in **OO programming using the Java language** and tools.

The Java SE course starts with a discussion of the basic language elements. This syntax knowledge is a prerequisite for constructing the **basic building blocks** of a (OO) Java application.

The OO concepts inheritance and polymorphism are discussed next, as well as the program **error and exception handling**.
Next we provide an overview of the collection framework, as well as some utilities for handling e.g. date/time and formatting data.

Input/output facilities in Java are huge. We present the basic stream handling, as well as advanced features via **functional programming**, available since Java 8.

A very important aspect of Java applications is the **event handling model**, which is obvious in graphical user interfaces (**GUI**), but which is also the foundation of the **Java beans**, the Java component model.

Optimisation of Java SE programming and advanced techniques are discussed in the ABIS 'Java advanced' course.
Building Java EE applications, with standard components like servlets, JSPs or Enterprise Java Beans, is discussed in the Java EE course curriculum, presented on www.abis.be.

Some other interesting publications and sites on Java are :

- **The Java Programming Language, 5th Edition** by Ken Arnold, James Gosling & David Holmes (Prentice Hall PTR, 2012)
  ISBN-10: 0132761688 - ISBN-13: 978-0132761680

- **Thinking in Java, 4th Edition** by Bruce Eckel (Prentice Hall PTR, 2006)
  ISBN 0131872486 - ISBN-13: 978-0-13-187248-6 (text updates and code at: www.bruceeckel.com).
  Bruce Eckel on Java 8: www.OnJava8.com ISBN 978-0-9818725-2-0

- **Java SE8 for the Really Impatient: A Short Course on the Basics** (Java Series) by Cay S. Horstmann (Addison-Wesley Professional - 2014)
  ISBN-10: 0321927761 - ISBN-13: 978-0321927767

- **Effective Java**: Programming Language Guide (Java Series - 3rd Edition) by Joshua Bloch (Addison-Wesley Professional - 2017)
  ISBN-10: 0134685997 - ISBN-13: 978-0134685991

  - www.oracle.com/technetwork/java/index.html
  - www.javaworld.com
  - www.javaranch.com

This course is based on **Java SE 8.0**

CHAPTER 1. JAVA INTRODUCTION

This chapter wants to give an introduction to Java-programming. Concepts won't be discussed, as they are treated in the ABIS course *Java concepts and techniques*.

First of all, compiling and running is dealt with. The compiled classes can only be run if the JRE (Java Runtime Environment) knows where this code resides, what the classpath is. How should this be defined?

Sometimes you create classes that are correct, but that generate warnings saying that you have used deprecated code. We explain what this means and what you should do.

Executable.java

```
// First Java Program
public class Executable {
    public static void main(String[ ] args) {
            System.out.println("Hello World!!");
    }
}
```

file name suffix: .java

file name = name public class

exact match - case sensitive

file can contain several class definitions

but only 1 public class

1.1 Getting started

On the opposite page, you see a small Java program, one that does only one thing: it writes the string “Hello World!!” on the console.

The program is written in a class *Executable*.

- The class is announced as being “First Java program”. The double slash makes this line a **comment**.
- The entire class definition is announced using the keyword ***class***. It is surrounded by braces.
- Notice that this class is more than a definition, it also executes a program. That is because it contains a method called ***main***. As we will see later on, this is the default **executable method**.

  The *main* method is accompanied by several **modifiers**. Some of them are mandatory, others aren’t.

  Braces are used to delimit the body of the method. In this case, the body exists of only one statement terminated by a semi-colon.

Java-source code is saved in a program with extension **.java**. A few things are worth mentioning:

- **several classes can be saved in one file**;
- **only one** of these **class** definitions can be **public**;
- give your java file exactly the same name as the (public) class: take upper and lower case into account (i.e. names are case sensitive).

Compiling

Java home directory JAVA_HOME -> **c:\jdk-17.0.2**

Default source and target -> current directory

```
%JAVA_HOME%\bin\Executable.java
%JAVA_HOME%\bin\javac.exe
```

%JAVA_HOME%*bin*> *JAVAC* Executable.java

result is byte code:
```
%JAVA_HOME%\bin\Executable.class
```

Specify source and target directory

```
%JAVA_HOME%\lib\Executable.java
%JAVA_HOME%\bin\javac.exe
```

%JAVA_HOME%\bin> JAVAC -sourcepath c:\jdk-17.0.2\lib\sub
-d c:\jjdk-17.0.2\class
c:\jdk-17.0.2\lib\Executable.java

result is byte code in:
```
%JAVA_HOME%\class\Executable.class
```

Environment (system) variable PATH

```
%JAVA_HOME%\bin\javac.exe C:\Exercises\Executable.java
```

C:\>SET PATH=%JAVA_HOME%\bin
C:\>Exercises> JAVAC Executable.java

result is byte code in (current directory):
```
C:\Exercises\Executable.class
```

1.2 Compiling Java programs

The **compiling** of your source code is done with the *javac* command. This command is part of the bin-subdirectory (adapt your path variable if necessary).

When compiling you should state which source file is the subject of this action:

```
javac <options> MyProgram.java
```

Table 1: Java compile options

-g	Generate all debugging info
-g:none	Suppress debugging info
-O	Optimize
-nowarn -Xlint:none	Suppress warnings
-verbose	Output messages about what the compiler is doing. This includes information about each class loaded and each source file compiled.
-deprecation -Xlint:deprecation	Output source locations where deprecated APIs are used
-classpath <path> -cp <path>	Specify where to find user class files
-sourcepath <path>	Specify where to find input source files
-d <directory>	Specify where to place generated class files (destination)
-target <release>	Generate class files for specific VM version
-source <release>	Provide source compatibility with specified release
-version	Version information
-Xlint	Enable warnings for legal but suspected program constructs
@<file_name>	Specify a file that contains arguments (can occur more than once)

If you are compiling one .java-file that refers to classes which are defined in another not yet compiled .java-file, that second file is compiled automatically.

After the compilation, one or more *.class-files* are created, one .class file for each class that is defined in the .java file. The name of each class-file is identical to the name of the corresponding class-definition. The .class files contain platform independent *byte code*.

> *Inner classes* are compiled in a file with a prefix that consists of the name of the outer class, followed by a dollar-sign ($) and the inner class (see 3.3 Inner classes on page 87).

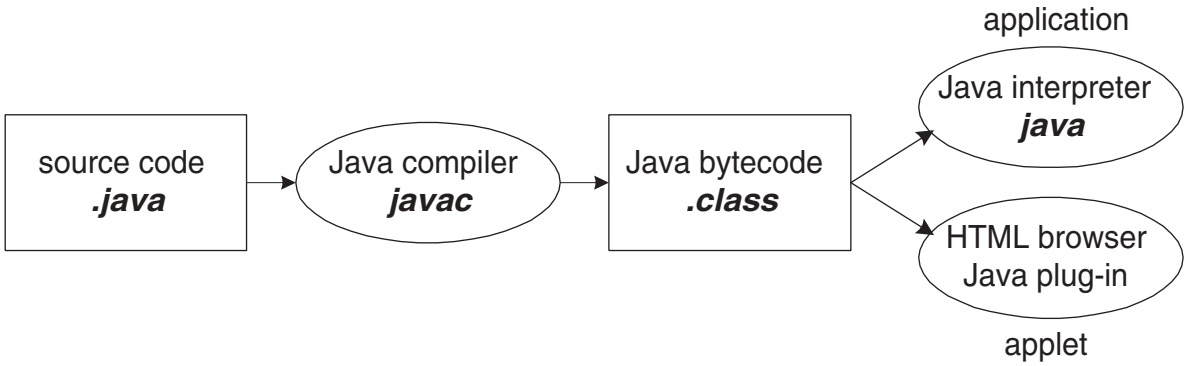
Running Java programs

java command runs JVM and interprets .class-file

```
%JAVA_HOME%\bin\java.exe
%JAVA_HOME%\bin\Executable.class
```

%JAVA_HOME%\bin\> java Executable

```
Hello World!!
```



applets are deprecated

1.3 Running Java programs

1.3.1 Applications

Running a Java-program means that you are **interpreting a .class-file** by a **Java Virtual Machine (JVM)** in a **Java Runtime Environment (JRE)**. The run command is ***java***.

```
java MyProgram
```

What is the usage of this command?

java [-options] class [args...]	to execute a class
java -jar [-options] jarfile [args...]	to execute a program from a jar

where options include:

Table 2: Java runtime options

-cp -classpath <directories and zip/jar files separated by ;>	Set search path for application classes and resources
-D<name>=<value>	Set a system property
-verbose[:class|gc|jni]	enable verbose output
-version	print product version
-? -help	print this help message
-X	print help on non-standard options

If you try to run a program that doesn't contain an executable method (i.e. main), you get the error message:

Exception in thread "main" java.lang.NoSuchMethodError:main

1.3.2 Applets

They cannot be run directly, they should be mentioned in an HTML file (<code> tag) that is invoked by a browser or an appletviewer.

These days, they are deprecated.

```
%JAVA_HOME%\bin\java.exe C:\Exercises\Executable
```

```
C:\>SET PATH=%JAVA_HOME%\bin
C:\>SET CLASSPATH=.;%CLASSPATH%;C:\Exercises;
C:\>JAVA  Executable
```

Deprecation

Java-items that might not be supported any more in future releases

produces warnings when compiling

find more details with

javac -Xlint:deprecation Executable.java

consult java-documentation for alternatives

Annotations

```
@Deprecated
@SuppressWarnings("deprecation")
```

1.4 Deprecation

After the compilation you sometimes get a warning that you are using deprecated code. Deprecated methods, constructors, classes... are Java-items that may not be supported any more in future releases.

The exact warning will be:

```
MyClass.java uses or overrides a deprecated API. Recompile
with -Xlint:deprecation for details.
```

These details are: the code that is deprecated as well as the advice to consult the documentation for an alternative. (see https://docs.oracle.com/javase/8/docs/)

The documentation lists all deprecated items and states if alternatives are provided.

Note: Annotations for deprecation

Documentation of deprecated elements can be done with the annotation `@Deprecated` in front of the element.

If you want to suppress the compiler warning messages about using deprecated elements, you can make use of the annotation

```
@SuppressWarnings("deprecation")
```

CHAPTER 2. JAVA SYNTAX

This chapter deals with specific Java syntax.

The following subjects will be discussed:

- adding comment to your code;
- blocks, statements;
- primitive Java types, their literal values and wrapper classes;
- identifiers and variables;
- final variables;
- arrays;
- enums;
- coding conventions;
- type conversion;
- operators;
- control flow.

Theoretical digressions will be reduced to a minimum, as examples will illustrate the underlying theory.

Detailed information can be found in the **Java Language Specifications (JLS)**
https://docs.oracle.com/javase/specs/

Coding conventions

```
// this is a class called Printer

public class Printer {
        public final static int COUNT = 0;
        public void printText(String s) {
            System.out.println(s);
        }
}

/* this is an executable class
   called Test   */

public class TestPrinter {
        public static void main(String[ ] args) {
            String song;
            song = "Don't take away the music";
            Printer remotePrinter = new Printer();
            remotePrinter.printText(song);
        }
}
```

class TestPrinter	**defines the class *TestPrinter***
remotePrinter.printText(song)	**variable (object) *remotePrinter* executes the method *printText(String)* with the argument song**
int COUNT	**defines the constant *COUNT***
output	**Don't take away the music**

2.1 Coding rules

2.1.1 Blocks and statements

A block is delimited by curly braces **{** and **}**. Everything between these braces belongs to the block. Blocks can be **nested**, i.e. one block can contain other blocks. A block contains **statements**, or can be empty. A statement is terminated by a **semi-colon**.

2.1.2 Comment

Java provides three ways to write comment:

- **//** ... for comment on **one line**;
- **/*** ... ***/** for **multiline** comment;
- **/**** ... ***/** for comment that will be transformed into **HTML documentation**, when the tool **javadoc** is invoked (see 4.4 Java documentation on page 103).

2.1.3 Coding conventions

Although you are free to use upper case and lower case as you wish, it is a wise idea to stick to a few conventions in order to obtain a standard environment. Not only is it important for you, but especially for client programmers who are using classes created by you.

2.1.3.1 Classes and interfaces

The names of **classes** and **interfaces start with an upper case**. Furthermore, the **first character of every noun** that can be distinguished in the name is in upper case, as well, in order to improve the **readability**.

2.1.3.2 Variables and methods

The **variables** and **methods** use **camel case notation**: they begin with a lower case, but every following noun that can be distinguished in the name starts in upper case.

Constants, i.e. *static final* variables (see 2.3.3 Final variables on page 37) or enum constants (see 2.8.3.1 Typesafe enums on page 57), are **entirely** written in **upper case**. (see e.g. the static variable `COUNT` on the opposite page)

2.1.3.3 Type variables

(see 7.2.2 Using generics on page 201)

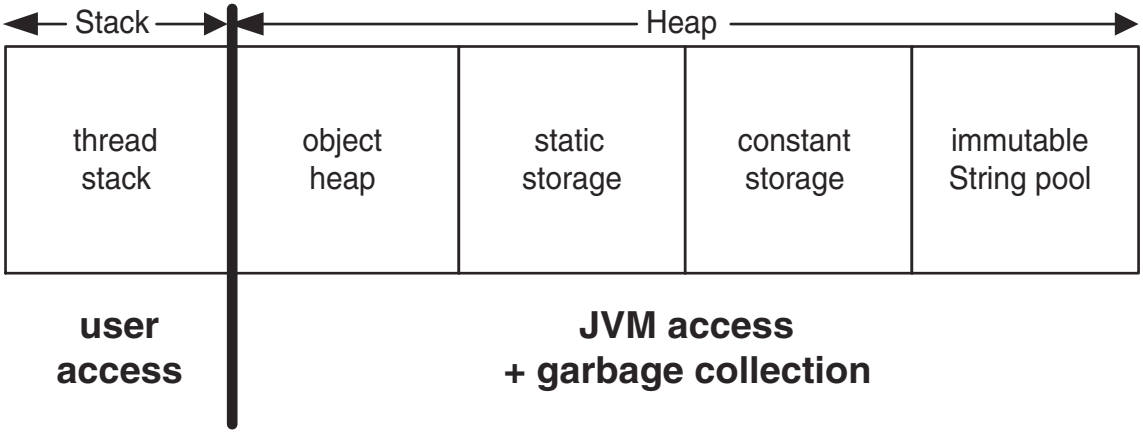
The naming convention for **type variables** is to use a **single capital letter**. This distinguishes these variables from the names of actual types since real-world types always have longer, more descriptive names. The use of a capital letter is consistent with type naming conventions and distinguishes type variables from local variables, method parameters, and fields, which are sometimes written with a single lowercase letter.

Collection classes like those in *java.util* often use the type variable <E> for "Element type". When a type variable can represent absolutely anything, <T> (for Type) and <S> are used as the most generic type variable names.

The memory model

```
public class Memory {
    private static String a = new String(“zebra”);
    public void method() {
        int b = 12;
    }
}
```

Figure 1: The memory model



handles a and b	- a and b are references to the memory - their scope is limited by the { }
int b	- primitive data type => saved on the stack - exact size and lifetime - default initialization value
new String(“zebra”);	- a is a handle/object reference to an object that is saved on the heap - size and lifetime are flexible - a constructor is used for initialization - the garbage collector is responsible for the cleaning up
static String a;	- static = everywhere available in the class - a is saved in a fixed location (static storage)
“zebra”	- Strings are stored as ‘immutable’ objects in the String pool, - will be reused if the same value is used in other places
12	- 12 = constant value in the program code (constant storage), 12 is saved as an int

2.2 The memory model

2.2.1 The stack

The **primitive data types** are **stored** on the **stack**. A stack pointer is moved down to create new memory and moved up to release that memory.

The java compiler must know the **exact size and lifetime** of all the data stored on the stack.

Because of the **fixed size,** all the primitive data types get a **default initialization value**.

The stack contains also

- all object references,
- and the local (scoped) method variables

and hence, the application user can access all program related references via the stack.

2.2.2 The heap

Java **objects** are stored on the **heap**. The JVM decides on the location (see picture on opposite page).

The java compiler doesn't need to know the size and the lifetime of the objects.

Objects can only be manipulated by using **handles** or **references**, available on the stack. The user/application will never ever access the heap directly!

If there is **no handle** to an object any more, the object could be destroyed by the **garbage collector**.

Primitive Java data types (I)

```
public class Test {
        public static void main(String[ ] args) {
            // logical type
            boolean b = true;

            // textual type
            char c = 'q';
            String text = "Welcome to the world of Java";
    }
}
```

2.3 Variables in Java

2.3.1 Java data types and literal values

Java defines eight primitive data types. These may be used as literal (fixed) values or in variables. The types may be considered to fall in four categories: logical, textual, integral and floating point.

2.3.1.1 Logical types

The only logical data type available in Java is ***boolean***. The two literal boolean values are true and false (no quotes).

2.3.1.2 Textual types

The data type ***char*** is a Unicode 2.0 character using a 16-bit unsigned number with a range of 0 to 2^{16} -1. A literal char is enclosed in single quotes. E.g.'a'.

Unicode is used instead of ASCII because it is an international character set standard. Hence, java is optimized for internationalized software coding: e.g. a literal 'é' is readily coded.

Other literals for special characters are represented by an *escape sequence*:

Table 3: Special characters

Character	comment
\n	new line
\t	tab
\r	return
\f	form feed
\\	backslash
\'	single quote
\"	double quote
\b	backspace

Note:

The last line on the opposite page mentions ***String***. However, this is not a primitive type, but refers to the complex type (class) java.lang and is used commonly for manipulating text strings (coded within double quotes!). You can find more info about the String class and its most important methods in 2.6 Working with String objects on page 47.

Primitive Java types (II)

```
public class Test {
        public static void main(String[ ] args) {
            // integral types
            byte number1 = 123;
            short number2 = 32000;
            int number3 = 2000000000;
            long number4 = 9_000_000_000_000_000_000L;

            // binary, octal and hexadecimal numbers
            int bin1 = 0b101                    //  4 + 0 + 1 = 5
            int oct1 = 016;                     //  8 + 6 = 14
            int hex1 = 0x16;                    //  16 + 6 = 22

            // floating point types
            float singlePrecision = 4.123132f;
            double doublePrecision = 1.5846d;
        }
}
```


2.3.1.3 Integral types

The four integral types are ***byte***, ***short***, ***int*** and ***long***. They are a sequence of binary, octal, decimal or hexadecimal digits. The start of the literal declares the base of the number:

- a leading 0 (zero) denotes an octal number;
- a leading 0b denotes a binary number, and
- a leading 0x or 0X denotes a hexadecimal number.

Any other set of digits is assumed to be a decimal number.

Integer literals are *long* if they end in L or l. Otherwise, integer constants are assumed to be of (default) type *int*.
You can add underscores in big numbers for readability.

By the way, it's a good idea to always use upper case suffixes, otherwise it's difficult to distinguish a 1 from an l.

Table 4: Integral types

type	length (bits)	range
byte	8	$[-2^7, 2^7 - 1]$
short	16	$[-2^{15}, 2^{15} - 1]$
int	32	$[-2^{31}, 2^{31} - 1]$
long	64	$[-2^{63}, 2^{63} - 1]$

2.3.1.4 Floating point types

The two floating point types are expressed as decimal numbers with a decimal point, optionally followed by an exponent. At least one digit must be present.

The number can be **followed by f** or **F** to denote a ***float***, which is a single-precision constant, or by **d** or **D** to denote a ***double***, which is a double-precision constant. Floating point literals are (default) of type double unless they are specified with a trailing f or F, which makes them float constants.

Table 5: Floating point types

type	length (bits)
float	32
double	64

Variables and identifiers

type identifier = expression;

```
public class Test {

    public static void main(String[ ] args) {
        int i1 = 7;
        int i2 = 5, i3 = 8, i4;
        int i5, i6;
        Printer remotePrinter = new Printer();

        i4 = 14;
        i5 = 25;
        i6 = i1 + i2 + i3 + i4 + i5;
    }
}
```

- **declaration of variables:**
  - **int i5, i6;**
  - **Printer remotePrinter;**
- **initialization of variables:**
  - **i5 = 25;**
  - **remotePrinter = new Printer();**
- **declaration + initialization:**
  - **int i5 = 25;**
  - **Printer remotePrinter = new Printer();**

2.3.2 Variables and identifiers

2.3.2.1 Identifiers

Identifiers are the **names** used **to identify variables** in Java code. Java keywords and the literals 'true', 'false' and 'null' cannot be used as identifiers. The keywords are listed in Appendix A. Java reference information on page 261.

2.3.2.2 Declaration of variables

The **declaration** of a variable involves choosing:

- a **type** (which specifies which values and behaviour are supported by the declared entities: primitive data types - object type);
- an **identifier** (if several declared for the same type, then they can be listed, separated by commas);
- **modifiers** (optionally) (e.g. *private*, *public*, *static*,...: will be discussed later).

--> declaration of variables: (**modifiers**) + **data type** + **identifier;**

2.3.2.3 Initialization of variables

Once the variables are declared, a value can be assigned to them, this is called the **initialization**, and is done for:

- a **primitive data type variable** by using the **assignment operator (=)**

  --> initialization of primitive data type variables:

  identifier = value;

- an **object reference variable** by using a **constructor** (which initializes a new object) or using the assignment operator for copying another object's reference.

  --> initialization of new object reference variables:

  identifier = *new* + constructor;

In some cases (during construction of members), variables get initialized with a default value. The following table shows these **default values**:

Table 6: Default values

type	default value
boolean	false
char	'\u0000'
byte, short, int, long	0
float	+0.0f
double	+0.0
object reference	null

Final variables

```
public class Test {
    public static void main(String[ ] args) {
        final int X;                        // X is blank final
        X = 10;
        final int Y = 20;
        System.out.println(X + Y);
    }
}
```

output

```
30
```

```
public class Test {
    public static void main(String[ ] args) {
        final int X = 10;
        X = 30;
    }
}
```

```
Test.java:4: cannot assign a value to a final variable X
X = 30
^
1 error
```

2.3.3 Final variables

A variable's behaviour can be modified in several ways. The first **modifier** we focus on, is ***final***. When you identify a variable as final, this means that the variable will have a value that **cannot be altered**.
When a variable is of type *Object*, the separate fields of that variable may change, but the variable itself will always refer to the same object.

Final instance variables are not initialized with a default value. If such a variable is **not initialized during declaration,** it is called a **blank final**. In order to use a blank final, you have to initialize it during construction. It is not possible to initialize such a final variable elsewhere, nor is it allowed to use a blank final.

> A final instance variable that is at the same time a class variable (*static final*) is called a *class constant*. Class constants are not allowed to be blank, in other words: they have to be initialized when they are declared. (see 3.2.3 Class variables and object variables on page 71)

It is also possible to declare local final variables and even to declare method parameters as final.

Note: Effectively final

A variable or parameter whose value is never changed after it is initialized is called **effectively final**. Starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or effectively final.

Type conversion

```
public class Test {
    public static void main(String[ ] args) {
        long l1         = 8;                    // implicit cast
        char c1         = 'a';
        int i2          = c1;
        int i3          = 97;


        long l2         = (long)i3;             // explicit cast
        double d1       = 8.345;
        int i4          = (int)d1;              // i.e. 8
        char c2         = (char)i3;             // i.e. 'a'

        int m           = 8.35;                 // => error
    }
}
```

```
Test.java:11: possible loss of precision
found   : double
required: int
int m = 8.35;
        ^
1 error
```

2.4 Type conversion

Java is a strongly typed language, which means that it checks for type compatibility at compile time in almost all cases. It provides **cast operations** if the compatibility of a type can be determined only at runtime, or if you want to explicitly force a type conversion for primitive types that would otherwise lose range, such as assigning a double to a float.

2.4.1 Implicit conversions

Some conversions are performed **automatically**, without you having to do something. There are two kinds of **implicit conversions**:

- **primitive conversion**, i.e. the conversion which applies to primitive values. Any numeric value can be assigned to another numeric variable whose type supports a larger range of values (widening conversion). Hence, Java allows implicit conversion of integer types to floating-point but not the other way around.

  Important remark: you can lose precision in some implicit conversions, e.g. when converting an int, which has 31 precision bits, into a float, which has only 24 precision bits, the 7 least significant bits are lost.

- **reference conversion**: since an instance of a class includes an instance of each of its supertypes, you can use an object reference of one type wherever a reference of any of its supertypes is required

2.4.2 Explicit casts

If such **implicit conversion** is **not possible**, a type can often still be explicitly cast to the other type. Explicit casts can also be used with object types. Two possibilities are provided for **explicit casts**:

- **down casting** (or **narrowing**):

  - be careful when casting primitive types: the result may be unpredictable, e.g. when an integer is cast to a char, only the 16 right most bits of data are used, the rest is discarded;

  - before down casting an object reference, you can (and usually should) test whether this cast is legal with the *instanceof* operator (see 5.3.4 Casting on page 135).

- **up casting** (or **widening**) is always legal: an explicit cast is not even necessary in this case because of Java's implicit conversions, though it may enhance readability.

converting primitive data types to objects

```
int i = 1;
Integer intWrapper = new Integer(i);
Double doubleWrapper = new Double(12.3456d);
char c = 'A';
Character charWrapper = new Character(c);
```

String conversion methods:

- **converting to a String:**
  - ***toString()***
  - ***toString(primitive type)***

```
String intString = intWrapper.toString();
String intString2 = Integer.toString(i);
String doubleString = doubleWrapper.toString();
String doubleString2 = Double.toString(12.3456d);
```

- **converting from a String: specific for data type**
  - Boolean.parseBoolean(String s)
  - Byte.parseByte(String s)
  - Short.parseShort(String s)
  - Integer.parseInt(String s)
  - Long.parseLong(String s)
  - Float.parseFloat(String s)
  - Double.parseDouble(String s)

```
int i = Integer.parseInt("-10");
```


2.4.3 Wrapper classes

All of the primitive types discussed above have a corresponding **wrapper class** (e.g. *Character*, *Integer*, *Double*...) to **convert primitive types into objects**. These wrapper classes are **useful to store primitives as objects**. E.g. Java's collections like lists, maps and sets cannot store primitive types.

Once the wrapper classes are instantiated and initialized, you cannot change the value of the object anymore. This is called **immutability** of the wrapper class objects.

These wrapper classes also provide several methods to convert the primitive data types to and from Strings:

- since the *char* type can already be converted to and from a String, nothing special needs to be done for it;
- to convert other primitive types to a String, use the *toString()* method on the wrapper class, if you have it available - if you do not have a wrapper class, it is possible to use a so-called static method (see 3.2.4.1 Class methods on page 73) of the wrapper class, also called *toString()* but it accepts an argument of the primitive data type that corresponds with the wrapper class.

To convert a String to a primitive data type, each datatype has its own methods. Usually, they are so-called ***static*** methods *parseXxx(String s)* where *Xxx* is the capitalized name of the primitive data type.

Some examples:

```
String s =  "123";
int i = Integer.parseInt(s);
s = "1.23";
double d = Double.parseDouble(s);
```

Autoboxing

conversion from primitive type to its wrapper class or vice versa

```
Float f = 365.25F;
char c = new Character('\t');
Double d = 2.718 + Math.PI;
```

Watch out for combinations

This is possible: unboxing + widening

```
long l = new Integer(345);
```

But this isn’t: unboxing + widening + boxing

```
Long l = new Integer(345);
```


2.4.4 Autoboxing

Automatic conversion or autoboxing eliminates the need of wrapper classes in a lot of circumstances. Conversion between primitives and their wrapper classes is no longer needed explicitly. It is as well possible to cast primitives to wrappers and vice versa.

Caution:

- this automatic conversion is paid with performance, therefore: if possible avoid autoboxing in sections where performance is an issue;
- a variable referring to a wrapper class can contain null - unboxing will result in a *NullPointerException*;
- the == operator performs reference identity comparisons on wrapper class objects and value equality comparisons on int (and mixed) expressions.

If a **new instance** of the wrapper class is **not required**, use the static ***valueOf*** methods instead of constructors, e.g.

```
Integer.valueOf(int i)

Long.valueOf(String s)
```

These methods are likely to be more performant than their corresponding constructors because they often only need to look up an existing object instead of constructing one.
In fact, this method call is exactly what happens when a value is boxed.

Invalid statements

The following statements try a little too much at once

```
1.toString();                    //autoboxing + method call
l = (Long) 2345;                 //implicit widening + boxing cast
Long l = new Integer(345);       //unboxing + implicit widening + boxing
```

Some additional examples of possible usage and non-usage of autoboxing are illustrated below.

Even this is possible ... The elements are (un)boxed when inserted, i.e. after creation of the array (for a discussion on arrays, see section 2.7 Arrays on page 51.)

```
Integer[ ] integers = {1, 3};
int[ ] ints =  {new Integer(1), new Integer(3)};
long[ ] longs = {new Integer(1), new Integer(3)};
```

... but not this:

```
integers = ints;     //you cannot "box an array"
longs = integers;
longs = ints;
```

Operators

```
public class Test {
        public static void main(String[ ] args){
            int a = 5 ,  b = 6, c = 1, y=10, x;

            x = a * b ;

            y = y + a;              y += a;
            x = x - a;              x -= a;
            x = ++y;                y = y + 1;
                                    x = y;
            x = y++;                x = y;
                                    y = y + 1;

            if ((a > b) && (b > c)) {
                a = b;
            }

        // conditional (ternary) operator
            c  = (a == b ? 8 : 9);

        }
}
```

Watch out:

```
x = 5;
x = ++x;
System.out.println(x--);
```

output **6**

2.5 Operators

Below you find a **table of all operators in order of precedence**, from highest to lowest precedence (e.g. assignment operators have lower precedence than equality operators).

Java guarantees that operands to operators will be evaluated **left-to-right**.

Except for the operators `&&`, `||` and `?:`, every operand of an operator will be evaluated before the operation is performed. && and || avoid evaluating their second operand if possible.

As to the conditional (ternary) operator `?:`, this provides a single expression that yields one of two values based on a boolean expression.

return_value = (true-false condition) ? (if true expression) : (if false expression);

The most important difference between this operator and an if-statement, is that the operator has a value.

Table 7: Operators

operator	symbols
postfix operators	`[]  .  (params)  expr++  expr--`
unary operators	`++`*`expr` `--`*`expr` `+`*`expr` `-`*`expr` `~  !` (NOT)
creation or cast	`new (`*`type`*`)`*`expr`*
multiplicative	`*  /  %`
additive	`+  -`
shift	`<<  >>  >>>`
relational	`<  >  >=  <=  instanceof`
equality	`==  !=` (NOT equal)
bitwise AND	`&`
bitwise exclusive XOR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
conditional (ternary)	`? :`
assignment	`=  +=  -=  *=  /=  %=` `>>=  <<=  >>>=  &=  ^=  |=`

Working with String objects

Most widely used class in Java

Part of java.lang package, imported automatically

Created without “new” keyword

```
String s= “Welcome to Java”;
```

Default value : null

- **watch out for NullPointerException**
- **test via == or !=**

Reside in the string pool

String objects are immutable

2.6 Working with String objects

2.6.1 Immutability and the String pool

The **String** class is one of the most widely used classes in Java. Since it is part of the java.lang package, the class is automatically imported.

Objects of type String can be created without the keyword “new”.

```
String s = “This is a string object”;
```

The default value of each String object is **null**, so watch out for the dreaded NullPointerException!
Null values can be checked by using == or !=

```
if (s!=null)...
```

Immutability and the String pool

String objects reside in the **String pool**, which is a special part of the heap which tries to **keep only 1 instance of each String object if possible**
(see Figure 1: The memory model on page 28).

The variable with which you access it, is just a reference to the object.
E.g. let’s create 2 variables:

```
String s1 =”abc”;
String s2 =”abc”;
```

If you compare the object identity of the 2 references using `(s1==s2)`, it should return true.

String objects are **immutable**, which means that their **value in memory can not be changed once it has been created**.

When you perform an operation on a String object, it will create a new object in the String pool. So beware to also (re-)reference the object before using it!

```
String s = “abc”;
s.substring(0,1);
System.out.println(s);     ->  will still return “abc”
```

vs.

```
String s = “abc”;
s=s.substring(0,1);
System.out.println(s);     -> will return “a” as expected
```

Important String methods

Common methods

- **equals() / equalsIgnoreCase()**

  ```
  “Bob”.equals(firstName);
  !“mArY”.equalsIgnoreCase(firstName);
  ```

- **concat()**

  ```
  firstName.concat(“by”);
  firstName+=“by”;
  System.out.println(“The firstName is” + firstName + “.”);
  System.out.println(1+2+”three”);
  System.out.println(“one”+2+3);
  ```

- **toUpperCase() / toLowerCase()**

  ```
  firstName.toLowerCase();
  ```

- **substring()**

  ```
  firstName.substring(0,2);
  firstName.substring(2);
  ```

- **replace()**

  ```
  firstName.replace(“A”,“O”);
  ```

- **length()**

  ```
  firstName.length();
  ```

- **trim()**

  ```
  firstName.trim();
  ```

Methods can be chained

```
firstName.substring(0,3).toLowerCase().replace(“a”,“o”);
```


2.6.2 Important methods

- **equals() / equalsIgnoreCase()**

  Although String objects are kept unique in the String pool as much as possible, one should always be careful when comparing them. When performing operations on String objects, or when using objects via different classes, the object identity could fail.

  The *equals()* method will **check whether the content of the two objects is equal**, so it should always be used to compare String objects.

  Whereas the *equals()* method respects the case of the two objects, the *equalsIgnoreCase()* does a case-insensitive check.

  The method signatures are as follows:
  ```
  boolean equals(String str)
  boolean equalsIgnoreCase(String str)
  ```

- **concat()**

  To concatenate two string objects, the method concat() can be used.
  The method signatures are as follows:
  ```
  String concat(String str)
  ```

  As an alternative, one can also use the “+” operator to concatenate String objects. Be careful when concatenating different types of objects.

- **toUpperCase() / toLowerCase()**

  These methods respectively convert the text into full upper- or lowercase letters.

- **substring()**

  The substring method **returns a part of a string object**. There are 2 overloaded versions:

  ```
  String substring(int beginIndex)
  String substring(int beginIndex, int endIndex)
  ```

  As usual in Java, the **index is zero-based**. The **endIndex is not included** in the outcome. If the endIndex is left out, it will take all characters starting at the beginIndex.

- **replace()**

  The replace method replaces all occurences of a value in a given String object by another value. Method signature:
  ```
  String replace(CharSequence c1, CharSequence c2)
  ```

- **trim()**

  Removes whitespaces (space, tab, newline,..) from the beginning and end of a String object.

- **length()**

  Calculates the length of a given String object. The return type is **int**.

Arrays

```
public class Test {
        public static void main(String[ ] args) {

            // declaring, initializing and filling an array of integers
            int[ ]  n1;   // also allowed: int  n1[ ];
            n1 = new int[2];
            n1[0] = 7;
            int n2[ ] = { 1 , 2 , 3 , 4 , 5 };

            // declaring, initializing and filling an array of Strings
            String[ ] s1  = new String [3];
            s1[2] = "is";
            String[ ] s2 = {“one”, null, “three”};

            // multiple subscripted arrays
            int[ ][ ] n4 = new int[2][ ];
            n4[0] = new int[1];
            n4[1 ]= new int[2];
            n4[0][0] = 7;
            n4[1][1] = 8;
            int[ ][ ] n5 = { { 1, 2 }, { 3, 4 } };
            int[ ] n6[ ];   // allowed, but not recommended

            // determining the length of an array
            System.out.println(n1.length);

            // printing the argument list
            for (int i = 0 ; i < args.length; i++){
                System.out.println(args[i]);
            }
    }
}
```

array index starts at 0

2.7 Arrays

Java arrays provide **ordered collections of elements of the same type**. Components of an array can be primitive types or references to objects. Even references to other arrays are allowed (because an array is a 'container' object).

2.7.1 Declaring and creating

Array dimensions are omitted in the declaration of any array variable. The number of components in an array is determined when it is created using *new*. An array object's **length** is fixed at its creation and **cannot be changed**.

When you create an array, you are actually creating an array of variables of the array's type. After the creation of the array, the object reference has a reference to an array of x variables that are initialized to their *default* values.

```
handle = new type [number of rows];
```

```
handle = new type [number of rows] [number of columns];
```

2.7.2 Initializing

It is also possible to declare an array and initialize it on the spot - with other values than default values. For that purpose you can just assign a comma-separated list of the initial elements enclosed by curly braces. This is only possible in the statement that declares the array!

2.7.3 Manipulating

An element of an array is referenced through the name of the array variable followed by the index of the element between **square brackets**. This reference is used like any other variable, i.e. you can retrieve its value or assign a new value to it.

The **first element** of the array has **index 0**; and the last element has index *length* - 1. The length of an array is available via the *length* field.

Note:

- although arrays are implicit extensions of *Object*, they cannot be extended with user-defined methods;
- when an invalid array reference is made, an *ArrayIndexOutofBoundsException* is generated.

Control flow (I) - if-else

```
public class Test {
        public int tryout(int testval) {
            int target = 12;
            int result = 0;
            if (testval > target) {
                result = -1;
            }
            else    if (testval < target) {
                        result = 1;
                    }
                    else { result = 0; }
            return result;
        }

public class Program {
        public static void main(String[ ] args) {
            Test t = new Test();
            int x = 10;
            System.out.println(t.tryout(x));
        }
}
```


2.8 Control flow

2.8.1 if

The *if*-statement evaluates the boolean expression between the parentheses. If this expression evaluates to “true”, the statements in the following block are executed; if it evaluates to “false” these statements are skipped.

```
if (<boolean expression>) <statement>
```

<statement> can be either a single statement or a block of statements (between curly braces).

2.8.2 if-else

The *if-else*-statement results in the same behaviour as the *if*-statement, except for the case when the expression evaluates to “false.” In the latter case, the statements in the block following the *else* keyword are executed.

```
if (<boolean expression>) <statement>
else <statement>
```

Of course *<statement>* may be a block of statements as well as yet another *if-(else)*-statement, such that a series of tests can be built.

Remark:

When building **boolean expressions** the use of **`&&` is preferred over `&`** when “and-ing” several conditions. The difference is that with `&&` the second condition is not even evaluated if the first condition evaluated as “false.” The `&` is only used when the second condition causes side effects - which is to be avoided at (almost) all costs, because it severely affects the readability of your program.

The **equivalent** reasoning holds for **`||` and `|`**.

Control flow (II) - switch

```
public class Test {
        public static final int LINE = 0, TRIANGLE = 1, CIRCLE = 2;

// switch on integral value
        public void draw(int shape) {
            switch (shape) {
            case LINE: System.out.println(“Draw a line.”); break;
            case TRIANGLE: System.out.println(“Draw a triangle.”); break;
            case CIRCLE: System.out.println(“Draw a circle.”); break;
            default: System.out.println("Draw whatever you want.");
            }
        }
}

public class Program {
        public static void main(String[ ] args) {
            Test t = new Test();
            t.draw(1);
        }
}
```

alternative: switch on String value

```
        public void draw(String shape) {
            switch (shape) {
            case “Line”: System.out.println(“Draw a line.”); break;
            case “Triangle”: System.out.println(“Draw a triangle.”); break;
            case “Circle”: System.out.println(“Draw a circle.”); break;
            default: System.out.println("Draw whatever you want.");
            }
        }
```

2.8.3 switch

A *switch* statement evaluates an expression to decide which *case* to execute. Execution continues at the ***case* label** inside the following block. The case labels may be followed by a *default* label. If no matching label is found, the statements following this ***default* label** are executed, if present.

The evaluated expression's value must be either of

- integral type (int, char, byte, ...) or
- an enum constant (see next page), or
- a String type.

A *case* or *default* label does not force a break out of the *switch*. In other words: after executing the statements for a certain *case* the program continues with the statements of the next *case.* In order to avoid "falling through" the statements following a *case* label usually end with a ***break*** statement.
(see also: 2.8.8 break - continue - return on page 63)

Typesafe enums

Declaration:

```
public enum Shape {
    LINE,
    TRIANGLE,
    CIRCLE;
}
```

Use:

```
public class Test {

    public void draw(Shape shape) {
            switch (shape) {
            case LINE: System.out.println("Draw a line."); break;
            case TRIANGLE: System.out.println("Draw a  triangle."); break;
            case CIRCLE: System.out.println("Draw a circle."); break;
            default: System.out.println("Draw whatever you want.");
            }
    }
}

public class Program {
        public static void main(String[ ] args) {
            Test t = new Test();
            t.draw(Shape.LINE);
        }
}
```


2.8.3.1 Typesafe enums

Declaration

The built-in support for the **enumerated type** (***enum***) is a typesafe implementation of choices to be used for instance in a switch statement.

These enums are references to a **fixed set of objects** that represent the various possible choices. Enums handle **discrete choices**, not combinations of choices.

Enums can be defined stand-alone (in their own .java files) or contained within other classes or even another enum. Inner class enums are considered static even if you don't use the static keyword.

Since **enum constants** are constants, normally they should all be capitalised. However, since the constant name is by default returned by its `toString()`, some people relax that rule and name constants with upper or lower case names as appropriate for the application.

Functionality

An enum constant is referenced by its type name followed by its name (with a dot in between). See the example on the opposite page.

Additionally, each enum type has some built-in functionality:

- static (type level) methods to fetch constants:
  - *`valueOf(`*`String value`*`)`* to find a constant with a specified name;

    ```
    v = Shape.valueOf("CIRCLE");
    v = Enum.valueOf(Shape.class, "TRIANGLE");
    ```

  - *`values()`* which returns an array of all constants from an enum type;
- methods that act on a specific constant:
  - *`name()`* returns the name of the constant as a String;
  - *`ordinal()`* to get the position of this constant in its declaration (counting starts from 0).

Note:

More information on the possibilities of enums can be found in 5.6 Typesafe enums and inheritance on page 157.

Control flow (III) - while

```
public class WhileTest {
    public static void main(String[ ] args) {
        double r = 0;
        while (r < 0.99d) {
            r = Math.random();
            System.out.println(r);
        }

        int i = 0;
        do {
            System.out.println("Are you finished yet ?");
            i++;
        } while (i < 10);
        System.out.println("At last !");

    }
}
```


2.8.4 while and do-while

The boolean expression in a **while loop** is evaluated and, if it is true, the following statement (or block of statements) is executed repeatedly until the boolean expression evaluates to false. A *while* loop **executes zero or more times** since the boolean expression might be false the first time it is evaluated.

The boolean expression in the **do-while loop** is evaluated after the **(block of) statement(s)** following *do* is **executed**.

Control flow (IV) - for

```
Public class ForTest {
    public static void main(String[ ] args) {

        int sum=0;
        for  (int c = 0; c < 12; c++) {
            sum += c;
        }
        System.out.println("total: " + sum);


        String[ ] words = { "Hello", "there", "Java", "programmer" };

        // for each String s in words
        for (String s : words) {
            System.out.print(s + " ");
        }
   }
}
```

```
output :

total: 66
Hello there Java programmer
```

2.8.5 for

The ***for*** statement is used to **loop over a range of values from beginning to end**. The initialization and iteration statements of a *for* loop can be a comma-separated list of expressions. The third expression is evaluated **after** executing the loop's body.

```
for (<initialization> ; <boolean expression> ; <iteration>)
    <statement>
```

All the expressions in the *for* construct are optional. If the boolean expression is left out, it is assumed to be true (presenting an infinite loop).

Infinite loops can always be coded as:

```
for ( ; ; )
```

2.8.6 enhanced for

The ***enhanced for*** or ***for-each*** **loop** is actually just a for loop with a slicker notation for iteration over groups of elements. In a *for-each* loop the iteration mechanism, which often only clutters up the code, is hidden.

When iterating over arrays, the index is hidden. Don't use a *for-each* if, as often is the case, this index is needed in the loop's body.

The strength of this flavour of the for loop is that it provides very elegant iteration over every class implementing the *Iterable* interface, as e.g. all Java's collections do (see Chapter 7. Collections on page 193).

Control flow (V)

```
public class Test {
    public static void main(String[ ] args) {
        for (int i = 10; i < 2000; i++) {
            if (i == 30) break;
            if (i % 2 == 0) continue;
            System.out.print(i + " ");
        }
        System.out.println("\n----------");

        int[ ] a = { 1, 2, 3, 4, 5, 6, 7 };

        outer: {
            for (int i = 0; i < a.length; i++) {
                for (int j= 4; j > 0; j--) {
                    System.out.println(a[i] + j + " ");
                    if ( i == j) break outer;
                }
            }
        }

        System.out.println("the end");
    }
}
```

```
output :

11 13 15 17 19 21 23 25 27 29
----------
5 4 3 2 6 5 4 3
the end
```


2.8.7 labels

Statements can be labelled. Those **labels** are typically used on **blocks** and **loops**. A label precedes a statement. Labelled blocks are useful with ***break*** and ***continue***.

2.8.8 break - continue - return

A ***break*** statement is used to exit from a block. An unlabelled *break* terminates the innermost *switch*, *for*, *while* or *do*. To terminate an outer statement, label the outer statement and use its label name in the *break* statement.

A ***continue*** statement skips the loop's body, updates the loop index and re-evaluates the boolean expression that controls the loop. A *continue* is often used to skip over an element of a loop range that can be ignored or treated with trivial code. A *continue* statement is only meaningful inside loops (*while*, *do-while*, *for*).

A *continue* statement can specify a label of an enclosing loop, which applies the *continue* to the named loop instead of the innermost loop. A labelled *continue* will break out of any inner loops on its way to the next iteration of the named loop.

A ***return*** statement terminates execution of a method and returns to the invoker.

Remark:

Use these statements with care:

- often a *for*-loop with a *break* is a confusing version of a *while*-loop with an appropriate condition;
- a *continue* statement in a loop is most often equivalent, though less readable, than an appropriate *if-else* block;
- it is considered good practice to have only one exit point for a method: therefore do not overuse the *return* statement.

CHAPTER 3. JAVA BUILDING BLOCKS

In this chapter we take a look at the different building blocks in Java, their characteristics and how they function.

Moreover we will discuss the accessibility of the building blocks from within a program, using different keywords.

Overview building blocks

- ***Packages***
  - contain
- ***classes, enums* and *interfaces***
  - contain
- ***class variables* and *class methods***
  - and
- ***object variables* and *object methods***

and perhaps, classes contain

- ***executable method* `main()`**
- ***initialization blocks***
- ***inner classes, enums* and *interfaces.***

classes can be instantiated, so there are

- ***objects***

which can be created (and initialized) with

- ***constructors***

Annotations (@): metadata for additional features

Java Beans: standardised Java classes/components

3.1 Overview building blocks

Java's all-embracing building block is the **package**. It is the largest Java structure, containing **classes**, **enums** and **interfaces**.

Classes can contain **constructors**, **methods** and **variables**.

Interfaces can contain the same items, although there are some restrictions as to variables. Interfaces are discussed when we look into the details of inheritance (cf. 5.5.2 Interfaces and methods on page 151).

Classes, enums and interfaces all declare **types**. But, unlike interfaces, classes can be instantiated and enums have a fixed set of instances.

Instantiating means creating a **concrete instance of a class** or, in other words, creating an **object** belonging to a class.

Objects have properties and behaviour:

- **properties** or **attributes** are referred to as instance or object **variables**;
- **behaviour** is determined by instance or object **methods**.

Objects are created by means of a **constructor**. In the constructor it is also possible to initialize an object's attributes or state.

In Java it is possible to generate properties and behaviour that is not related to the instances of a class, but to the class itself. These are the so-called **class variables** (class attributes) and **class methods**. One of these class methods, is the method that is going to set up and start the application (i.e the executable method, or, **main method**). Initialization of class attributes can be achieved using **initialization blocks**.

Interestingly, classes can contain other types, i.e. **inner classes, enums** or **interfaces**. This possibility is used inside a lot of standard Java SE classes, but can also be used for your own business classes.

Notes:

1. Annotations (@...)

   Annotations are a form of **metadata**, and provide data about additional program features, located outside the current program itself. This information is used:

   - by the compiler to detect errors or suppress warnings (see e.g. Note: Annotations for deprecation on page 11);
   - at compile-time and deployment-time processing to generate additional code, XML files, and so forth;
   - even at runtime processing.

2. Java beans

   Java Beans are Java **software components** conforming to a software component specification, or in other words, standardised Java classes that conform to a number of rules. These rules will be discussed throughout the course.

class Person

```
public class Person {

        // constructor for Person
        public Person() {
            System.out.println("New person created");
        }
}
```

Test program

```
public class Program {
    public static void main(String[ ] args) {
        Person p1, p2;
        p1 = new Person();
        p2 = new Person();
        Person p3 = new Person();
    }
}
```

class Person	**defines the class *Person***
main(){}	**defines the executable method**
Person p1, p2;	**declares two handles p1 and p2 of type *Person***
p1 = new Person();	**creates an object belonging to the *Person* class and binds it to handle p1**
p2 = new Person();	**creates an object of type *Person* and binds it to handle p2**
Person p3 = new Person();	**defines an object of type *Person***

3.2 Classes and objects

3.2.1 Classes

A class can be a part of a package, although this is not strictly necessary. Class-definitions are put between curly braces.

In the class you define the **common characteristics and behaviour** of the corresponding objects.

The class-definition can also contain a **constructor**, which must have the **same name as the class.** The constructor is used **to create a new object**.

Note: defining any of the afore mentioned items is optional. It is even possible to create a class without any definition. Empty braces are sufficient.

3.2.2 Objects

In an application program, which itself is defined in a class file, we will execute a main method, where objects are created and used (see 4.1 Program structure on page 97).

A **class** is **instantiated in an object** using the keyword *new*. The class definition declares what the object will look like. The ***new*** keyword is used in combination with a **constructor** of the class.

Objects can be asked to execute a method. Through the object reference or object handle, their variables and methods can be accessed.

Variables (or fields or members)

```
public class Company { }

public class Person {
    static int retirementAge = 65;          // class variable
    double weight = 75.6;                   // object variable
    Company comp;                           // object variable
}
```

Test program

```
    public static void main(String[ ] args) {
            Person x = new Person();
            Person y = new Person();

            System.out.println(x.weight);
            x.weight = 57;
            System.out.println(x.weight);
            System.out.println(y.weight);

            System.out.println(Person.retirementAge);
            x.retirementAge = 67;
            System.out.println(x.retirementAge);
            System.out.println(y.retirementAge);
    }
```

`double weight ;`	defines the object variable *weight* of type double
`Company comp ;`	declares a handle for objects of the class *Company*
`static int retirementAge = 65 ;`	defines a class variable *retirementAge*
`output`	75.6 57.0 75.6 65 67 67

3.2.3 Class variables and object variables

3.2.3.1 Class variables

A class variable is a **property of a class**, not of an object. E.g. in the *Integer* class two static variables are declared: MINVALUE and MAXVALUE. It is obvious that these values are linked to the class and not to a specific object of that class.

A class can have attributes of *any type*: **primitives or object types**. They can be declared at any moment in the class definition (at the beginning, in the middle, at the end).

Use the modifier ***static*** to declare a variable as being a class variable. Because a static variable belongs to a class, you should access it via the class name. An object reference isn't necessary.

A static variable is **initialized one time**: the first time an object of the class is created, or the first time a static method or static variable of the class is accessed; in short, as soon as the Java interpreter has to locate the *.class-file.

- Note that Java does not support global variables! Variables are local to a block, a method or a class, but, it is not possible to define variables (or constants) with program scope.

3.2.3.2 Object variables

When defining a class, you can define the **properties**, the attributes of an instance of that class. These **attributes** are called **fields, members, object variables**...

An object can also have attributes of *any type*: **primitives or object types**. They can be declared at any moment in the class definition (at the beginning, in the middle, at the end).

Java initializes member variables **automatically**, so you are not obliged to specify an **initialization** value. If you do want to initialize a variable, this can be achieved in several manners. You address them either directly, or via methods, or in the constructor, or, in rare cases, by means of an initialization block.

The object variables are *known* throughout the *class definition* i.e they have class scope. This implies that it may be unwise to declare a variable *var* and re-declare it in - let's say - a method as a local variable. Doing so may obscure the object variable in the body of such method.

Variables can have several **modifiers** (*public*, *static*,...: will be discussed in 3.4 Access control on page 89).

Methods

```
public class Person {
        private static String  function = "employee";
        private String name;

    //  class method
        public static void sayFunction() {
                System.out.println(function);
        }

    //  object methods
        public String getName() {
                return name;
        }
        public void setName(String aName) {
                this.name = aName;
        }
}
```

this: explicit reference to active object

Test program

```
        public static void main(String[ ] args) {
                Person x = new Person();
                x.setName("Bond");
                Person.sayFunction();
                System.out.println(x.getName());
        }
```

output

```
employee
Bond
```

3.2.4 Class methods and object methods

3.2.4.1 Class methods

The **class methods** are methods that define the **behaviour of a class**, not of an object. They have exactly the same structure as object methods but have an additional modifier, namely ***static***.

Examples of existing class methods:

- class *String*:

```
static String valueOf(int anInt){}
```

- wrapper class *Integer*:

```
static int parseInt(String aString){}
```

Notice that a static method can never refer to non-static variables or methods without an object reference, even if these variables or methods are defined in the same class.

3.2.4.2 Object methods

Defining the **behaviour of an object**, what an object is capable of, is expressed through methods. A method consists of a **method header** and a method body. In order of appearance the header (also referred to as the method's signature) consists of:

3. **visibility modifier** (*public, private, protected*) - determines whether the method belongs to the public interface or the private core of the object - the visibility modifier is optional;

4. **special modifiers** (*static, final, abstract*) - specify whether the method is a class method or an object method, whether or not it can be overridden (cf. 5.4.1 Final methods on page 137) or whether it is abstract (cf. 5.5.5 Abstract classes and interfaces on page 155);

5. **return type** - can be anything (a primitive or an object type) as long as it is explicitly returned as the last action of the method with a return statement - the return type is mandatory: if nothing is to be returned, you should state **void**;

6. **name of the method** - usually started in lower case - distinguishable words begin with an upper case letter to improve the readability;

7. **argument list** (between parentheses) - is mandatory - an empty list is allowed - use '(' to start and ')' to terminate the list - the arguments are separated by commas - the type of every argument (primitive or object type) must be declared.

Defining multiple methods with the same name, but different argument lists (the other parts may or may not be the same) is called **method overloading**. When calling a method the compiler selects the right one depending on both elements.

The header is followed by the **method body** between curly braces ('{' and '}'). The body consists of a list of statements. If your method is part of a "normal" class, i.e. a class that is to be instantiated, this part cannot be left out. However, an empty body is allowed.

Within the method body of an instance method, you can always explicitly reference the active instance using the **this** reference.

The executable method - *main()*

Test program

```
    public class Executable {
        private static String greeting=”Hello”;

        public static void main(String[ ] args) {
            Person p1, p2;
            p1 = new Person();
            p2 = new Person();

            p1.setName("Bond");
            p2.setName("Poirot");

            System.out.println(greeting);
            System.out.println("My name is " + p1.getName());
            System.out.println("My name is " + p2.getName());
        }
    }
}
```

output

```
Hello
My name is Bond
My name is Poirot
```


3.2.5 The executable method - *main()*

The ***main()*** method starts up an application. This method is **static** and it takes **one argument:** an **array of strings**. Whenever the *java* command is executed, the *main()* method of the class specified in the command is invoked. Hence the reason why it is a static method: as there are no objects before the start of the application, it is a method that can only be related to a class, not to an object of that class.

Command line arguments are stored in separate slots of the array argument of main. The number of slots in args[ ] is automatically adjusted to the number of strings passed with the command line. These arguments can then be accessed in the normal manner using an array index.

> Normally, a *main()* method is only written if the class has to be executable. But sometimes *main()* methods are added to other classes to make debugging easier.

Note that good object-oriented programs have (only 1) small *main()*-methods.

3.2.6 Invoking variables and methods

Variables and methods can be invoked by objects using the **object reference**, followed by a **dot**, which - in turn - is followed by the **method** or **variable name**.

Even **static** methods or variables can be invoked in this way. The difference is that for the latter, an object isn't required. You can invoke a class variable or class method by **referring to the class**.

An example of method invocation is shown on the opposite page.

Note:

Invoking variables and methods of inner classes is a bit more complex. Inner classes are only accessible via the outer class where they are defined. So you will always need a handle to the outer object. (see 3.3 Inner classes on page 87)

Initialization of object variables

```
public class Person {
    String lastName;
    String firstName;
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getLastName(){
        return lastName;
    }
}
```

Test program

```
    public static void main(String[ ] args) {
        Person p1 = new Person();

        p1.firstName = "James";                    // explicit - to be avoided
        p1.setLastName("Bond");                    // with set-method

        System.out.print(p1.firstName + " " + p1.getLastName());
    }
```

output

```
James Bond
```

3.2.7 Initialization of object variables

3.2.7.1 In an explicit way

This has already been illustrated. With the dotted notation you access the variable and give it an appropriate value.

However, this way you are not respecting the object-oriented encapsulation principle.

3.2.7.2 Using set-methods

A better way to initialize object variables is with **set-methods**. You define a method that is called

```
void setXXX(aValueForXXX)
```

in which the method body will contain at least something like

```
this.XXX = aValueForXXX;
```

For example, a class *Person* with an attribute name can offer the possibility to initialize that name as shown on the opposite page in the method *setLastName*.

Additional conditions as to the passed argument can be added.

Note:

In the example on the opposite page, you can find the corresponding get-method, which should be used to obtain the contents of the member variable.

Constructors

no return value!

definition of constructor not mandatory

- default constructor automatically created
- if overloaded, define explicitly

```
public class Person {
    String lastName;
    static int count = 0;

    public Person(String lastName) {
        this.setLastName(lastName);
        count++;
    }
}
```

Test program

```
    public static void main(String[ ] args) {
        Person p1, p2;
        p1 = new Person(“Mister M”);
        p2 = new Person(“Bond”);
        System.out.println(Person.count + " person objects created");
    }
```

output

```
2 person objects created
```

3.2.7.3 Initialization with constructors

If you choose for this kind of initialization, you **guarantee that your object has the right values for specified fields**. You can even produce behaviour at creation time, e.g. log some information. Conditions can be added to make sure that the initialization is done properly.

A constructor has the following **signature**:

1. the **name of the constructor** is exactly the same as the **name of the class** (upper and lower case must match), and is followed by

2. an **argument list between parentheses**: the argument list can be empty or can contain several arguments, each of them separated by a comma;

   If you didn't define a constructor in your class, Java implicitly uses a **default constructor**, i.e. a constructor without arguments (or **no-arg** constructor).

   However, as soon as you have defined a **constructor with an argument list**, a default (no-arg) constructor is no longer generated, so you have to define one explicitly.

3. following the argument list, the **constructor body** is specified: this is a sequence of **statements** in a block, i.e. put **between curly braces**.

   The constructor body can contain anything: initialization statements for fields, methods can be invoked, other objects can be created objects, etc. It is possible to call constructors from constructors.

Note that **a constructor never has a return value**. If you do specify a return type for your constructor, Java will consider it as an ordinary method.

Method- and constructor-overloading

```
public class Person {
        String lastName, firstName;
        java.util.Date dateOfBirth;

        public Person(String lastName) {
            this.setLastName(lastName);
        }
        public Person(String lastName, java.util.Date dateOfBirth) {
            this(lastName);
            this.setDateOfBirth(dateOfBirth);
        }

        public void talk(String str) {
                System.out.println(str);
        }

        public void talk(String str, int number) {
                for(int i = 0; i < number; i++)
                    this.talk(str);
        }
}
```

Test program

```
        public static void main(String[ ] args) {
                Person p1 = new Person("Bond");
                Person p2 = new Person("Poirot", new java.util.Date());

                p1.talk("Hello");
                p1.talk("How do you do ?",  2);
        }
```

output

```
Hello
How do you do ?
How do you do ?
```

3.2.8 Method- and constructor-overloading

You can create several constructors or methods with the **same name**, but that do something **different**. The difference is created through the number, type or order of arguments (also called **method signature**). This mechanism is called **overloading**.

Depending on the number and/or types and/or order of arguments you pass when invoking a constructor or method, the JVM will execute the appropriate constructor or method.

3.2.8.1 Method overloading

If two methods have the same name, but the number of arguments, their order or types differ, we call them **overloaded methods**.

An example is given on the opposite page.

Remark:

It's not allowed to overload an array argument and a vararg.

3.2.8.2 Constructor overloading

If two constructors have the same name, but the number of arguments, their order or types differ, we call them **overloaded constructors**.

An example is given on the opposite page.

Calling constructors from constructors:

When you write several constructors for a class, you can call one constructor from another, to avoid duplicating code. You can do this using the ***this*** keyword.

```
this();
```

Notice that the call should appear as the **first line** of code in a constructor and can occur only once.

Order of initialization

- **static initialization:**
  - **occurs only once;**
  - **initialization of static attributes and execution of static initialization blocks in order of appearance.**

- **non-static initialization:**
  - **allocating storage for an object;**
  - **initialization of attributes and execution of initialization blocks in order of appearance;**
  - **executing constructors.**

3.2.9 Order of initialization

Each time a .class file is loaded (you perform the *java* command for that file, or in your program you are creating an object of a class...), the following sequence of actions occurs:

- **static initialization** is done first (static blocks and static variables, in order of appearance), but *only if they haven't been initialized before*: if the Person class has been loaded once to create a person and later again to create another person, the initialization isn't repeated;

- when a **new object** is **created**:

  - first of all, enough **storage** for each field is allocated on the heap;

  - secondly, all **attributes and initialization blocks** are dealt with in order of appearance: attributes that are not explicitly initialized are set to their default values;

  - finally, all **initialization statements in the constructor** are executed.

Note:

Class and instance initializations are both preceded by their counterparts in superclasses (see: Chapter 5. Inheritance - Polymorphism on page 113).

Varargs

Variable number of arguments

declaration: Type ... (ellipsis)

```
public class Person {
    ...
    public void setChildren(Person... children) {
        for (Person p : children) {          // children is of type Person[ ]
            // do something
        }
    }
    public void schedule(Task task, Task... tasks) { ... }
    public void schedule(Date d, Task task, Task... tasks) { ... }
}

public class Program {
    public static void main(String[ ] args) {
        Person person = new Person();
        Person child1 = new Person();
        Person child2 = new Person();
        Person child3 = new Person();

        person.setChildren(child1, child2, child3);
        person.setChildren(child2);
        person.setChildren();
        // or:
        person.setChildren(new Person[ ] {child1, child2, child3});

    }
}
```

3.2.10 Varargs

A more **flexible way of writing/calling methods** is possible with an arbitrary number of arguments called ***varargs.***

3.2.10.1 Declaring and using a varargs parameter

- Only the **last argument** of a method's parameter list may be a varargs.
- To declare a varargs, its type is followed by an ellipsis '**...**'.
- In the body of a method with a varargs parameter, this parameter is used as if it were an array of the type associated with this varargs.

Remark:

- It's not allowed to declare overloaded methods, one with an array and one with a varargs.
- It is possible, though not at all recommended, to override a varargs with an array and vice versa.

3.2.10.2 Calling a method with a varargs parameter

It is in the method call that we will learn to appreciate this new feature. It is now possible to call the method with as much values as needed of the varargs type (or a subtype). Of course it is still possible to call this method with an array (backward compatibility).

Remark:

Take care when passing *null* values into varargs; in this case it is recommended to explicitly cast either to an array or a single parameter of its type. Inside the method the former results in a *null* value and the latter in an array containing a *null* value.

Which method is called?

```
p.schedule(null, null);
```

In fact the first one, but a cast is desirable.

```
p.schedule(null, (Task[ ])null);
```

Inner classes

```
public class Person {

    // 2 member classes
    class Ear {
        public String hear(String noise) {
            return noise;
        }
    }
    class Voice {
        public void talk(String word) {
            System.out.println(word);
        }
    }

    public void repeat(String vibrations) {
            Ear ear1 = new Ear();
            String sound = ear1.hear(vibrations);
            Voice voice1 = new Voice();
            voice1.talk(sound);
    }
}
```

Test program

```
    public static void main (String[ ] args) {
            Person bond = new Person();
            bond.repeat("My name is Bond");
    }
```

output

```
My name is Bond
```

3.3 Inner classes

As the name already indicates, inner classes are **classes within other classes** or, in other words, ***nested classes***. They have exactly the same structure as “normal” classes. The only difference is that, instead of being a direct member of a package, they are “owned” by another class. How to address methods and variables of such classes is discussed later in this chapter.

Inner classes come in four flavours:

- a *static member class* is a static member of a class: like any other static method, a static member class has access to all static methods of the parent, or top-level, class;
- a *member class* is also defined as a member of a class: unlike the static variety, the member class is instance specific and has access to any and all methods and members, even the parent's `this` reference;
- *method local inner classes* are declared within a block of code and are visible only within that block, just as any other local variable;
- finally, an *anonymous inner class* is a local class that has no name, defined and instantiated on the spot.

Non-static inner classes, in their turn, may not contain static elements.

From an object-oriented point of view, inner classes are very useful because they allow to **separate**, for instance, **a structure from its logic** to traverse it. E.g. when you create a tree structure, from an object-oriented perspective, this is only a structure, not a traversal algorithm. The algorithm, though, needs intimate knowledge and access to the structure. Because member classes can access all private data from their owning instance, such a class allows for separation of the structure part from the algorithm part without making the internals accessible to the rest of the world.

Inner classes (usually anonymous and member classes) are also frequently used to define callbacks - in a Java context usually called *listeners.* We will delve further into this subject in Chapter 12. Event handling on page 271. Typical examples of this pattern are found in graphical user interfaces and the Java Bean architecture.

Access control

```
public class Member {
    private String word1 = "aaa";
    public  String word2 = "bbb";

    private void repeat1(String vibrations) {
            System.out.println(vibrations);
    }
    public void repeat2(String vibrations) {
            System.out.println(vibrations);
    }
}
```

Test program

```
public class Test {
    public static void main(String[ ] args) {
        Member obj1 = new Member();

        obj1.word1 = "xyz";
        obj1.repeat1("not accessible");

        obj1.word2 = "xyz";
        obj1.repeat2("accessible");
    }
}
```

```
Test.java:5: word1 has private access in Member
  obj1.word1 = “xyz”;
       ^
Test.java:6: repeat1(java.lang.String) has private access in Member
  obj1.repeat1(“not accessible”);
       ^
```

3.4 Access control

The **access control** defines the degree in which elements in the program are accessible to one another. For instance, some properties or behaviour of objects should not be accessible to other objects in the application but objects belonging to subclasses; some objects should not be addressable by instances of classes in other packages etc.

To define access control for your Java building blocks, use the **visibility modifiers**:

- **public**

  A public variable and method **can be accessed by everyone**. It doesn't matter if client objects belong to classes in the same or in another package. Note that variables are seldom public.

  A public class is a class that can be instantiated by everyone. When creating a *.java-file that contains more than one class, you can only define one public class.

- **protected**

  This modifier makes **access** possible for **subclasses** and for **all classes from within the same package**. This item will be discussed later in the chapter about inheritance (see: 5.4.3 Protected variables and methods on page 141).

- **package friendly**

  This is the **default value** if no access modifier is specified. This means that **access** is **only** possible **from within the same package**. This is discussed more in depth on the next page.

- **private**

  Private **variables** and **methods** belong to the private core of the object, thus they **cannot be accessed directly by objects of a different type**. Variables that are private can only be read by means of appropriate **get-methods** and can be set by **set-methods** or in constructors. Obviously, such query- and modifier-methods should be more accessible than the variables they access.

  Note that variables are often private. Private methods are often defined for internal reasons, e.g. calculations that are needed in a public method. You can leave a variable accessible for readers, so you don't use the private modifier, but you can make it inaccessible for writers by making it final.

  Classes cannot be private. It would mean that they cannot be instantiated. There is a way to prevent a class from being instantiated, but this is done in a different manner.

Packages

- **packages create a grouping of related classes**
- **classes without package are allowed**
- **class names: uniqueness required *inside* the package**
    - use internet names (reverse URL)

```
package be.abis.products;
public class ProductCatalog { }
```


3.5 Packages

3.5.1 Concepts

Packages are the largest building block in Java. They **group related classes**. What that relation is, can differ from package to package. You can create a package that holds classes and interfaces for general use. Or a package can contain classes and interfaces representing a whole application.

3.5.2 Defining a package

Creating a package and allocating a class or interface to a package is quite simple. The keyword ***package*** is used. The announcement of the package name is the first line of your code when creating a class or interface and by convention package names are all lower case.

When resolving a reference to a another class, the JRE searches for a class file in its classpath under a folder structure that matches the package structure for the referenced class.

3.5.3 Package friendly access

As mentioned on the previous page the second restrictive access level is package friendly. To declare package friendly access, specify no access modifier at all. Package friendly elements can be **seen by all classes in that package**, but not in a subpackage.

Referring to classes

- **explicit reference**

```
package be.abis.sales;
public class CashRegister {
    be.abis.products.ProductCatalog catalog =
            new be.abis.products.ProductCatalog();
}
```

- **import**

```
package be.abis.sales;
import be.abis.products.ProductCatalog;

public class MyClass {
    ProductCatalog catalog = new ProductCatalog();
    System.out.println(catalog.toString());
}
```

```
package be.abis.sales;
import be.abis.products.*;
import static java.lang.System.*;

public class MyClass {
    ProductCatalog catalog = new ProductCatalog();
    if (catalog.isEmpty())
        err.println("catalog is empty");
    else
        out.println(catalog.toString());

}
```


3.5.4 Referring to classes outside the current package

Direct access to other classes is only possible within the same package. What do you have to do if you want to access a class outside the package you are working in?

- Using an **explicit reference** - i.e. referring to the package to which a class belongs.

- Using the **import statement** - before defining your class(es), you specify which packages should be imported. If a class cannot be found in the current package, Java looks for a class with the same name in one of the packages you imported. It is easy and you don't have to repeat the package name every time you refer to a class.

  A **static** import statement construct allows unqualified (without referring explicitly to the class) access to static members (methods, fields, enums and inner classes) of classes. You should only use it when you require frequent access to static members from one or two other classes. If you overuse the static import feature, it can make your program unreadable and unmaintainable. And if you need only one or two members, import them individually instead of importing all of them.

Note that the first approach can result in duplicate code, while the second approach can be confusing when two packages are imported, each holding a class with the same name. The actions of the JVM towards loading the class will be unpredictable in that situation.

CHAPTER 4. JAVA APPLICATIONS

The Java programming language is designed to meet the challenges of application development in the context of heterogeneous, network-wide distributed environments. Paramount among these challenges is secure delivery of applications that consume a minimum of system resources, can run on any hardware and software platform, and can be extended dynamically.

How do you create reliable and well-structured applications? And how can these applications be documented and deployed? The answer can be found on the following pages.

application:

- **one file for the *main* program**
- **a set of files: one for each related class definition**

example:

```
public class Executable {     /* main program */
   public static void main(String[ ] args){
      Address address = new Address("3000","Leuven","Belgium");
      Person person1 = new Person("An");
      person1.setAddress(address);
      Person person2 = new Person("John","Smith");
      person2.attendCourse();
      System.out.println("An's town: "+person1.getAddress().getTown());
      System.out.println("John's last name: "+person2.getLastName());
   }
}


public class Address {
      /* variables */
       private String street,zipcode,town,country;
       private int nr;
       /* constructor */
       public Address(String zipcode, String town, String country) {
           this.zipcode = zipcode;
           this.town = town;
           this.country = country;
       }
       /* accessor methods */
       public String getStreet() { return street; }
       public void setStreet(String street) { this.street = street; }
       public int getNr() { return nr; }
       public void setNr(int nr) { this.nr = nr; }
       .......
}
```

4.1 Program structure

Design and structure of Java programs is (or should be) subjected to object-oriented software programming principles. This means that the solution to a specific problem has to be encapsulated in a class definition. However, the complete solution to any problem will not be solved by 1 single class, but by a combination of interacting software components.

A **complete application** can be seen as a main program, in which the collaborating objects are defined, followed by sending messages to the individual objects to get the desired response. The reaction to these messages is implemented in the associated methods of the corresponding classes. These methods are dynamically invoked by the Java Runtime Environment (or Java Virtual Machine).

This means that the program source is situated in at least two files:

- **one file for the main program**, which contains a *main()* method. This main method defines/declares the objects by calling the appropriate constructors. Afterwards the objects are used to invoke the required methods on these objects.

- **one file per class**, associated with the used object(s). Each class defines the structure for the objects and the methods associated with the object. This functionality depends on the kind of object and is discussed in subsequent chapters.

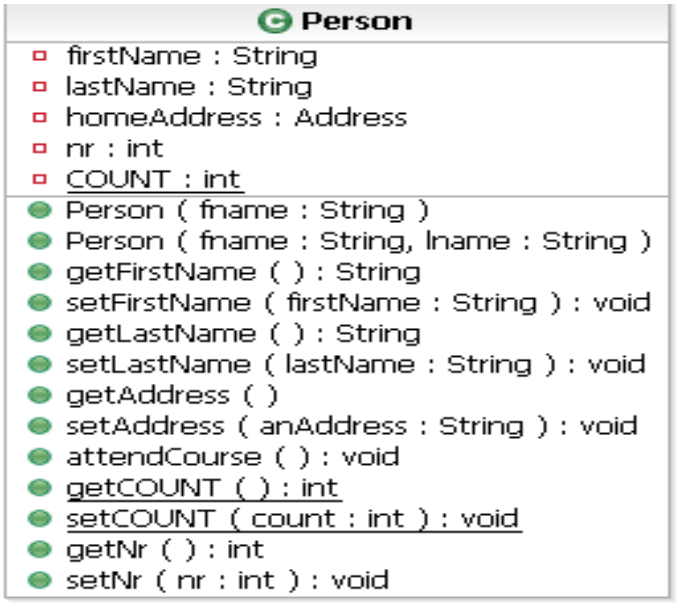
An example of the defined files (classes) for an application, handling personal and address information, is shown on the opposite page.

In the main program, the *Person* object is asked about its address and another *Person* object about its name in order to display it on the console.

Details on the *Person* class are shown on the next page.

So the structure of an application is quite easy.

Figure 2: Person model




```
public class Person {
 /* variables */
    private String firstName, lastName;
    private Address homeAddress;
    private int nr;
    private static int COUNT;
 /* constructors */
    public Person(String fName) {
        this.firstName = fName;
    }
    public Person(String fName, String lName) {
        this(fName);
        this.lastName = lName;
    }
 /* accessor methods */
    public String getLastName()   { return lastName; }
    public void setLastName(String newName) { lastName = newName; }
    public Address getHomeAddress()   { return homeAddress; }
    public void setHomeAddress(Address address) { homeAddress =
                                                        address; }
     . . .
 /* business methods */
    public void attendCourse() { System.out.println("attending course"); }
}
```

4.2 Class structure

Once the application behaviour is modelled into the class domain, the individual classes can be coded according to the model.

An example for a typical Java class is depicted on the opposite page.

A person object is used to store personal information. The object itself refers to *String* objects, an *Address* object, ...

In the class you will find the

- **variables** (member fields), of **primitive data type** or **object type**:
  - **object fields**, attributes used by the class instances during the life cycle of an object, e.g. `firstName, lastName`;
  - **class fields**, (common) attributes on the class level, that do not depend on individual objects - defined by the modifier ***static,*** e.g. `COUNT`;
- **constructors**: used to construct the object instance and fill the appropriate member fields, e.g. `Person(String fname)`;
- **accessor methods**: used to access (**get/set**) the different member fields;
- **business methods**: implementing the responsibilities of the class objects - in the example, a person can attend a course using method `attendCourse()`.

The individual methods in the classes can be quite complex. A good design is required to reduce the complexity of the individual methods, by making the objects/classes more granular. Also, it is good practice to **limit the number of methods** for an object.

More information on code structuring can be found in the Appendix A.3 Class structure on page 264.

Note:

If applications have to be created for a distributed environment, the different parts of the application (user interface, business logic and data access) will be coded in different packages. In this course, the creation of the user interface is discussed in Chapter 11. Building GUIs on page 257.

The creation of server side programs is handled in the ABIS courses on Java EE. (see www.abis.be)

Java Beans

= Java class + well-defined protocol

= Java's reusable software component model

- **properties - the things a bean knows**
  - manipulated publicly
  - not always attributes
- **methods - the things a bean can do**
  - all public class methods
- **events - the things a bean can tell**
  - notification to event listeners

characteristics/rules:

- **naming conventions (camel case)**
- **get/set methods**
- **no-arg constructor**
- **implements Serializable**
- **optional: support of event-model**

4.3 Java Beans

The goal of the JavaBeans API is to define a software component model for Java, as to enable third party vendors to create and ship Java components that can be composed together into applications by end users.

The Java Beans specification defines a Java Bean as:

> a **reusable software component** that **can be manipulated visually in a builder tool**.

In short, Java Beans are Java **software components** conforming to a software component specification: **rules** that describe what a **Java class** must do and understand to be considered a "bean". In other words, beans are Java classes **with specific characteristics** allowing them to be treated as **components**. Programmers, software developers and, equally important, integrated development environments (IDEs) have to be able to rely on any class that advertises itself as a Bean to conform to the rules set out in the bean specification document. If a class does not conform to these specifications, the contract has been broken and the Bean is said to be defective.

The bean developer must always make sure that a class, to be considered as a bean, conforms to the **three bean characteristics.**

1. A bean has **properties** that define the state (data) of the bean. Properties are **the things the bean knows.** They are named attributes that can be read and/or written by calling appropriate methods on the bean. Note that not all properties of a bean have to be attributes. Every item mentioned in a *getXXX()* method will be regarded as a property at the moment of introspection.

2. A bean has **methods** that define its behaviour. Bean methods can be called by anyone just by making each one available in the class interface. Bean methods are thus normal Java methods that can be called from other components. By default, all of a bean's public methods will be exported as bean methods. Methods are **the things the bean can do**.

3. A bean can generate **events** to notify event listeners about state changes of the bean. Events are **the things a bean can tell**.

The main rules for a Java Bean are:

- naming conventions for classes, attributes and methods are based on **camel case notation**;

- a **default (no-arg) constructor** must be provided.

- names of the **property accessor methods** have to meet particular naming conventions, according to the property type and composition (simple, boolean, indexed). Typical names are getXXX(), setXXX(), isXXX();

- because it should be possible to make objects persistent, the class has to implement the ***Serializable*** interface (marker interface) (or *Externalizable*) (see also 9.6 Object streams on page 255).

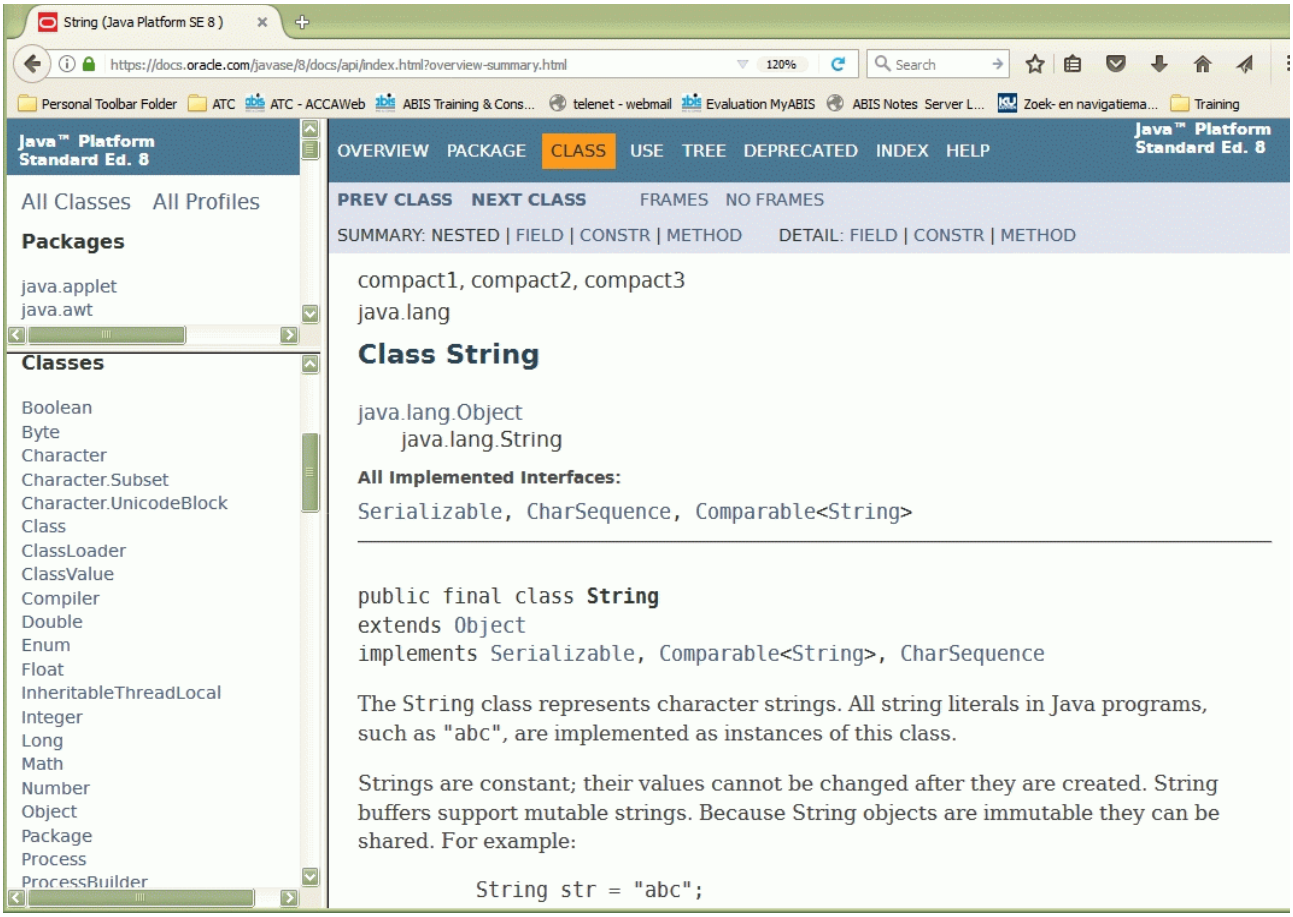
Optionally, if events are associated with the bean class, the bean must implement the Java event model. A **bean event** is defined by incorporating **methods for the addition and removal of event listeners** for that event in the class definition of the bean. (see Chapter 12. Event handling on page 271).

HTML-pages automatically created with *javadoc* command

options exist to have more or less information

core Java classes already documented

Figure 3: Java documentation



4.4 Java documentation

4.4.1 Tool javadoc

Documentation is indispensable, but alas, it is often forgotten. Java has provided a tool, called *javadoc*, to be found in the bin-directory. By running this tool on your source-code, **HTML-files** are created automatically, which contain a standardised description of fields, constructors and methods in the class.

The **usage** is:

```
javadoc [options] [packagenames] [sourcefiles] [@files]
```

It would lead us too far to deal with all the options. The most important ones are:

Table 8: Javadoc options

option	description
-public	show only public classes and members
-protected (default)	show protected/public classes and members
-package	show package/protected/public classes and members
-private	show all classes and members
-sourcepath <pathlist>	specify where to find the source files
-d <directory>	destination directory for output files
-version	include @version paragraphs
-author	include @author paragraphs
-nodeprecated	do not include @deprecated information
-notree	do not generate class hierarchy
-noindex	do not generate index

Notice that you can download the documentation for the core java-classes:

```
https://docs.oracle.com/javase/8/docs/
```

HTML pages

same structure as documentation on core Java-classes

formatting of code in source-file

- /** */
- @see
- @param
- @return
- @exception
- @deprecated
- @author
- @version
- @since

4.4.2 Format of documentation

Now, what of your source file will be regarded as documentation? The code itself won't become a part of the HTML file. Only the declaration of variables, constructors and methods will. And apart from that, additional text, that you layout in a certain way, reaches the documentation file. What is that layout?

Doc comments start with **/**** and continue until the next ***/**. Each doc comment describes the identifier whose declaration immediately follows. Leading * characters are ignored. The first sentence (!) of the comment is the summary for the identifier.

Doc comments can contain **tagged paragraphs** that hold particular kinds of information. All these tags start with **@**. These paragraphs are treated in detail in the generated documentation, resulting in marked paragraphs, links to other documentation and other special treatment.

@see

creates a link to other javadoc documentation.

@param

documents a single parameter to a method.

@return

documents the return value of a method.

@exception or @throws

documents an exception thrown by the method.

@deprecated

marks an identifier as being deprecated.

@author

specifies an author of the code.

@version

lets you specify an arbitrary version specification.

@since

lets you specify an arbitrary version specification that denotes when the tagged entity was added to your system.

Note: enhanced formatting

It is possible to add HTML tags inside the Javadoc comments, in order to enhance the presentation of the information, e.g. usage of the `<b>` tag.

Javadoc - an example

```
/**  Person (class) describes the personal behaviour
*@version 1.2
*@author ABIS
*@since 1.0
*/
public class Person {
/** name field: person's name */
  private String name;
/** addr field: person's address */
   private Address addr;

/** Person (constructor) creates a new person with a given name */
  public Person(String aName) { /* constructor */ }

/** getName returns the person's name */
   public String getName()   { return  name; }

/** setName sets the person's name
*@param newName
*@see #getName()  */
   public void setName(String newName) { name = newName; }
}
```


4.4.3 Javadoc - an example

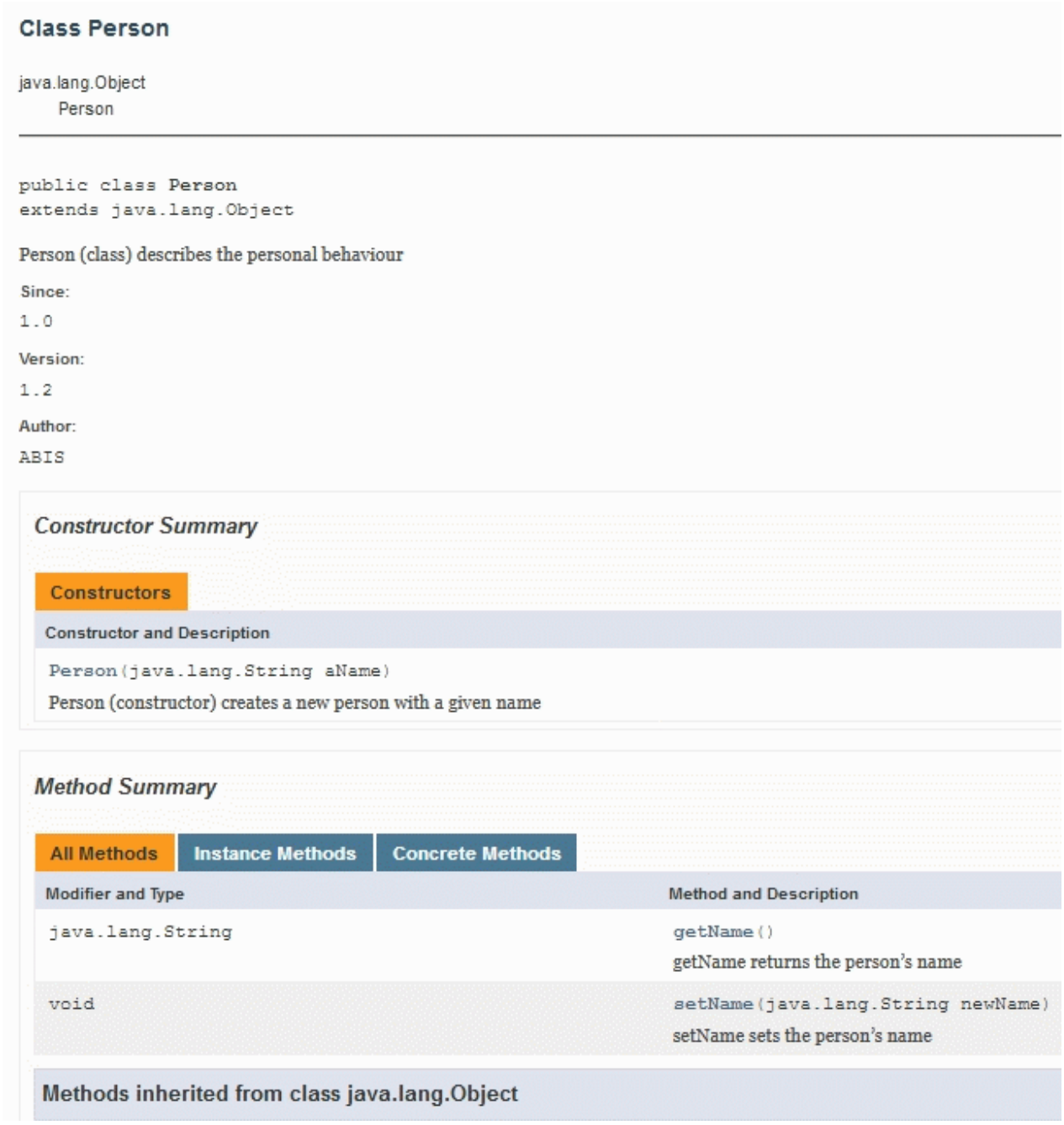
The documentation that is generated by using the ***javadoc*** command, has got the same layout as the documentation of the core Java classes.

The exact command is:

```
javadoc -private -author -version Person.java
```

Resulting documentation

Figure 4: Generated class documentation



Class Person

java.lang.Object
    Person

```
public class Person
extends java.lang.Object
```

Person (class) describes the personal behaviour

Since:

1.0

Version:

1.2

Author:

ABIS

Constructor Summary

Constructors
Constructor and Description
`Person(java.lang.String aName)` Person (constructor) creates a new person with a given name

Method Summary

All Methods	Instance Methods	Concrete Methods

Modifier and Type	Method and Description
`java.lang.String`	`getName()` getName returns the person's name
`void`	`setName(java.lang.String newName)` setName sets the person's name

Methods inherited from class java.lang.Object

The result of the specified comments is reflected in the generated HTML document, shown on the opposite page.

JAR files

Java ARchive file format

to collect a group of files into a single (compressed) file

advantage:

cross-platform <-> zip-files

use: internet applications

how can they be obtained?

--> jar-utility

example

Manifest file Manifest.mf

```
Manifest-Version: 1.0
Main-Class: HelloWorld
```

(+ <new line> or <carriage return>)

create jar

```
c:\>jar cvfm Hello.jar Manifest.mf *.class
```

execute application from jar

```
c:\>java -jar Hello.jar
```


4.5 JAR files

Different class files can be gathered and compressed into a single **J**ava **AR**chive-file, identified by the suffix ".jar". A JAR file is a **ZIP format archive file**, that may also include audio- and image files. JAR files are used to collect class files, serialized objects, images, help and similar resource files that have to be shipped or deployed together.

A JAR file may optionally have a **manifest file** including **additional information** about the contents of the JAR file.

> Manifest's entries take the form of "header: value" pairs. The name of a header is separated from its value by a colon.
>
> The text file from which you are creating the manifest must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return.
>
> By convention, JAR file entries are given slash-separated names such as "a/b/c".

Jar files are platform-independent. They are particularly helpful when dealing with the internet. E.g. using isolated class files would result in repeated requests of a Web server in order to download all the files that make up an applet. By combining all the files for a particular applet into a single JAR file, only one server request is necessary. Moreover, the transfer is faster because of the compression.

So, JAR files help in deploying applications to the users.

JAR files can be obtained by using the jar-utility (bin-directory). The jar-tool is used as follows:

```
jar {ctxu} [vfmOM] [jar-file] [manifest-file] [-C dir] files ...
```

Some options are depicted in the table below

Table 9: Options for JAR files

c	create new archive
t	list table of contents for archive
x	extract named (or all) files from archive
u	update existing archive
v	generate verbose output on standard output
f	specify archive file name
m	include manifest information from specified manifest file
O	store only; use no ZIP compression
M	do not create a manifest file for the entries
-C	change to the specified directory and include the following file

CHAPTER 5. INHERITANCE - POLYMORPHISM

In this chapter we will discuss the important topics of inheritance and polymorphism, as well as some related subjects, such as the modifiers final and protected, and abstract classes.

And as a matter of fact, the power of interfaces in Java will be highlighted.

Inheritance - concepts

- **every variable and method of the *superclass* is inherited by the *subclass***
- **important: constructors are never inherited**
- **root of the class hierarchy: *Object* class**
- **multiple inheritance impossible: use interfaces**
- **inherited private variables not available for subclasses**

keywords: extends

super

5.1 Inheritance

5.1.1 Concepts

The mechanism of inheritance is the object-oriented implementation of the rule that states that an object can be a special case of other objects.

The **common characteristics** are associated with the parent class or **superclass**, while **specific characteristics** are stored with the child class or **subclass**.

The behaviour of the objects of a child class is defined by the common characteristics which are inherited from the parent class, and by the specific definitions (additional or redefinition) from the derived class. The subclasses generally contain more information than their parent classes.

If you want to subclass a specific class, you need the keyword ***extends*** followed by the class you want to subclass. When you do this, you automatically get all the variables and methods of the super class.

The standard root class is ***Object***, so every time you create a class, you automatically inherit the attributes and methods of class *Object*. No keyword is required to indicate this: a class without the keyword extends in its definition assumes that the superclass is *Object*.

In Java multiple inheritance is not possible, though - as will be discussed later in this chapter - it can be simulated by using **interfaces**.

As constructors are never inherited, you are obliged to create a constructor with exactly the same number and type of arguments. The way to express this, is by using the ***super***-keyword (see: 5.2 Constructors on page 123).

Inheritance - basic example

```
public class Person {
  private String name;

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  public void saySomeThing(String someThing) {
    System.out.println(someThing);
  }
}

public class Instructor extends Person {
  private double salary;

  public void teach() {
   System.out.println("take your book");
  }
}
```

Test program

```
 public static void main(String[ ] args) {
    Instructor i = new Instructor();
    i.setName("Lecter");
    i.saySomeThing("How do you do ?");
    i.teach();
    System.out.println(i.getName());
  }
```

```
output : How do you do ?
         take your book
         Lecter
```

5.1.2 Inheritance: basic example

In the example on the opposite page, we created two different classes: the class Person and the class Instructor.

The inheritance structure is indicated by the keyword *extends*:

```
public class Instructor extends Person
```

This means Instructor is a subclass of *Person*. All variables and all methods from *Person* are inherited by the class Instructor. So, although the method *saySomeThing(String someThing)* is defined in class *Person*, I can ask an object of class Instructor to *saySomeThing(String someThing)*. This is illustrated in the little test program.

Every time you create an object of type *Instructor*, it has a *name*. Notice that **you can't code**:

```
i.name = "Lecter";
```

This is because *name* is a **private** attribute, and can only be accessed inside the class *Person*. Also notice that class *Instructor* inherits method *getName()* from superclass *Person*. This is the way we can access attribute *name*.

Method overriding

- **modifying existing behaviour of the parent class**
- **use *super* to refer to a method or member of the superclass**

```
public class Person{
    public void sayHello() {
        System.out.println("How do you do?");
    }
    public void sayGoodbye() {
        System.out.println("Have a nice day.");
    }
}

public class Instructor extends Person {
    @Override
    public void sayHello() {
        System.out.println("Welcome in this course.");
    }
    @Override
    public void sayGoodbye() {
        super.sayGoodbye();
        System.out.println("Hope to see you again.");
    }
}
```

Test program

```
    public static void main(String[ ] args) {
        Instructor i = new Instructor();
        i.sayHello();
        i.sayGoodbye();
    }
```

```
output : Welcome in this course
         Have a nice day.
         Hope to see you again.
```

5.1.3 Method overriding

By extending a class, all variables and methods are inherited. Sometimes a subclass needs specific or additional behaviour. Both super- and subclass can have **methods** which have the **same name**, and possibly the **same number and type of arguments**, but need a **different implementation**.

If the method signature (same name and same number, type and order of parameters) in the superclass and the subclass is the same, we speak about **method overriding**:

- either **modify** the **existing behaviour** of the method of the parent class;
- either **repeat** that behaviour, using for that purpose the keyword ***super*****,** and **add additional behaviour.**

The use of this *super* keyword is however, unlike with constructors, not mandatory. The *super* keyword can be used whenever you want to refer to the parent (an instance variable, a method or a constructor).

Overriding methods should adhere to the following restrictions:

- overriding methods cannot be less accessible than the corresponding parent method;
- they may not declare to throw other exceptions, but of course more specific exceptions (subclasses) are allowed (see: 6.4 Throwing exceptions on page 179);
- it is allowed for the return type of an overriding method to be a subclass of the original.

Remark:

With the advent of *annotations* in Java, you can use `@Override`. This annotation communicates to the compiler that a class is supposed to override a method from its parent. The compiler will complain if this is not true. This prevents a common mistake where, e.g. due to a typing mistake, a new method is declared instead of overriding one. Usage is shown in the example on the opposite page.

Inheritance - private methods and variables

```
public class Person {
  private String name;
  private String secretInfo = “secret“;

  public String getName() { return name; }
  public void setName(String n) { name = n; }

  private String getSecretInfo() { return secretInfo; }
  private void setSecretInfo(String info) { secretInfo = info; }

  public void tellName() {
    System.out.println(name);
  }
}

public class Instructor extends Person {
  public void tellName() {
    System.out.println(“I am instructor “ + getName());
  }
}
```

Test program

```
 public static void main(String[ ] args) {
    Instructor i = new Instructor();
    i.setName(“Lecter”);
    i.tellName();
    System.out.println(i.getSecretInfo());       // error
  }
```

```
output : I am instructor Lecter
```

5.1.4 Inheritance: private methods and variables

Variables and methods which are declared ***private*** in the superclass, are not accessible inside the subclass. This has some consequences:

- **methods**: private methods in the superclass can not be invoked from inside the subclass;.

- **variables**: private variables in the superclass can not be accessed from inside the subclass. This does not mean they are not inherited, because in fact they are! So the question arises: what can be the use of an inherited variable you can not use? Well, they are useful:

  - inherited public methods can make use of this variable: this is illustrated in the methods ***getName()*** and ***setName(String n)***;

  - moreover, you can always reach those variables making use of the get- and set-methods, if provided: this is illustrated in the override method *tellName()*.

Parent class: no-arg constructor

```
public class Person {
        private static int count = 0;
        private int pno;
        public Person() {
            pno = ++count;
            System.out.println("Creating person with pno " + pno);
        }
}

public class Instructor extends Person {
        private double salary;
        public Instructor() {}
        public Instructor(double salary) {
            this.salary = salary;
            System.out.println(salary);
        }
}
```

Test program

```
    public static void main(String[ ] args) {
        Instructor i1 = new Instructor();
        Instructor i2 = new Instructor(200.0d);
    }
```

```
output : Creating person with pno 1
         Creating person with pno 2
         200.0
```

5.2 Constructors

As we saw in 3.2.7.3 Initialization with constructors on page 79, objects are initialized by making use of constructors. The point here is: constructors are not inherited. At first sight, this makes sense: having a superclass *Person* and a subclass *Instructor*, you can not write:

```
Instructor i = new Person();
```

because a *Person* is not (always) an *Instructor*. The other way around: an *Instructor* is a *Person* is always true, so in this relationship, one can always write:

```
Person i = new Instructor();
```

So far, so good. But when we take a close look at our two classes, we see that an Instructor is a *Person*, because it is a subclass. So in some way when instantiating an *Instructor*, the object must not only be initialized as an *Instructor*, but the *Person* part of the *Instructor* must be initialized as a *Person*. How we can accomplish this, is explained in the next pages.

5.2.1 Parent class: no-arg constructor

The **superclass** contains **no constructors or a no-arg constructor** (i.e. one that does not take arguments, see example on opposite page). If no constructor is coded, the compiler will generate a default (no-arg) constructor.

Subclasses can have as many constructors as necessary. Regardless of the used subclass constructor, the **constructor of the superclass** will always be **called automatically**.

Let's again take the example of the *Instructor* class, which extends the *Person* class: each time you create an Instructor, the constructors of both *Object* and *Person* are called as well.

Parent class: no-arg and overloaded constructors

```
public class Person {
        private static int count = 0;
        private int pno;
        private String lastName;
        public Person() {
            pno = ++count;
            System.out.println("Creating person with pno " + pno);
        }
        public Person(String lastName) {
            this();
            this.lastName = lastName;
            System.out.println("Person's name is " + lastName);
        }
}

public class Instructor extends Person {
        private double salary;
        public Instructor(double salary) {
            this.salary = salary;
            System.out.println(salary);
        }
        public Instructor(double salary, String lastName) {
            super(lastName);
            this.salary = salary;
        }
}
```

Test program

```
    public static void main(String[ ] args) {
        Instructor i1 = new Instructor(200.0d);
        Instructor i2 = new Instructor(200.0d, "Poirot");
    }
```

```
output : Creating person with pno 1
         200.0
         Creating person with pno 2
         Person's name is Poirot
```

5.2.2 Parent class: no-arg and overloaded constructors

The **superclass** contains **several constructors**, one of which does not take arguments (the so-called no-arg constructor).

Even if the superclass contains a constructor with the same number, type and sequence of arguments as one of the constructors of the subclass, it is nevertheless **always** the **no-arg constructor** from the superclass that will be **called**.

It's **only when** there is an explicit reference to the constructor of the superclass, using the keyword ***super***, that a **specific constructor** will be **called** instead of the no-arg constructor.

This is illustrated in the example on the opposite page.

Parent class without no-arg constructor

```
public class Person {
        private static int count = 0;
        private int pno;
        private String lastName;
        public Person(String lastName) {
            this.lastName = lastName;
            pno = ++count;
            System.out.println("Person's name is " + lastName);
        }
}

public class Instructor extends Person {
        private double salary;
        public Instructor(double salary) {
            super("Smith");
            this.salary = salary;
            System.out.println(salary);
        }
        public Instructor(double salary, String lastName) {
            super(lastName);
            this.salary = salary;
        }
}
```

Test program

```
    public static void main(String[ ] args) {
        Instructor i1 = new Instructor(200.0d);
        Instructor i2 = new Instructor(200.0d, "Poirot");
    }
```

```
output : Person’s name is Smith
         200.0
         Person’s name is Poirot
```

5.2.3 Parent class without no-arg constructor

The **superclass** contains **several constructors**, but **none of them** is a no-arg **constructor** (or the no-arg constructor is private).

In this case, you have to provide **at least one constructor** in the subclass that refers explicitly to a constructor of the superclass using *super*. The *super* call has to pass arguments in exactly the same form as the constructor of the superclass.

The keyword ***super*** must be written in the **first line** in the **constructor**. Moreover, you can refer only once to a constructor of the superclass.

Polymorphism: concepts

Observations:

- **an object can have multiple (inter)faces, that:**
  - **represent its own class, or**
  - **represent a superclass, if its own class extends one**
- **an interface can represent objects belonging to different classes**

Polymorphism is sending the same message to objects of different classes.

5.3 Polymorphism

5.3.1 Concepts

The term polymorphism has Greek roots: poly = many, morphos = form. In the context of object-oriented programming, the notion of polymorphism refers to the fact that an **object** can have **more than one manifestation**.

In general, polymorphism means hiding different implementations behind the same interface to an object.

The **interface to an object** is the way to access that object, so it can be as well:

- the **class** itself, of course, providing the methods to access the object;
- the **possible superclasses**, as they provide methods which the class inherits from.

So, for instance:

```
Person i = new Instructor();

Person s = new Secretary();
```

The object *i* is of type *Instructor* and of type *Person* (superclass). The methods of both types are available to access *i*.

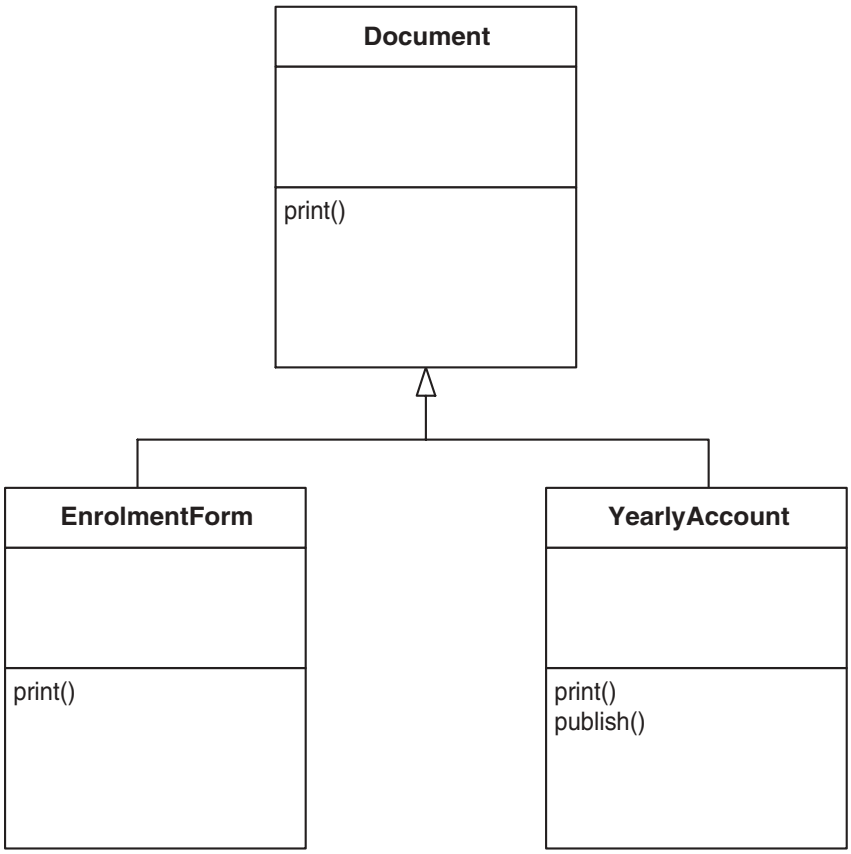
Both objects, *i* and *s*, hide behind the **same *Person* interface**. They will share some **common methods**, but these could have a **different implementation.**

So, as you see, objects can behave differently, depending on their type. Implementations can be defined in several classes, allowing objects derived from those classes to handle the same functionality in different ways. It is not necessary to know the type of object to send a message: the fact that the receiving object will understand the message and will be able to handle it, is sufficient. Polymorphism thus is the ability to send the same message to objects of different classes.

The mechanism which provides for polymorphic behaviour is called dynamic binding, also called late binding or runtime binding.

Polymorphism: example

Figure 5: Polymorphism: example




```
public class OutputDevice {
    public void output(Document do) {
        do.print();
    }
}
```

5.3.2 Use of polymorphism

The example on the opposite page shows how polymorphism can be used. As you can see, we defined a class Document. This class specifies some general behaviour and attributes for specific subclasses, in this case specific documents: *EnrolmentForm* and *YearlyAccount*.

In this example, we assumed that each document can print itself, this means: has the knowledge about its content (of course ...). So it is obvious that not the class *Document* itself will specify how a *Document* should be printed, but that each subclass itself will specify this. The class *Document* only specifies that a specific subclass must implement the method print.

What is the advantage of this approach? This becomes clear if you look at the class *OutputDevice*. An *OutputDevice* can *output()* a *Document*. The way this happens is defined inside the specific subclass of *Document*, but the funny part is that you do not have to know this in the class *OutputDevice*. You just ask the *Document* (be it an *EnrolmentForm* or a *YearlyAccount*) to print itself. If the specific object passed is an *EnrolmentForm*, it will print itself as an *EnrolmentForm*, and if it is a *YearlyAccount*, it will print itself as a *YearlyAccount*!

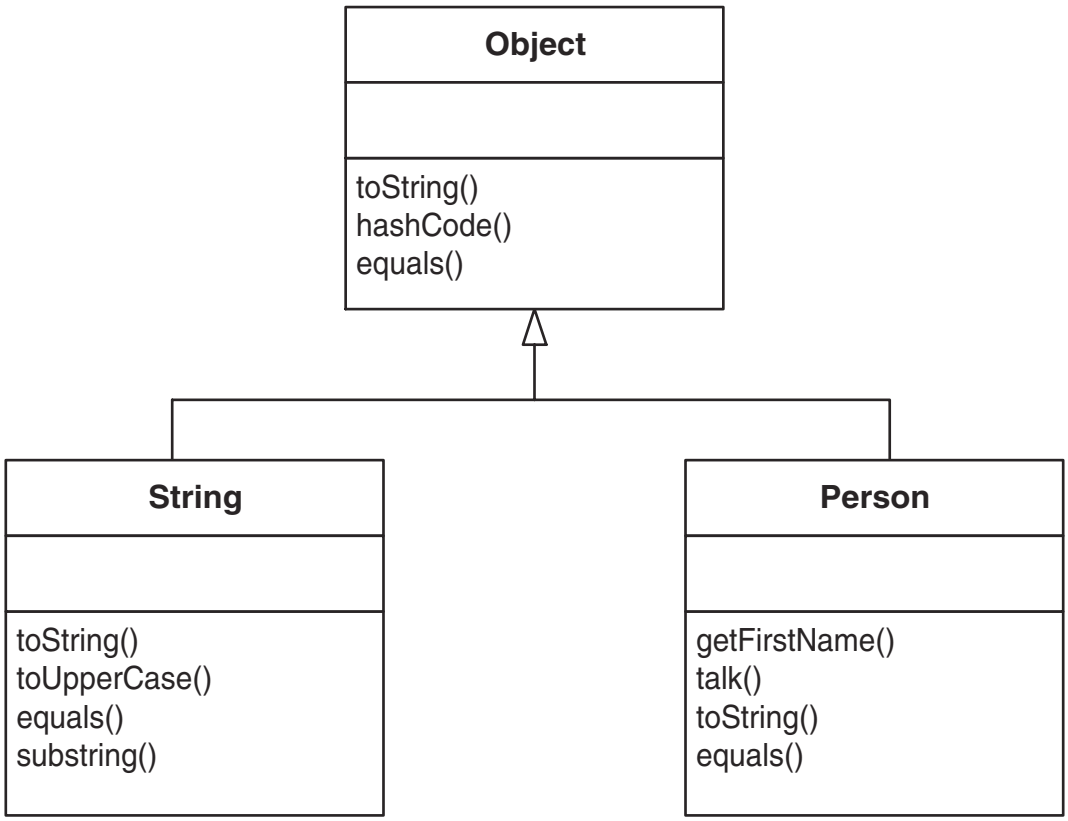
The figure on the opposite page shows clearly is the nature of polymorphism: **different objects** can hide themselves behind **the same interface** (in this case **Document** is the **interface to** the object ***do***), and can **behave in different ways**.

Note:

The common behaviour (in this case the method *print()*) must be defined in the superclass. If this is not the case, you can not ask to a generic object of type *Document* to print itself.

Polymorphism: toString()

Figure 6: toString()



default implementation toString() in class Object

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Example

```
Person p = new Person("James","Bond");
System.out.println("I am: " + p);
```

```
I am: Person@15db9742
```

custom implementation toString() in class Person

```
@Override
public String toString() {
    return "Person " + firstName + ", " + lastName;
}
```

```
I am: Person James, Bond
```

5.3.3 toString()

A well known case of polymorphism is the toString() method. This method, which returns a String object, is used a lot

- for debugging purposes
- for visualising an object
- for printing

The standard implementation in the class Object

```
public String toString()
```

returns a string representation of the object, which should be a concise but informative representation that is easy for a person to read, i.e.

- the name of the class of which the object is an instance,
- the at-sign character `@', and
- the unsigned hexadecimal representation of the hash code of the object.

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

It is advised that all subclasses override this method.

The example on the opposite page shows an overridden toString() method in the Person class. The output is more meaningful.

Note: equals() and hashCode()

Also the equals() method, for checking object equivalence, and the hashCode() method, for generating a consistent object identifier, are candidates for polymorphism. This will be discussed a.o. in 7.2 The Collections Framework on page 197.

Upcast and downcast

```
public class Executable {
    public static void main (String[] args) {
        EnrolmentForm ef = new EnrolmentForm();
        YearlyAccount ya = new YearlyAccount();
        Document[] docs = {ef,ya};
        OutputDevice od = new OutputDevice();

        for (Document d : docs) {
            od.output(d);
            if (d instanceof YearlyAccount) {
                ((YearlyAccount) d).publish();
            }
        }
    }
}
```


5.3.4 Casting

Polymorphism does not make downcasting redundant. In specific circumstances, you are forced to do it. This is the case whenever for any reason upcasting has occurred and the object is calling for a method that does not exist in the class to which it has been upcast before: this method is not accessible at that moment, so you have to perform a specific downcast.

This may happen when you make use of general purpose classes such as classes for input- and outputstreams (see Chapter 9. Input/Output on page 231) or the Java reflection API. For now we are going to give an arbitrary but simple example (on the opposite page).

We create an instance of both an EnrolmentForm and a YearlyAccount. We want to print both documents in one go (imagine hundreds of them at the same time). To accomplish this, we gather them in an array of type Document. This way polymorphism will do its job and the OutputDevice will output each document in its own specific way.

While we are processing documents, one wants to immediately publish the YearlyAccount(s). Since at that moment, the YearlyAccount was upcasted to Document, this is not possible. So we have to cast it back to its correct type before we can call the method.

The **instanceof** operator is used to **test** whether a cast is legal.

Final methods

- to prevent methods from being overridden
- use for:
  - final version
  - security reasons
  - optimization

```
public class Person {
        public final void printName(String aName) {
            System.out.println(aName);
        }
}
public class Instructor extends Person {
        public void printName(String aName) {
            System.out.println(aName.toUpperCase());
        }
}
```

```
Instructor.java:2: printName (java.lang.String) in Instructor cannot
override printName (java.lang.String) in Person; overridden method is final
   void printName (String aName) {
        ^
1 error
```

5.4 Modifiers final and protected

5.4.1 Final methods

A method that is declared **final** cannot be overridden in a subclass. As the example shows, this is taken care of by the compiler at compile time. **Reasons** to mark a method as final might include:

- **completeness**: because it is the 'final' method of a class, no modifications are necessary - this method is well defined for all subclasses that are derived from it;
- **security**: overriding may not always be advisable as this can make code insecure, e.g. for a method that validates a password, the method you wrote will do what is advertised, instead of always returning true;
- **optimization**: for non-final methods the actual class of an object is determined at runtime, next the method invocation is bound to the correct implementation of the method for that type, and finally the method's implementation is invoked. When declaring a method final, the runtime environment will immediately execute the implementation, and skip the steps to invoke the method. This mechanism is known as "inlining code" and improves performance. The major benefit of this mechanism is that you can hide variables for the outside world, and yet have code that is very fast.

Final classes

- **to prevent classes from being subclassed**

```
public class Person {}

public final class Instructor extends Person {}

public class JuniorInstructor extends Instructor {}
```

```
JuniorInstructor.java:1: cannot inherit from final Instructor
class JuniorInstructor extends Instructor {}
                               ^
1 error
```

- **final classes <--> class with all methods final**

5.4.2 Final classes

The main benefit of making a class final is that it can't be subclassed any longer: the java runtime environment will throw an error at compile time if you do. The main benefits are already mentioned above:

- **security**: others can not redefine the methods in this class;
- **optimization**: inlining from code.

You must be aware of the fact that making a class final, makes your code less flexible.

Another way to prevent others from overriding methods defined in your class, is making every method final, without making the class itself final. By doing so, others can still extend your class, without overriding the methods you defined.

Protected members (I)

- only applicable to fields and methods
- the field or method can be accessed
  - by a subclass (even in another package)
  - from within the same package (= package friendly)

5.4.3 Protected variables and methods

5.4.3.1 Concepts

When discussing access control of Java building blocks, ***protected*** was mentioned. At that moment we preferred not to explain it in detail, as it is related to inheritance.

You can access protected fields and methods in two ways**:**

- you can access them **from subclasses**, and **even** when those subclasses reside **in another package**: In this case, you can only access these fields and methods through a reference to the own class, and not via another class in that package, even if this class is a sibling class (this will be treated on the next page *super- and subclasses in different package)*;

- you access them **from within the same package** through a reference to an object of that class (*unrelated classes in same package*).

Classes cannot be made protected (except for inner classes).

Protected members (II)

super- and subclass in different package

```
package be.abis.companies;
public class Person {
        protected int personNr = 2250;
        private int companyNr = 1500;
        protected void printName(String aName) {
            System.out.println(aName);
        }
}
```

```
package be.abis.customers;
public class Client extends be.abis.companies.Person {
        public void printOut() {
            System.out.println("personNr = " + personNr);
            System.out.println("companyNr = " + companyNr);    // error
            this.printname(“Bond”);
        }
}
```

```
Compilation errors :

companyNr has private access in be.abis.companies.Person
```

5.4.3.2 Super- and subclass in different package

The example on the opposite page gives an illustration.

- First, the extending class, *Client*, has a *personNr* (personal number) and a *companyNr* (company number). However, the member *companyNr* cannot be accessed by the subclass, as it is declared to be *private*. Only *Person* instances can obtain information of that variable. So in this example there is no way to access the variable directly (unless you define public (or in this case also protected) get/set-methods for the attribute).

- Secondly, the variable *pno* is of type *protected*. Two consequences:

  - a *Client* has a *personNr* and that variable can be accessed;

  - the *personNr*-variable of the *Person* remains inaccessible for classes which make object of this class in other packages.

The consequences are valid for the protected method *printName()* as well.

Protected members (III)

unrelated classes in same package

```
package be.abis.companies;
public class Person {
        protected int pno= 2250;
        private int cono = 1500;
        protected void printName (String aName) {
            System.out.println(aName);
        }
}

/* Note: the class Company is NOT a subclass of Person */

package be.abis.companies;
public class Company {
        Person cp = new Person();
        public void printOut() {
            System.out.println("pno = " + cp.pno);
            System.out.println("cono = " + cp.cono);        // error

            cp.printName("Bond");

        }
}
```

```
Compilation errors :

cono has private access in be.abis.companies.Person
```

5.4.3.3 Unrelated classes in same package

In the example shown, two classes, *Person* and *Company*, belong to a same package. The variable *companyNr* is private. It cannot be read or written, and of course it is not a characteristic of class *Company*.

As to the variable *personNr*: it only characterizes a *Person*, not a *Company*. However, it can be accessed from the class *Company,* but, only if that class contains a valid reference to an instance of class *Person*.

The consequences are valid for the protected method *printName()* as well.

Abstract classes and methods

- **definition**
  - **a *class* which can't be instantiated**
  - **a *method* which isn't implemented**
- **purpose**
  - **to define only a part of implementation**
- **abstract class mandatory when one abstract method**
- **abstract classes possible without abstract methods**

5.5 Abstract classes and interfaces

5.5.1 Abstract classes and methods

An **abstract class** is a class that **can not be instantiated**, so you can never use the keyword *new* to instantiate an object of this class.

An **abstract method** is only declared. It has **no method body** and, thus, no implementation. If only one method is abstract in a class, the class itself must be declared abstract. Otherwise you will get an error at compile time.

It is possible to create an abstract class without abstract methods (i.e. none of its methods is abstract). This is useful when you want to prevent creating instances of that class. Also, you can give an implementation to some of the methods in an abstract class, provided that those methods are not declared abstract.

Note that **abstract classes** are used to establish a **basic type**, allowing you to specify **common properties** of a series of derived classes. In this case, make sure that the abstract class has no concrete meaning, so that the need to create instances of it is non-existent.

> For instance, an *Animal* object has no concrete meaning: it is an abstraction we make of what is common in different kinds of animals (a horse, a frog, a salmon,...).

Abstract classes

```
public abstract class Document {
    private int documentNumber;
    private static int count = 0;

    public Document() {
        documentNumber = ++count;
    }

    public int getDocumentNumber() {
        return documentNumber;
    }

    public abstract void print();
}

public class EnrolmentForm extends Document {
    public void print() {
        System.out.println("enrolment form number " +
                                                getDocumentNumber());
    }
}
```

Test program:

```
    public static void main(String[ ] args) {
        Document doc = new EnrolmentForm();
        doc.print();
    }
```

In the example on the opposite page, the class *Document* is declared abstract. This is because, according to the analysis of our application, instances of *Document* have no concrete meaning. The *Document* class only serves to define behaviour that is common to specific subclasses (for example the class *EnrolmentForm*).

In the *Document* class one method is declared abstract: the method *print()*. This is because the information of how exactly a specific form should be printed is only available in the subclasses. This means that every subclass must implement this method *print()*.

Because every *Document* will have a *documentNumber*, and the implementation of the *getDocumentNumber()* method will not change among the different subclasses, this method is already implemented in the class *Document*.

Note:

- abstract methods have **no curly braces**, not even empty ones;
- although you can not use the **constructor** of the abstract class *Document* directly, for example by writing:

  Document doc = new Document(); (is not possible!)

  you can define one in this class to do **some general initialization** for a specific *Document*.

Interfaces

expression of pure design <--> implementation

creation of interface:

- **keyword *interface***
- **package friendly (default) or *public***
- **interface can extend other interfaces**
- **fields: always *public, static* and *final* (by default)**
- **methods: must always be defined *public***
  - **only declarations (*abstract*)**
  - **default methods (Java SE 8)**
  - **static methods (Java SE 8)**
- **eventually inner classes - enums - interfaces**

use of interface:

- **a class can implement interfaces**
  - **keyword: *implements***
- **every (abstract) interface method must be implemented!**

5.5.2 Interfaces and methods

Like classes, **interfaces** establish types including: method names, argument lists and return types, but no method bodies. So interfaces are pure design, a kind of **templates**. Implementation is totally absent.

The **creation** of an interface:

- you use the ***interface*** keyword, not *class;*
- you can let an interface **package friendly** (no keyword) or make it ***public***;
- an interface **can extend other interfaces**;
- an interface can contain **data members**, which are ***public***, ***static*** and ***final*** (no blank finals!), even if you don't provide these keywords (which is obvious, since interfaces don't provide implementation);
- **methods** specified in interfaces can
  - have **no body** and, at that time, are always ***abstract*** and ***public*** (even if the keywords aren't specified);
  - have **a body** and, are indicated as **default** (public) methods (available since Java SE 8)
  - be defined static (available since Java SE 8)
- an interface can also contain inner classes (member classes) and enums (or even interfaces): member classes or interfaces are always *public* and *static*. They need not be final nor abstract.

The **use** of an interface:

To create a **class** that conforms to a particular interface (or - which is possible as well - multiple interfaces) you use the ***implements*** keyword. A class that implements an interface has to give a concrete implementation for every (abstract) method that is declared by the interface. A class that implements an **interface**, remains an ordinary class that may be extended and instantiated in the regular manner. However, the instances of such class are simultaneously of the type established by the class and the implemented interface(s), thus, resembling objects produced by multiple inheritance (which is not supported in Java).

Note that **every abstract method** declared in the interface **must be implemented** in the class (you can use an empty method body), otherwise the class should be declared abstract. Moreover, each method needs the ***public*** keyword. Without it, the method would be package friendly and then you are restricting the accessibility of a method during (a kind of) inheritance. This is not allowed by the Java compiler.

Default methods (Java SE 8) may be overridden in the implementing class.

Static methods (Java SE 8) must not be overridden in the implementing class.

Note:

An interface doesn't need to declare any methods. An **interface without any body** is called a **marker interface**. It is used to "mark" a class which supports a certain capability e.g. the interface ***Serializable*** (see 9.6 Object streams on page 255)

Default and static interface methods

since Java SE 8

```
interface Document {
    //  abstract method
    public int getDocumentNumber();

    //  default method
    default public void print() {
        System.out.println("enrolment form number " +
                                getDocumentNumber());
    }
    //  static method
    static public void log() {
        System.out.println("enrolment form created.”);
    }
}

public class EnrolmentForm implements Document {
    private int documentNumber;
    private static int count = 0;

    public EnrolmentForm() {
        documentNumber = ++count;
    }

    public int getDocumentNumber() {
        return documentNumber;
    }

    // no need to override the print() method !
    // not possible to override the log() method !
}
```

5.5.3 Default interface methods

Java SE 8 allows interfaces to contain *public default methods* (aka *defender methods*) with concrete default implementations. The implementing class will still be able to override these methods, if needed.

In order to declare a default method, the **default** modifier keyword should be placed at the beginning of the method signature.

We have reproduced the previous example using an interface *Document* rather than an abstract class (see Abstract classes on page 148). A consequence of this approach is that the *getDocumentNumber()* method can no longer be implemented at this level since all methods of an interface are implicitly abstract.

EnrolmentForm implements the interface *Document*, and, as a result, every method declared in the Document interface **must** be given concrete code, i.e. a method body. The default interface method print() does not need to be implemented, but may be overridden.

Note:

For classes that implement the original interface, without the default method, the *contract* will not break when a default method is added — the class simply receives the new default method.

5.5.4 Static interface methods

Java SE 8 allows as well interfaces to contain *static methods*, with concrete implementations. The implementing class can not override these methods.

As is the case for other static methods, you can use interface static methods using the class name.

Abstract classes vs interfaces

Table 10: Interfaces vs abstract classes

Abstract classes	Interfaces
only single inheritance	simulation multiple inheritance
partial implementation possible	abstract definition default method implementation
mostly in top of class-hierarchy	can be stand-alone

Simulation of multiple inheritance via interfaces

Example:

```
public class SmartPhone extends Phone implements Camera, Computer
```

5.5.5 Abstract classes and interfaces

Just like an abstract class, interfaces provide a way to define abstract behaviour for classes which implement them, though there are some major differences.

The table on the previous page illustrates the **related semantics** of abstract classes and interfaces. When further extending a class that has implemented an interface, the obligation to provide a method body for each method declared in the interface is dropped. The reason is obvious: this class inherits all the methods, including their implementation, from its superclass.

Notice that all methods in an interface are public, hence, they are explicitly said to be *public* in the classes implementing them. Other methods can be added to each class, of course, leaving you free to choose their accessibility.

5.5.6 Interfaces and multiple inheritance

Java does not support multiple inheritance of *classes*: it is not possible for a single class to extend more than one class. On the other hand, Java does support multiple inheritance of *interfaces*, enabling you to **simulate multiple inheritance**.

An example is shown on the opposite page.

Typesafe enums and inheritance (I)

enum is more than just constants:

- **an enum type:**
  - **can have static fields and methods**
  - **may implement interfaces**
  - **can have a constructor: always private**

- **enumeration elements (= instance):**
  - **can have fields and methods**
  - **are sole instances of anonymous subclasses**

```
public enum FileCommand {
    OPEN_FILE,
    CLOSE_FILE;
}
```


5.6 Typesafe enums and inheritance

Typesafe enumerations, as introduced in 2.7.3.1 Typesafe enums on page 41, are not merely simple enumerations of constants as in e.g. C++. Enums combined with the concepts of inheritance and inner classes produce a much more powerful construct.

As an enum is in fact a class, all enums can have both static and instance fields and methods. An enum can even implement interfaces.

The **elements of the enumeration** are themselves **instances of subclasses of their enum type**. In fact, each one of them is the only instance (singleton) of an anonymous inner class, extending the enum type. Therefore, the enumeration's elements **can have state and behaviour of their own**.

This sounds - and is - pretty complex. An example will make it a lot clearer.

5.6.1 Enum example: basic

The example on the left depicts a simple enumeration describing the actions that can be executed by some application. The application is very simple, there are only two actions: it can open or close a file. (e.g. a file viewer).

Suppose that we have a GUI where a user can select an *Open* command in a menu. When this happens, the OPEN_FILE element is passed to the application logic and e.g. a switch statement can determine what will happen. By the way, this is not the way Swing GUIs work.

Now, this is not very useful, let's explore some more advanced features of Java's enumerations.

Typesafe enums and inheritance (II)

```
public class Environment { /* describes current application state */ }

public enum FileCommand {
    OPEN_FILE,
    CLOSE_FILE;

    public void execute(Environment e) {
        switch (this) {
        case OPEN_FILE:
            // use environment to open a file
            break;
        case CLOSE_FILE:
            // use environment to close a file
        }
    }
}
```

Usage:

```
Environment env = new Environment();
FileCommand command = FileCommand.OPEN_FILE;
command.execute(env);
```


5.6.2 Enum example: adding behaviour

The example on the left adds some behaviour to the commands. The commands contain the logic for executing whatever they're supposed to do. An *Environment* argument describes the application's state, e.g. the working directory and a pointer to the current file or whatever might be needed to execute the command under the actual conditions.

It is now possible to call the execute method on each of the constants of the enumeration.

It is also possible to add static methods. Or you could even implement the methods of an interface, just as you would do in a normal class.

Typesafe enums and inheritance (III)

```
public class Environment { /* describes current application state */ }
public enum FileCommand {
    OPEN_FILE("Open File"),
    CLOSE_FILE("Close File");

    private String description;

    FileCommand(String description) {      // enum constructor (which can
        this.description = description;     // have multiple arguments)
    }

    public String getDescription() {
        return this.description;
    }

    public void execute(Environment e) {
        switch (this) {
        case OPEN_FILE:
            // use environment to open current file
            break;
        case CLOSE_FILE:
            // use environment to close current file
            break;
        }
    }
}
```

Usage:

```
Environment env = new Environment();
FileCommand command = FileCommand.OPEN_FILE;
String desc = command.getDescription();
command.execute(env);
```

5.6.3 Enum example: adding constructors and state

The example on the left adds a private member. Each instance of *FileCommand* will have its own private field description, therefore each enumeration constant can have its own value for the description field.

To initialize the value of this field, we'll need a constructor. The **constructors** for enum types are **always private**, even if you don't mention the modifier.

Now there's the problem of calling this constructor: the elements are instantiated when the enum class is loaded and these elements are final.

A **special notation** is introduced to pass in arguments for each of the constants. The name of each constant is followed by the list of values that should be used for the constructor call.

Finally, the example then adds a getter to retrieve the description for a certain command.

If you have a very good reason for it, it is also possible to add setters, or other methods that modify the state of a constant. Be very careful when doing so. Probably it would be better to reconsider the design.

Typesafe enums and inheritance (IV)

```
public class Environment { /* describes current application state */ }

public enum FileCommand {
    OPEN_FILE("Open File") {
        public void execute(Environment env) {
            // use environment to open the current file
        }
    },
    CLOSE_FILE("Close File") {
        public void execute(Environment env) {
            // use environment to close the current file
        }
    };

    private String description;
    FileCommand(String description) {
        this.description = description;
    }
    public String getDescription() {
        return description;
    }

    public abstract void execute(Environment env);
}
```

Usage:

```
Environment env = new Environment();
FileCommand command = FileCommand.OPEN_FILE;
String desc = command.getDescription();
command.execute(env);
```


5.6.4 Enum example: constant specific behaviour

In this final case we go even beyond just calling the constructor. We declare a complete new (anonymous) subclass of *FileCommand* on the fly. There will never be more than one instance of this type, namely `FileCommand.OPEN_FILE`. This subclass overrides one method, the method that holds the logic **specific** to this particular command.

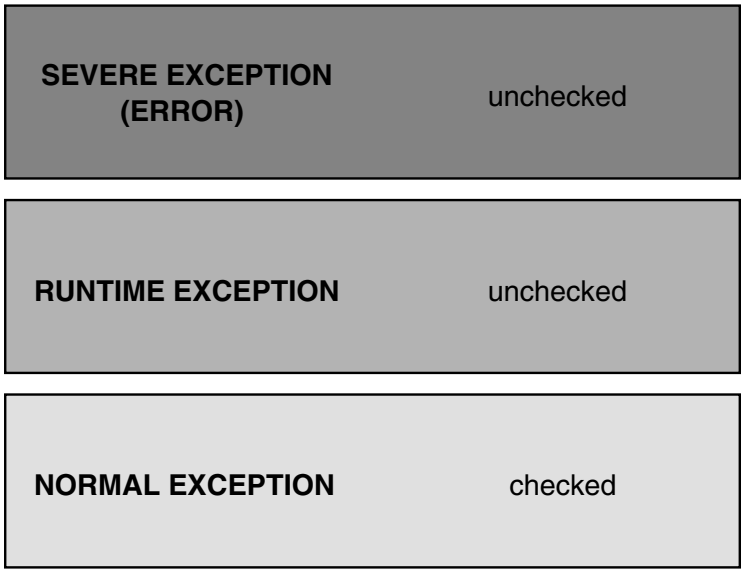
In order to use a method overridden by an enum constant, it should also be declared (in this example abstract) in the enum itself. The enum type - *FileCommand* - is what a user of the class is aware of, therefore it is only possible to use the method if it is also declared in this class.

CHAPTER 6. EXCEPTION HANDLING

Java has developed a convenient mechanism to handle exceptions. As this is done in an object-oriented manner, it doesn't make your code illegible. This chapter discusses the Java exception handling mechanism, starting off with an overview of the different types of exceptions and handling and throwing exceptions. Afterwards, it is shown how to create your own exception classes.

Types of exceptions - overview

Figure 7: Types of exceptions



6.1 Exception definition

Three categories of exceptions are distinguished:

- **errors** or “severe” exceptions;
- **runtime exceptions;**
- **normal exceptions**.

Severe exceptions and runtime exceptions are also called

unchecked exceptions;

normal exceptions are called

checked exceptions.

The reason for this qualification will become clear later on in this chapter.

Errors

- **severe problem (with Java Virtual Machine (JVM))**
- **should not be handled**
- **examples:**
  - running out of memory
  - instantiating an abstract class or interface

```
abstract class Person {}

public class Test {
    public static void main(String[ ] args) {
        Person p1 = new Person();
    }
}
```

```
Test.java:3: Person is abstract; cannot be instantiated
Person p1 = new Person()
            ^
1 error
```

6.1.1 Errors

An error is a **serious mistake**.

It occurs in a number of circumstances, including the following:

- the application calls an abstract method (*AbstractMethodError*);
- the JVM attempts to read a class file which is malformed or cannot be read (*ClassFormatError*);
- the application attempts to access or modify a field or to call a method that it does not have access to (*IllegalAccessError*);
- the application tries to create an instance of an abstract class or an interface (*InstantiationError*);
- an internal error occurs in the JVM (*InternalError*);
- the application tries to access or modify a specified field of an object or a specified method of a class and that object no longer has that field or that class no longer has that method (*NoSuchFieldError*, *NoSuchMethodError*);
- the JVM cannot allocate an object because it is out of memory (*OutOfMemoryError*);
- the JVM attempts to read a class file and determines that the major and minor version numbers in the file are not supported (*UnsupportedClassVersionError*);
- the verifier detects that a well-formed class file contains some sort of internal inconsistency or security problem (*VerifyError*).

As can be noticed, these mistakes are very severe and it wouldn't be a wise idea to recover from the error and let the application continue. That's why errors are considered to be **unchecked** exceptions.

Runtime exceptions

- **indication of design- or implementation problem**
- **should not be handled**
- **examples:**
  - array out of bounds
  - ClassCastException

```
class Person {}
class Instructor extends Person {
    private String name;
    public void setName(String aName) {
        name = aName;
    }
}

public class Executable {
    public static void main(String[ ] args) {
        Person p1 = new Person();
        Instructor i1;
        i1 = (Instructor)p1;
        i1.setName("Michael");
    }
}
```

```
Exception in thread “main” java.lang.ClassCastException : Person
at Executable.main(Executable.java:5)
```


6.1.2 Runtime exceptions

Runtime exceptions can occur during the normal operation of the JVM.

A few examples:

- an exceptional arithmetic condition has occurred (e.g. divide by zero) (*ArithmeticException*);
- an attempt has been made to store the wrong type of an object into an array of objects (*ArrayStoreException*);
- the code has attempted to cast an object to a subclass of which it is not an instance (*ClassCastException*);
- a method has been passed an illegal or inappropriate argument (*IllegalArgumentException*);
- an index of some sort is out of range (*IndexOutOfBoundsException*);
- an application attempts to use null in a case where an object is required (*NullPointerException*).

As these examples illustrate, this kind of exceptions are thrown whenever there are **mistakes in the code**. They shouldn't occur in an application which is considered to be in its final version. That's why these exceptions **should not be treated**, shouldn't be recovered from. These are **unchecked** exceptions as well.

Normal exceptions

- **caused by environmental effects**
- **should be handled**
- **examples:**
  - user typing error in the command line argument
  - trying to open a file that doesn't exist

```
import java.io.*;

public class OpenFile {
    public static void main(String[ ] args) throws FileNotFoundException {
        FileInputStream fi = new FileInputStream(args[0]);
    }
}
```

```
Exception in thread "main" java.io.FileNotFoundException: c:/Filename.txt
(The system cannot find the file specified)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(FileInputStream.java:68)
at OpenFile.main(OpenFile.java:5)
```


6.1.3 Normal exceptions

Normal exceptions are **exceptional situations** that may occur. They aren't too severe and they couldn't be avoided by writing better code.

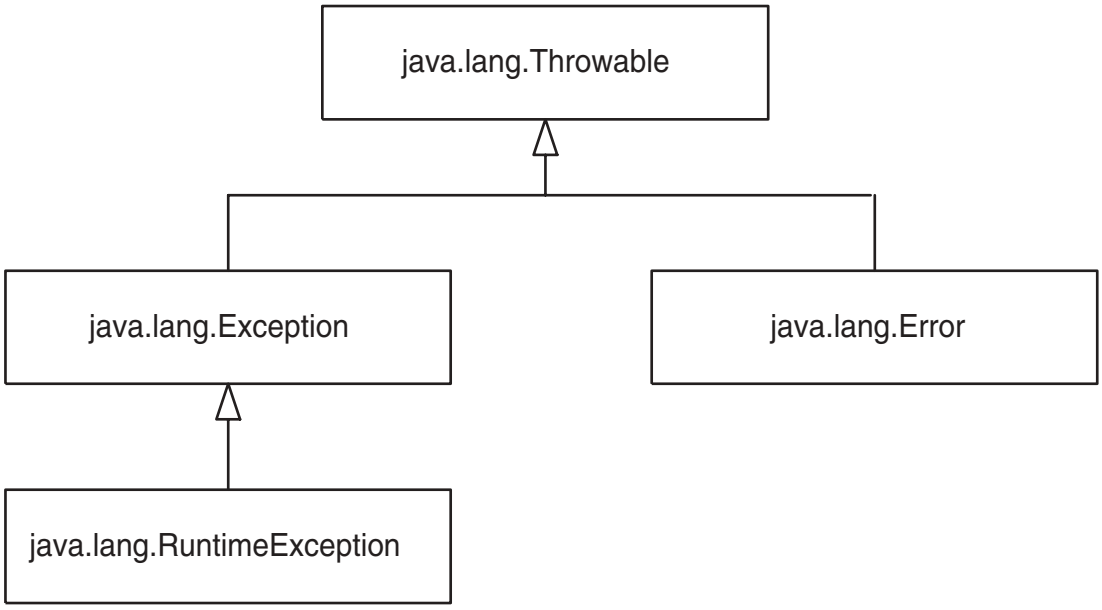
Normal exceptions are called **checked** exceptions: **you will have to check them**, **do something with them** (declaring or handling them).

Let's take a look at some of these situations.

- an application tries to load in a class through its string name using the *forName()* method in class *Class* (*ClassNotFoundException*);
- the *clone()* method in class *Object* has been called to clone an object, but the object's class does not implement the *Cloneable* interface (*CloneNotSupportedException*);
- an application tries to load in a class that it does not have access to through its string name using the *forName()* method in class *Class* (*IllegalAccessException*);
- a thread is waiting, sleeping, or otherwise paused for a long time and another thread interrupts it using the *interrupt()* method in class *Thread* (*InterruptedException*);
- the class doesn't have a field or a method of a specified name, field or method that are accessed through the methods *getField()* and *getMethod()* of the *Class* class (*NoSuchFieldException* - *NoSuchMethodException*);
- an application tries to open a file denoted by a specified path name but has failed (*FileNotFoundException*).

classes concerning the different exception types

Figure 8: Exception classes



more exceptions in other packages as well (e.g. java.io.*)

6.2 Exceptions hierarchy

The exception handling mechanism is, in the Java approach, object-oriented. This means that **Java** consistently **treats exceptions as objects** described by an *Exception* class.

All exceptions are derived from class ***Throwable***, which is situated in the *java.lang-package*. This is the package where most of the exception classes can be found. Two derived classes are the ***Exception*** class, where the normal and runtime exception classes are derived from, and the class ***Error***.

RuntimeException is a subclass of *Exception* and all its subclasses are runtime exceptions.

Other exceptions and errors can be found in other libraries as well. Consult the documentation for a list of these *Exception-* and *Error* classes.

create subclass of java.lang.Exception

```
public class SalaryCalculationException extends Exception {
    public SalaryCalculationException() {
            super();
    }

public SalaryCalculationException(String msg) {
            super(msg);
    }

public SalaryCalculationException(Throwable thr) {
            super(thr);
    }
}
public SalaryCalculationException(String msg, Throwable thr) {
            super(msg, thr);
    }
```

6.3 Creating your own exceptions

6.3.1 Why?

Good reasons to create own exception types include:

- you feel the need to **add useful data** when throwing the exception;
- the **type** of the exception forms an **important part of the exception data**, because exceptions are caught according to their type;
- you want to **handle** one **kind of error** and not another kind.

6.3.2 How?

When consulting the Java documentation you will find a lot of predefined exceptions and errors. However, sometimes you may need to create your own exceptions. This can be achieved by extending the *java.lang.Exception* class or *java.lang.Error* (or one of their subclasses). By doing so you can also define your own constructors if necessary. Existing methods can be overridden, new methods can be created.

Let's take a look at a few *Exception* methods (or better: the *Throwable* methods, as *Exception* has no specific methods).

- there is - besides the default constructor - another one with a String argument, which initializes the message field: that message can be asked for with the ***getMessage()*** method;
- constructors can also be used to chain exceptions: they take a *Throwable* as argument, which is the cause of the exception, and the cause can later be asked for with the ***getCause()*** method;
- a class of type *Throwable* always contains a snapshot of the execution stack of its thread at the time it was created, the so-called **stack trace**, which can be manipulated as well - this information can be shown with one of several ***printStackTrace()*** methods: several since you can define the output streams by means of arguments;
- a short description of the *Throwable* class is returned by the ***toString()*** method.

In the example on the opposite page we defined a class *SalaryCalculationException*. This is the exception class we will use in 6.4 Throwing exceptions on page 179.

Throwing exceptions

- **keywords: *throw* + *new***
- **declare exception in method declaration**

```
public class Person {
    private double salary, maximumSalary;
    public void calculateSalary() throws SalaryCalculationException {
        /* perform some calculation */
        if ( salary > maximumSalary ) {
            throw new SalaryCalculationException(“earns too much”);
        }
    }
}
```

6.4 Throwing exceptions

Exceptions are just normal java objects, so to create an object you have to use the ***new*** keyword. But there is one big difference: this object must be thrown. If you want to throw an exception, you have to use the keyword ***throw***. This is illustrated on the opposite page: in the **method** *calculateSalary()* of the class *Person*, we throw an exception if someone earns more than a particular amount of money.

When something goes wrong inside a method, it is the responsibility of the caller of this method to undertake an action. But how does the caller know whether something can go wrong, and moreover, how does he know it is his responsibility to perform some action? This is quite simple: you declare in the method definition that something can go wrong. In our example we wrote in the definition:

throws SalaryCalculationException

The caller of the method now knows something can go wrong and must undertake an action. What kind of action is specified further in this chapter.

Unchecked Exceptions

- RuntimeExceptions, Errors could be handled
- declaring automatically done by Java
- disadvised

Checked Exceptions

- normal exceptions need declaring or handling
- both for own exceptions as for some predefined exceptions (e.g.`FileNotFoundException`, `SQLException`)
- compile time check

6.5 Handling and passing exceptions

6.5.1 Concepts

Handling (or catching) an exception refers to the action undertaken by the programmer in order to solve the exception. When an exception occurs (in Java parlance: “is thrown”), an exception object is created. Handling the exception consists of “catching” this object in a catch-block. In such a block you can provide any code to deal with the exception e.g. announce to the user that something went wrong, or propose that an action be undertaken in order to solve the problem.

Declaring an exception refers to notifying that a particular method may generate (throw) an exception. This means that a specified problem can occur when someone uses this method or constructor. Using such a method implies that you catch the exception in a catch-block, or declare it in the header of the method. So with this solution, you don’t take care of the exception, but throw it further to the caller of the method.

6.5.2 Handling exceptions

6.5.2.1 Unchecked exceptions

It is **discouraged to handle errors and runtime exceptions**: since the problem is too severe, the user will not be able to solve it in a decent way.

Declaring an error or a runtime exception can be done without any problem but as handling them would not be a wise thing to do, why declare them?

For instance, catching an *ArrayIndexOutOfBoundsException* is useless, because the **developer** should change the code in such a way that he never passes the array boundaries.

Exception handling - catching exceptions

```
public class Company {
    Person[ ] employees = new Person[10];

    public void paySalaries() {
        for (int i = 0; i <= 10; i++) {
            try {
                employees[i].calculateSalary();
            }
            catch (SalaryCalculationException sce) {
                System.out.println(sce.getMessage());
            }
        }
    }
}
```

Notes:

- no code between try- and catch-block
- catch block without implementation allowed (but not very smart)
- catch block can re-throw the same (or another) exception

6.5.2.2 Checked exceptions

Normal exceptions do need declaring or handling. E.g. you ask the user to type the name of a file he or she wants to open, but a non-existing file is entered. This leads to an exception, but one that can be corrected. If you don't want to correct (i.e. catch) it, you can limit yourself to declaring, i.e. announcing it.

Whenever you write code that may give cause to an exception, the compiler forces you to do something with that code.

As mentioned earlier, Java has created an exception model with a rigorous object-oriented structure. Each exception is an instance of a class. As soon as an exception occurs, an instance is created and is passed to the code block where the programmer is stating what should be done next.

The **code that could generate an exception**, should be written in a **try-block** (between {} and prefixed by the keyword try).

The **action to be undertaken if an exception-instance is created**, is coded in the **catch-block**. The exception instance, created automatically when an exception occurs, is passed as an argument to the catch-block, where it can be queried for information.

See the example on the opposite page.

Notes:

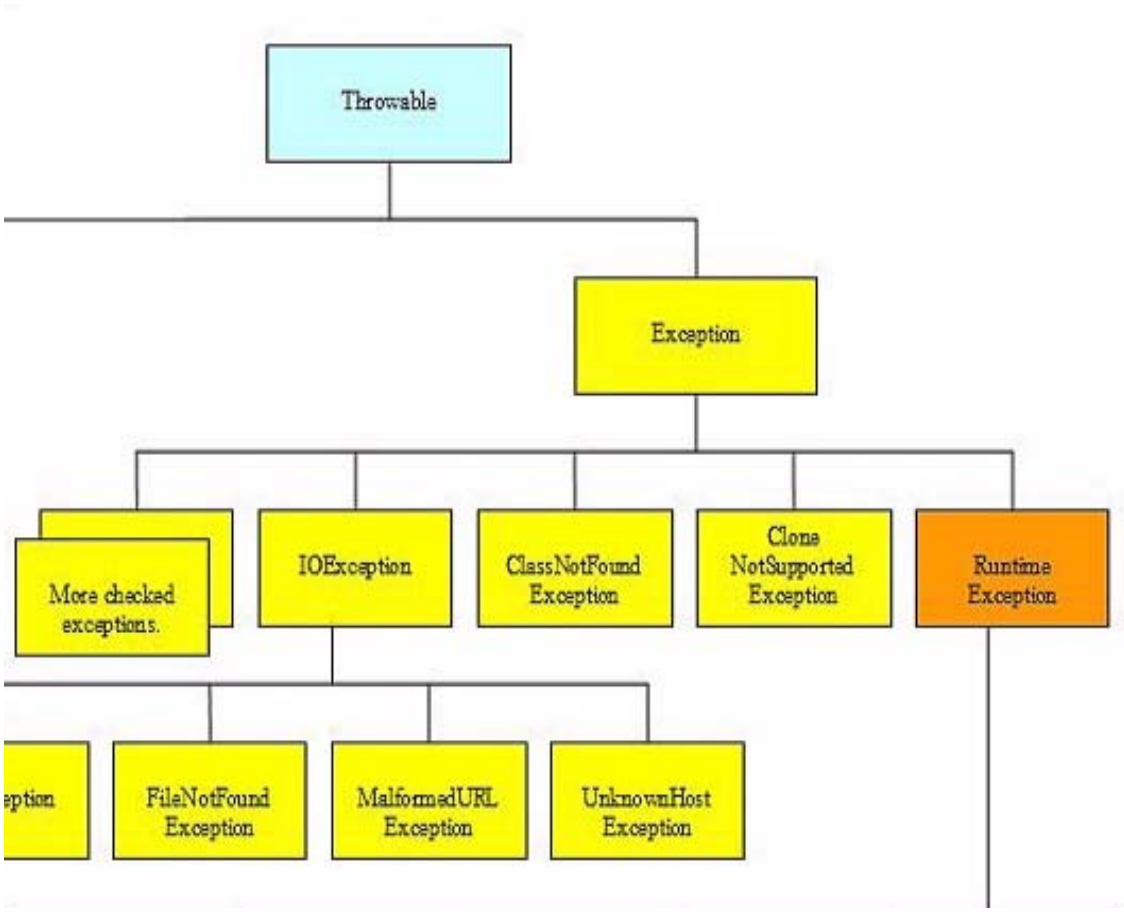
- a catch-block without implementation is allowed: however, the catch-block is suited to find an alternative solution for the problem encountered, and then continue the application in a controlled/desired way;
- a try-block must be followed by a catch-block or a finally-block (see 6.5.2.4 Keyword finally on page 191) - you are not allowed to write code between the try-block and the catch-block or the finally-block.
- a catch-block is allowed to throw the same exception again, or throw another exception. This can be useful if the problem can not be solved within the current method, and control is to be passed to the next level.

  In this case, don't forget to mention the (new) thrown exception in the **throws** clause of the method. (see 6.5.2.3 How to pass exceptions on page 189)

more than one catch-block possible

```
try {
    FileReader fr = new FileReader("file.txt");
    fr.read();
    } catch (FileNotFoundException fe) {
    fe.printStackTrace();
    } catch (IOException ioe) {
    System.out.println(ioe.getMessage());
    }
```

Figure 9: Multi catch exception



Combining multiple exceptions

```
catch (IOException | SalaryCalculationException exc) {
    exc.printStackTrace();
}
```

Additional possibilities for catching checked exceptions:

- several exceptions can be caught using **consecutive catch-blocks**:

  When writing several catch-blocks, make sure that you start with the more specific ones **widening** them into the more general ones, according to the inheritance tree (see picture).
  If you do not respect this order, the compiler will complain that it cannot reach certain catch-blocks;

  An example is shown on the opposite page.

- it is also possible to create so called multi-catch blocks, which make it possible to optimise or reuse exception handling code.

resource must implement AutoCloseable interface

try with resource

```
    try (LineNumberReader in =
            new LineNumberReader(new FileReader("in.txt"));
         PrintWriter out = new PrintWriter(new FileWriter("out.txt")) )
{
      ..... //  file handling code
}
    catch(IOException ioexc) {
        System.out.println(ioexc.getMessage());
    }
```

no finally block necessary for closing the resource(s)

Automatic Resource Management (ARM)

Another nice feature in the exception handling, is the "**Automatic Resource Management**" (ARM) in the try-instruction. This feature is (mostly) related to I/O handling.

The basic idea is to control the scope of a resource to a block, where it is used, and automatically close the resource when control exits the block.

In order to accomplish this automatic closing, the resource (file, stream, ...) must implement the *AutoCloseable* interface (superinterface of *Closeable*). Then the resource(s) is/are specified in the try clause within parenthesis, and the compiler detects automatically the scope.

More examples of ARM can be found in Chapter 9. Input/Output on page 231.

passing the exception to the caller:

```
public class Company {
    Person[ ] employees = new Person[10];

    public void paySalaries() throws SalaryCalculationException {
        for (int i = 0; i < 10; i++) {
            employees[i].calculateSalary();
        }
    }
    public void addEmployee(Person p) { /* ... */ }
}
```

Test Program:

```
    public static void main(String[ ] args) {
        try {
            Company comp = new Company();
            /* add employees to the company with method addEmployee */
            comp.paySalaries();
        }
        catch(SalaryCalculationException sce) {
            System.out.println(sce.getMessage());
        }
    }
```


6.5.2.3 How to pass exceptions

Sometimes when an exception is thrown to you, it is possible that you do not can/want to handle the exception. This is the case if you judge that it is the responsibility of the code calling the method you are writing is responsible for solving the problem. In this case you can **pass the exception to the caller**.

To pass an exception, you **just declare** it in the definition of the method. Use the keyword ***throws*** in the header of the method, to specify that a method can generate the declared exception. Everyone who wants to use this method is forced to handle the exception, i.e. it should be incorporated in a try-block.

So, although the exception is not generated in the method-block itself, it throws an exception, which was passed to this method, to its own caller.

This mechanism is illustrated in the example on the opposite page.

example:

```
public static void main(String[ ] args) {
    try {
        Company comp = new Company();
        /* add employees to the company with method addEmployee */
        comp.paySalaries();
    }
    catch(SalaryCalculationException sce) {
        System.out.println(sce.getMessage());
    }
    finally {
        System.out.println("this code block is always executed");
    }
}
```


6.5.2.4 Keyword *finally*

Code in a finally-block is always executed whether or not an exception has occurred. The ***finally*** keyword can be combined in two ways:

- with try and catch:

```
class X {
  void doingWhatever() {
    try {
      //...
    }
    catch (Exception e){}
    finally {
      // e.g. obj = null;  to enable for garbage collection
    }
  }
}
```

- with try and finally.

```
class Y {
  void doingWhatever() {
    try {
      //...
    }
    finally {
      //...
    }
  }
}
```

A finally-block can be **useful to perform necessary actions at the end of a method block**, e.g. to close database connections, or making objects eligible for garbage collection (i.e. object = null;).

Closing input- and/or outputstreams can now be established via the automatic resource management feature.

CHAPTER 7. COLLECTIONS

Collections are special-purpose objects that in fact represent **groups of objects** (hence, a collection can contain - among other things - another collection).

Collections can be ordered or not, can contain duplicates or not, can be defined as immutable or not, fixed or variable size, ...

The *java.util* package offers some interesting and indispensable classes that deal with collections.

Important note: Java 8

With the advent of Java 8, a lot of improvements have been incorporated in the collections framework, to take advantage of lambda expressions and streams. These concepts and possibilities are described in Chapter 10. Functional Programming on page 251.

You can find most of these new and enhanced classes in the following packages:

- *java.util*: integrates the Java Collections Framework with streams and provides general utility functionality used by streams.
- *java.util.function*: contains general purpose functional interfaces that provide target types for lambda expressions and method references.
- *java.util.stream*: contains the majority of interfaces and classes that provide functionality to streams and aggregate operations

Arrays vs containers

arrays

- **most efficient for random access**
- **but the size is fixed**
- **object type is known**

containers

- **contain object handles (no primitives)**
- **generics used to specify the type of elements**

***Collection* interfaces**

Set	**no particular sequence - no duplicates**
List	**particular sequence - duplicates allowed**
Queue	**to hold elements prior to processing**

***Map*: a group of key-value object pairs**

7.1 Arrays versus containers

7.1.1 Arrays

An array is the most efficient way that Java provides to store **a randomly accessible sequence of object references**. The array is a simple linear sequence, which makes random element access fast, but you pay for this speed: when you create an array object, the **size** is fixed and **cannot be changed** during its lifetime. If you run out of space, you need to create a new array and move all the references from the old array to the new array.

Or you can make use of (dynamic) collections.

7.1.2 Containers

Containers are objects which can store other objects (no primitives). They are an **important part of the *Collections Framework***. There are four major types of containers: *Set*, *List*, *Queue* and *Map*.

All of them make extensive use of the **generics facilities** (see 7.2.2 Using generics on page 201) to create instances that handle specific types of elements. This avoids the usage of the appropriate casts, which easily results in a lot of mistakes.

Any concrete container type in the *Collections Framework* may or may not permit the presence of null elements.

7.1.2.1 Collection

A *Collection* is a **group of individual elements**, with some rule applied to them:

- a ***Set*** holds elements in an **arbitrary order** and **no duplicates** are allowed;
- a ***List*** holds elements in a **particular sequence** and can contain **duplicates**;
- a ***Queue*** is a collection used to hold **multiple elements prior to processing** (FIFO - First In First Out, LIFO - Last In First Out, or any arbitrary order).

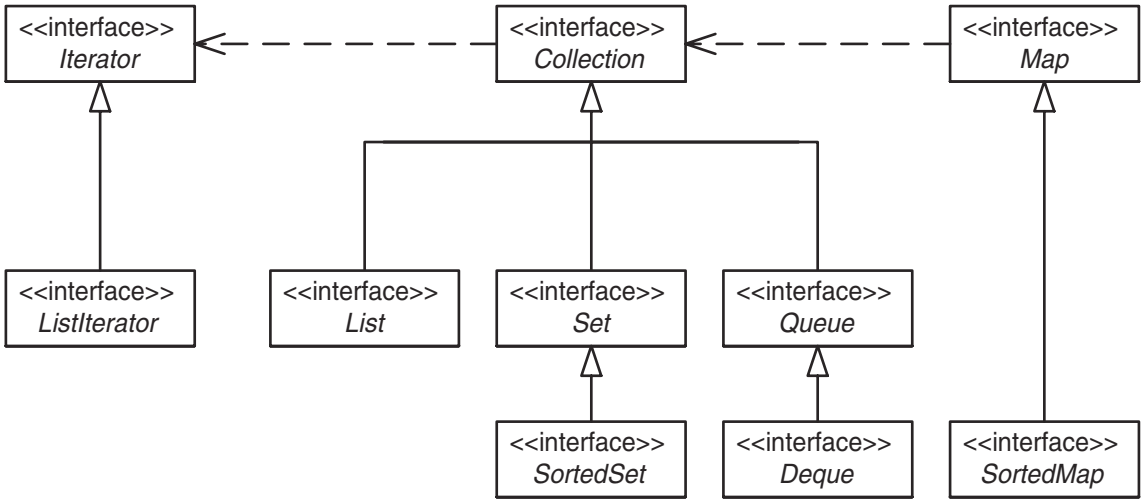
7.1.2.2 Map

A ***Map*** is a **group of key-value object pairs**.

At first glance, this might seem like a *Collection* of pairs where the keys are unique. A *Map* can return a *Set* of its keys, a collection of its values, or a *Set* of its pairs.

The Collections Framework (I)

Figure 10: The Collections Framework



a *Collection* can return an *Iterator*

Recommended use (with generics):

```
Interface<Type> ref = new ClassImpl<Type>();
```

Examples:

```
Set<Person> employees = new HashSet<Person>();
List<Course> courseList = new ArrayList<Course>();
```

7.2 The Collections Framework

The interfaces (see: 5.5.2 Interfaces and methods on page 151) that are concerned with holding objects are *Collection*, *Set*, *List, Queue* and *Map*.

Ideally you'll write most of your code to talk to these interfaces and the only place where you'll specify the precise type is at the point of declaration.

7.2.1 Main container interfaces

7.2.1.1 Set

In a *Set*, each element that you add must be unique; otherwise the Set doesn't add the duplicate element. Objects added to a set must define the *equals()* method to establish object **uniqueness**.

A Set in general provides **no particular order** but leaves that up to the specific implementations.

- A ***HashSet*** is fast in getting the elements or checking the existence of a particular element. It is implemented using a **hash table**. The order in which elements are returned by an iterator is not controllable by the programmer.
- A ***LinkedHashSet***, again implemented as a hash table, orders its elements based on the order in which they were inserted into the set.
- A ***TreeSet*** has the additional property that its iterator will traverse the set in a **particular sequence**. This sequence is determined by a natural order, given by implementing the *Comparable* interface for the elements added to the set, or a specific order, given by passing an object of a class that implements *Comparator* to the constructor of the *TreeSet*.
- An ***EnumSet*** is a specialized *Set* which allows **only** the **elements of one specific enum type**. This class is designed to be very efficient in time and space requirement, which makes it a typesafe alternative for traditional "bit flags".

7.2.1.2 List

The most important feature of a ***List*** is **order**. It promises to maintain elements in a **particular sequence**, laid down by the programmer. It allows **duplicates**. A *List* will produce a *ListIterator*, using this you can traverse the *List* in both directions as well as insert and remove elements in the middle of the *List*.

- An ***ArrayList*** is implemented as a **resizable array**: allows rapid random access to elements but is slow when inserting and removing elements from the middle of the list.
- A ***Vector*** is very similar to an *ArrayList*. The main differences between a *Vector* and an *ArrayList* are that a *Vector* contains **supplementary methods** besides those of the *List* interface for backward compatibility, and

  A *Vector* is **synchronized** (see 13.2 Concurrency on page 299) whereas an *ArrayList* is not.
- A ***LinkedList*** is implemented as a **doubly-linked list**: each node of the list has two links, one that points to the previous node and one to the next node. This explains why it provides **optimal sequential access**, with inexpensive insertions and deletions from the middle of the list. Relatively slow for random access.

Table 11: Container classes in the Collections Framework

		Implementation				
		Hash table[1]	Resizable array	Balanced tree[2]	Linked	Other
Interface	Set	HashSet		TreeSet	Linked HashSet	EnumSet
	List		ArrayList Vector		LinkedList	
	Map	HashMap		TreeMap	Linked HashMap	EnumMap
	Queue				LinkedList	Priority Queue

1. Stored objects need to implement the *hashcode()* and *equals()* methods.
2. Stored objects need to implement the *Comparable* interface or a *Comparator* object needs to be passed in the constructor (see 7.4 Sorting collections on page 215).

7.2.1.3 Queue

A queue is used to **hold elements temporarily prior to processing** and deliver them in a specific order to the processing object. Most queues supply their content in a *FIFO* or *LIFO* (stacks) ordering, but any other ordering is possible, e.g. priority queues. Also, queues may force capacity restrictions.

The **Queue** interface is implemented in the following classes:

- A ***LinkedList*** implements this interface as well as the *List* interface.
- A ***PriorityQueue*** offers its elements according to their natural ordering, or specified by a *Comparator*. The implementation provides logarithmic performance for the enqueuing and dequeuing methods (offer, poll, remove and add).

Deque, or double-ended-queue, is a subinterface of Queue. Deque supports supports element insertion and removal at both ends.

Other *Queue* implementations are part of the *Concurrency API* and hence out of this chapter's scope.

7.2.1.4 Map

Maintains **key-value associations** (**pairs**), so you can look up a value using a key.

The **Map** interface is implemented in the following classes:

- A ***HashMap*** provides **constant-time performance** for inserting and locating pairs. Performance can be adjusted via constructors that allow you to set the capacity and load factor of the **hash table**.
- A ***TreeMap*** is holding the keys (with the associated value) in a **sorted order**. This sorting sequence is provided in the same way as with a *TreeSet* using the *Comparable* or *Comparator* interface.
- An ***EnumMap*** is a special-purpose map that allows **only keys of a specific enum type**. Operations execute in constant time and, to cite the specifications: "They are likely (though not guaranteed) to be faster than their *HashMap* counterparts."

Java collections implement

interface *Collection*<E>

E is a type variable:
it represents the type of the collection's elements

validation of element type at compile time

example:

```
List<Course> courseList = new LinkedList<Course>();
```

or, using 'type inference'

```
List<Course> courseList = new LinkedList<>();

courseList.add(new Course());
Course firstCourse = courseList.getFirst();
```


7.2.2 Using generics

The idea behind the **generics** mechanism is to parameterize classes such that they can be instantiated with **class type parameters** at the time they are used.

The container classes of the *Java Collections Framework* are all defined as such generic classes, this means:

- when you declare or create an instance of a container you can specify the type of its elements;
- when storing, retrieving or otherwise manipulating elements of that container the compiler will check whether the type of these elements is the type that you declared (or a subclass as a matter of fact).

An example is shown on the opposite page.

If you make a *LinkedList* of *Course* elements, you can only insert elements that are instances of class *Course* and its subclasses. An *Iterator* over the *LinkedList* will return all elements as references to *Course* instances.

Prior to the introduction of generics, all elements were considered as subclasses of *Object* and the programmer was responsible for inserting and using the right instances and castings.

Writing your own generic classes is another issue and will not be dealt with in this chapter.

Some notes on generics:

- it is still possible to use these classes **without parameters**: this is equivalent to use *Object* as a parameter;
- **type inference** is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. As illustrated in the example, specification of generic type is not required in the construction of the collection;
- you cannot construct arrays of a generic type, though you can declare such an array and you can cast towards an array of a parameterized type;
- **type checking** happens **at compile time**, not at runtime: this ensures compatibility with classes that do not use generics - sadly, this also means that it is in certain circumstances possible to fool the compiler into accepting code that is guaranteed to cause exceptions;

safe way to iterate (and possibly modify) a collection

```
import java.util.*;
public class Test {
    public static void main(String[ ] args) {
        public List<Person> personList = new ArrayList<Person>();
        personList.add(new Person("Edison"));
        personList.add(new Person("Planck"));
        personList.add(new Person("Lotti"));
        personList.add(new Person("Einstein"));

        Iterator<Person> iter = personList.iterator();
        while (iter.hasNext()) {
            Person p = iter.next();
            if (“Lotti”.equals(p.getName())) {
                iter.remove();
            } else {
                System.out.print( p.getName() + " " );
            }
        }
    }
}
```

```
Output : Edison Planck Einstein
```

Subinterface ListIterator

navigation in both directions

+ `hasPrevious()` and `previous()`

Note: java.lang.Iterable interface enables “enhanced for” loop

```
for (Person p : personList)
  System.out.println(p.getName());
```

7.2.3 Iterator

Accessing multiple elements in a collection is possible by iterating over the collection. Two interfaces are available **to loop the different elements of a collection**: *Iterator,* and its subinterface *ListIterator*.

The interface ***Iterator*** has three methods:

- *hasNext()* tests if there are more elements available in the iterator;
- *next()* returns the next element in the iterator, provided that there are more elements in the iterator;
- *remove()* removes the previously returned element in a safe way from the collection from which the iterator was obtained.

  If this last method is not supported by an iterator, you will get an

  ```
  UnsupportedOperationException
  ```

The example on the opposite page shows how you can use an iterator to walk through a collection and remove elements from it.

ListIterator

defines additional methods to iterate through lists in either direction. For this purpose, the following methods ***hasPrevious()*** and ***previous()***, and, analogically, ***hasNext()*** and ***next()*** are declared.

Remarks:

- because of generics usage in the iterator, no casting is needed any more
- If you try to modify a collection in a non-safe way (without using an **Iterator**) you will have a great chance to trigger a

  ```
  ConcurrentModificationException
  ```

- all collections implement the *java.lang.Iterable* interface. This enables the usage in an **enhanced for loop** (see 2.7.6 enhanced for on page 45). When generics are used, no casting is necessary.
- *Enumeration*

  The interface *Enumeration* is available since Java 1.0 and is, strictly speaking, not part of the *Collections Framework*. It has been replaced with *Iterator*. *Enumeration* is still in use in some API's (e.g. Swing), but if it's not necessary, don't use *Enumeration*.

Set

no duplicates

```
import java.util.*;
public class Test {

    public static void main(String[ ] args) {

        String[ ] presidents = { "Roosevelt","Washington", "Eisenhower",
            "Johnson", "Truman", "Hoover", "Harrison", "Monroe",
            "Jackson", "Springsteen", "Jackson", "Washington" };

        Set<String> tree = new TreeSet<String>();
        Set<String> hash = new HashSet<String>();

        for (String s : presidents) {
            tree.add(s);
            hash.add(s);
        }
        System.out.println("Tree set:")
        for (String s : tree) {
            System.out.print( s + " " );
        }
        System.out.println("\n\nHash set:");
        for (String s : hash) {
            System.out.print( s + " " );
        }
    }
}
```

```
Tree set:
Eisenhower Harrison Hoover Jackson Johnson Monroe Roosevelt Springsteen
Truman Washington

Hash set:
Johnson Harrison Hoover Truman Springsteen Eisenhower Monroe Jackson
Roosevelt Washington
```

7.3 Examples of collection interfaces

Explaining all possibilities in the collection framework is not our goal. Hence we will illustrate some common usages of the most important subinterfaces of *Collection*.

7.3.1 Set

A ***Set*** is a collection that does not contain duplicate elements. The constructors are influenced by this restriction: if you provide a *Collection* that contains duplicates as arguments, these **duplicate elements will be stored only once**.

Common used methods are:

- *add(Element)* and *addAll(Collection)*: if the *Set* already contains one or more of the specified elements, the duplicate elements are not added to the set.

  Both methods return “true” if (at least) one element is added to the *Set*.

- *remove(Object)* and *removeAll(Collection)*: if the *Set* already contains one or more of the specified elements, they are removed.

  Both methods return “true” if (at least) one element is removed from the *Set*.

- *contains(Object)* returns true, if the set contains the indicated object.

Sets compare their elements using an object’s ***equals()*** method. Furthermore two sets are equal if the compared object is also a *Set* and if every element of their elements are the same, according to their respective *equals()* methods.

Implementations of the *Set* interface (extend the *AbstractSet* class):

- ***HashSet***: uses a kind of **hash table** to find and sort the elements, so, when iterating the elements, you can’t be sure of the order of the set. This ordering can even change over time. Furthermore, the null element is allowed.

- ***TreeSet***: uses a **balanced tree structure** to find and sort the elements, so, unlike the *HashSet*, this class guarantees a **fixed order** of the elements. It is therefore that it implements *Set*’s subinterface *SortedSet*.

  The order will be according to the natural order of the elements, given by the *Comparable* interface, or sorted by a *Comparator* which is passed to the constructor.

- ***LinkedHashSet***: works like a *HashSet* but **remembers the order in which the elements were added** to the *Set*. That way, the order in which the iterator traverses the elements is predictable.

- ***EnumSet***: can contain **only elements of a certain enum type**. It is designed to have a low memory cost and very fast operations while null values are not permitted. A typical use would be as an object-oriented (and typesafe) alternative to traditional “bit flags”. To create instances, instead of providing a public constructor, a number of static *factory methods* is available.

Note:

Whenever you decide to override the *equals()* method of an object, you should make sure that the object’s *hashCode()* method behaves consistently. If not, objects like *HashSet* cannot guarantee correct behaviour.

List

ordered - duplicates allowed

```
import java.util.*;
public class Test {
    public static void main(String[ ] args) {
        String[ ] presidents = { "Roosevelt","Washington", "Eisenhower",
            "Johnson", "Truman", "Hoover", "Harrison", "Monroe",
            "Jackson", "Springsteen", "Jackson", "Washington" };

            LinkedList<String> linkedList = new LinkedList<String>();
            ArrayList<String> arrayList = new ArrayList<String>(20);

            for (String s : presidents) {
                linkedList.add(s);
                arrayList.add(s);
            }
    printList("LinkedList: ", linkedList);printList("ArrayList: ", arrayList);

            linkedList.remove("Jackson");
            linkedList.removeFirst();
            arrayList.trimToSize();
            System.out.println("Nr 3: " + arrayList.get(2));

    printList("LinkedList: ", linkedList);printList("ArrayList: ", arrayList);
    }
}
```

```
length of LinkedList: 12
Roosevelt Washington Eisenhower Johnson Truman Hoover Harrison Monroe
Jackson Springsteen Jackson Washington

length of ArrayList: 12
Roosevelt Washington Eisenhower Johnson Truman Hoover Harrison Monroe
Jackson Springsteen Jackson Washington
Nr 3: Eisenhower

length of LinkedList: 10
Washington Eisenhower Johnson Truman Hoover Harrison Monroe Springsteen
Jackson Washington

length of ArrayList: 12
Roosevelt Washington Eisenhower Johnson Truman Hoover Harrison Monroe
Jackson Springsteen Jackson Washington
```


7.3.2 List

The added functionality of List resides in the fact that it is an **ordered Collection**. You have precise control over where in the list each element is inserted. Duplicates are allowed.

The ***List*** interface provides several (additional) methods in which the **(zero-based) integer index** is used:

- *add(int, Element)* and *addAll(int, Collection)* to insert the element(s) at the specified position
- *get(int) to retrieve an element at the specified position*
- *indexOf(Object)* and *lastIndexOf(Object)* to return the position of the element; -1 is returned if the list does not contain the object.
- *remove(int)* and r*emove(Object)* to remove an element (at the specified position). Subsequent elements are shifted to the left
- *set(int, Element)* replaces an element at the specified index by another instance of type E;
- *subList(int fromIndex, int toIndex)* returns a new *List* containing the elements between *fromIndex* (inclusive) and *toIndex* (exclusive).

Implementations of the *List* interface:

- **LinkedList**: all elements are possible in this type of collection (even null). Some additional methods are provided: you can get, remove and insert elements at the beginning and end of the list (as shown in the example).
- **ArrayList**: is a resizable-array implementation. Hence methods to manipulate the size, like e.g. *trimToSize()*

  An *ArrayList* has a **size** and a **capacity**. The size gives the number of elements in the *List*; the capacity indicates how much space is provided for the internal array. Another variable, **capacityIncrement**, indicates by how many units the capacity of the *ArrayList* is extended each time its capacity is reached. As a result, the capacity of an *ArrayList* is always larger than or equal to its size.

  Before inserting a large amount of objects in an *ArrayList*, it is good practice to increase the capacity (using *ensureCapacity(int)*). In this manner one avoids many incremental reallocations.

  ArrayList can be compared with *Vector* but *ArrayList* is unsynchronized and *Vector* is synchronized. A *Vector* contains supplementary methods besides those of the *List* interface for backward compatibility.

The example on the opposite page shows some method usage. The output is done via the following method:

```
static void printList(String listType, Collection<String> c) {
   System.out.println("length of " + listType + c.size());
   for (String s : c)
       System.out.print(s + " ");
   System.out.println("\n");
}
```

Queue

hold elements prior to processing

Method	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

```
import java.util.*;

public class Test {

    public static void main(String[ ] args) {
        String[ ] presidents = { "Roosevelt","Washington", "Eisenhower" };

        Queue<String> aQueue = new PriorityQueue<String>();

        for (String s : presidents) {
            aQueue.add(s);
        }
        while (aQueue.peek() != null)
            System.out.println( aQueue.poll() );
    }
}
```

```
Eisenhower
Roosevelt
Washington
```

7.3.3 Queue

Queues are collections, used for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations.

The methods of the **Queue** interface exists in two forms:

- one throws an exception if the operation fails
  - `add(E)` inserts an element E to the tail of the queue.

    throws an `IllegalStateException` if the element can not be inserted (because of queue's capacity restrictions).
  - `remove()` removes an element from the head of the queue.

    throws a `NoSuchElementException` if the queue is empty.
  - `element()` examines, but does not remove, the element from the head of the queue

    throws a `NoSuchElementException` if the queue is empty
- the other returns a special value if the operation fails (either null or false, depending on the operation).
  - `offer(e)` tries to insert an element e to the tail of the queue, but returns `false` if the element can not be inserted (because of queue's capacity restrictions)
  - `poll()` tries to remove an element from the head of the queue, and returns `null` if the queue is empty.
  - `peek()` examines, but does not remove, the element from the head of the queue and returns `null` if the queue is empty.

Queue implementations generally do not allow insertion of null elements.

- The **LinkedList** implementation is an exception, because of historical reasons
- **PriorityQueue** is used to sort a collection of elements according to their natural order (cf. Comparable), or according to a Comparator (see 7.4 Sorting collections on page 215).

A small example of some Queue methods is shown on the opposite page.

Note: Double ended queue

Deque (pronounced “deck”), is a subinterface of Queue supports element insertion and removal at both ends.

Interface Map

Map <K, V>

- **set of key-value mappings**
  order is left unspecified
- **set of (unique) keys**
- **collection of values**

subinterface: SortedMap<K, V>

7.3.4 Map

The ***Map<K, V>*** interface allows for the processing of three collections of objects:

- a set of unique **keys** (of type K),
- a collection of **values** (of type V) and
- finally a set of **key-value mappings**.

The interface does not specify anything about an explicit ordering of its keys or values. The subinterface *SortedMap<K, V>* and various implementations can be used for that purpose.

You should provide two constructors when creating a general-purpose map implementation: one default constructor and one so-called ***copy constructor*** with an argument of type *Map<? extends K, ? extends V>*.

Similar to the *Collection* interface, the implementation of some methods declared by *Map* is mandatory, while other methods are optional.

The **methods** of which the **implementation** is **mandatory** are:

- four methods returning a boolean value:
  - *containsKey(Object key)*: returns “true” if there is a value for this key;
  - *containsValue(Object value)*: returns “true” if there is a key for this value;
  - *equals(Object)*: compares an object to those of another *Map* for equality;
  - *isEmpty()*: returns *true* if no key-value pairs exist in the *Map;*
- three methods returning a *Collection*:
  - *entrySet():* returns a *Set<Map.Entry<K, V>>* of the mappings;
  - *keySet()*: returns a *Set<K>* of the keys;
  - *values()*: returns a *Collection<V>* of the values;
- other mandatory methods:
  - *size()*: returns the size of the map (the number of key-value pairs);
  - *hashCode()*: a hash code value for the map;
  - *get(Object key)*: returns the value corresponding to the specified key.

The **optional** method *put(K key, V value)* is used to add a key-value pair. Eventually an older mapping for the key will be replaced.

SortedMap is a subinterface of *Map*: it **guarantees an order** in which the keys are returned. The order will be according to the natural order of the keys, given by the *Comparable* interface, or the keys are sorted by a *Comparator,* which is passed to the constructor.

Map example

```
import java.util.*;
public class MapTest {
    public static void main(String args[]) {
        Map<String, Integer> scmmap = new HashMap<>();

        scmmap.put("CVS", 4);
        scmmap.put("SVN", 5);
        scmmap.put("GIT", 12);
        scmmap.put("ClearCase", 1);
        scmmap.put("SourceSafe", 2);
        System.out.println("Number of SCM systems: " + scmmap.size());

        for (String s : scmmap.keySet())
            System.out.println("key: " + s + " / value: " + scmmap.get(s));

        String searchKey = "GIT";
        if (scmmap.containsKey(searchKey))
            System.out.println("Found " + scmmap.get(searchKey) + " " +
                                    searchKey + " installations!");

        scmmap.clear();
        System.out.println("After clear operation, " +
                                "number of SCM systems: " + scmmap.size());
    }
}
```

```
Output :
Number of SCM systems: 5
key: CVS / value: 4
key: GIT / value: 12
key: ClearCase / value: 1
key: SourceSafe / value: 2
key: SVN / value: 5

Found 12 GIT installations!
After clear operation, Number of SCM systems: 0
```

using TreeMap instead of HashMap enables sorting!

The *Map* interface is implemented in the following classes:

- **HashMap** has the same functionality as the *Hashtable*, but *HashMap* is unsynchronized. Null values are allowed, as well as a null key.

  WeakHashMap is a *Map* with weak keys, i.e. an entry will automatically be removed when its key is no longer in ordinary use.

- **TreeMap** is an implementation of the subinterface *SortedMap*, and provides an ordered collection.

- **LinkedHashMap** works like a *HashMap* but remembers the order in which the elements were added to the *Map*. That way, the order in which the iterator traverses the key elements is predictable.

- By analogy with the *EnumSet*, an **EnumMap** implementation is available as well.

Some examples are shown on the opposite page.

Sorting collections (I)

Natural ordering with Comparable

```
import java.util.*;
public class Person implements Comparable<Person> {
    private int pno;
    ...
    //* the natural ordering of Persons is ascending on the person number
    public int compareTo(Person p) {
        return (this.pno < p.pno ? -1 : (this.pno == p.pno ? 0 : 1));
    }
```

Alternative ordering with Comparator

```
    /** Inner class to provide an alternative ordering:
      * descending on the person number.
      */
    public static class DescendingOnPersonNumberComparator
                                        implements Comparator<Person> {
        public int compare(Person p1, Person p2) {
            return (p1.pno < p2.pno ? 1 : (p1.pno == p2.pno ? 0 : -1));
        }
    }

}
```

7.4 Sorting collections

For sorting collections of objects, Java provides comparison functionality through the interfaces *Comparable* and *Comparator*. Both use generics.

7.4.1 Comparable

The first way to compare two objects is through natural comparison (i.e. comparison based on a **natural ordering**), that is imparted to a class by implementing the *java.lang.**Comparable***<*T*> interface.

The method ***compareTo(T)*** takes another instance of the same type (or subtype) as an argument, and returns the following values:

Table 12: Return values for compareTo(Object o)

this		o	return value
	<		< 0
	==		0
	>		> 0

When you define the comparison function, it is the **programmer's responsibility** that both the methods *equals()* and *hashCode()* behave consistently.

Note: compareTo() for String

The *compareTo()* method for Strings compares two strings lexicographically, based on the Unicode value of each character in the strings.

7.4.2 Comparator

If you don't want to use the natural ordering, you can also code a **different comparison function** for the type. To do this, create a separate (possibly inner) class that implements the ***Comparator*** interface.

The method ***compare(T, T)*** is very similar to *compareTo()* from the *Comparable* interface. The method *compare()* takes two Objects as arguments, and produces the following results

Table 13: Return values for compare(Object o1, Object o2)

o1		o2	return value
	<		< 0
	==		0
	>		> 0

As you can see in the example on the opposite page, reverting the order of the collection elements is possible by using the "right" values to return from the comparison operation: in this case a positive value for "less than", and a negative value for "greater than".

Sorting collections (II)

```
import java.util.*;

class Test {
    public static void main(String[ ] args) {
        Person p1 = new Person(1);
        Person p2 = new Person(2);
        Person p3 = new Person(3);
        Person p4 = new Person(4);

        Set<Person> tsn = new TreeSet<Person>();
        Set<Person> tso = new TreeSet<Person>(new
            Person.DescendingOnPersonNumberComparator());

        tsn.add(p4); tsn.add(p1); tsn.add(p3); tsn.add(p2);
        tso.add(p4); tso.add(p1); tso.add(p3); tso.add(p2);

        print_set(tsn); print_set(tso);
    }

    public static void print_set(Set<Person> s) {
        for (Person p : s) {
            System.out.print(p.getPersonNumber() +" ");
        }
        System.out.println();
    }
}
```

```
output :
1 2 3 4
4 3 2 1
```

As the example shows, you can indicate to your *Set* (or *Map)* that you wish to use an ordering which is different from the natural ordering. This is done by passing a *Comparator* object to the constructor of your *TreeSet* (or *TreeMap)*. Using the *compare()* method, the *TreeSet* (or *TreeMap)* can then find out the order of the elements.

Some explanation about the example:

The business requirements for the *Person* class determine that the most natural ordering for *Person* objects is ascending according to their person number. We indicate this by implementing the *compareTo()* method of the *Comparable* interface.

Another possible ordering that comes out of the business requirements, is a descending ordering according to the person number. Since the ***Comparable*** interface only allows the implementation of the ***most natural ordering*** but none other, implementing **other orderings** for the *Person* class should be implemented by using ***Comparator***. Although it is possible to create a separate class for this, the decision was taken to provide an inner class on the *Person* class. This way, it is very clear that the *DescendingOnPersonNumberComparator* class is tightly coupled to the *Person* class. It is now also possible to access, from this inner class, the private variables and methods of the *Person* class.

In the *Test* class, two *TreeSets* are created:

- `tsn` is a *TreeSet* which will use the natural ordering of the *Person* class to order its elements
- `tso` is a *TreeSet* which uses the other ordering, given by *Person.DescendingOnPersonNumberComparator*, to order its elements descending.

Four persons are added to each *TreeSet* and an iterator is used to traverse the set and print each element. We see that, as expected, the order in which the elements are accessed through the iterator is determined by the ordering used.

Collection utilities (I)

java.util.Arrays

manipulation of array objects

Examples:

```
int[ ] ints = {21, 13, 8, 5, 3, 2, 1, 1};
Person pp = new Person(“Paul”);
Person[ ] persons = { new Person(“Mary“), new Person(“John“), pp,
                                  new Person(“An“), new Person(“Peter“) };

// create a list
List<Person> plist = Arrays.asList(persons);

// sort elements
Arrays.sort(ints);
Arrays.sort(persons, new PersonComparator());

// search
int indexp = Arrays.binarySearch(persons, pp);

// convert to String
String string1 = Arrays.toString(ints);
String string2 = Arrays.toString(persons);
```

7.5 Utility classes for manipulation of collections

java.util contains two utility classes that will be helpful to manipulate array and collection objects.

7.5.1 Arrays

Class *Arrays* contains static methods to manipulate array object, for instance sorting and searching. Most methods come in different flavours. One for each of the primitive types and another one - some of them generic - for object types.

The class offers methods to:

- convert to a *List:* a static factory that allows arrays to be viewed as lists
- fill with an element
- sort its elements
- convert to a String
- search an element (in sorted arrays!)
- ...

Some examples can be found on the opposite page.

Note: System.arraycopy

```
public static void arraycopy(Object src, int srcPos, Object dest,
                                          int destPos, int length)
```

this static method copies an array from the specified source array (*src)*, beginning at the specified position, to the specified position of the destination array *(dest)*.

The number of elements to be copied is specified by the length argument.

Collection utilities (II)

java.util.Collections

operations on collections (Set, List, Queue, Map)

Examples

```
Person pp = new Person("Paul");
List<Person> plist = ....

// search a collection
int indexp = Collections.binarySearch(plist, pp,
                                      new PersonComparator<Person>());

// a synchronized view of your list
List<Person> splist = Collections.synchronizedList(plist);

// a dynamically typesafe view of the specified list
List<Person> cplist = Collections.checkedList(plist, Person.class);

// the immutable empty list
List<Person> elist = Collections.emptyList();

// copy 1 list to another, first arg is destination
Collections.copy(cplist, plist);

// shuffle the elements in a list
Collections.shuffle(plist);

// sort the elements in a list
Collections.sort(plist);
```

7.5.2 Collections

Class *Collections* contains only static methods to manipulate collections of objects.

The class offers methods to:

- create a synchronised collection;
- create an unmodifiable version of a collection;
- create a lightweight empty collection;
- sort a collection;
- search (for the minimum or maximum of) a collection;
- shuffle a collection
- ...

Some examples can be found on the opposite page.

CHAPTER 8. UTILITIES

In the *java.util* package we can find other utility classes like working with date/time or formatting classes. We will discuss the general use and give some examples of formatting.

Java 8 introduced a new date/time package *java.time* with even more (and better) features for handling date and time.

Lastly, number formatting will also be covered. The *java.text* package will provide us with the necessary utility classes.

Traditional way

- java.util.Date - many methods deprecated
- java.text.SimpleDateFormat

```
Date d1 = new Date();
SimpleDateFormat sdf1 = new SimpleDateFormat("dd/MM/yyyy");
SimpleDateFormat sdf2 =
                    new SimpleDateFormat("dd/MM/yyyy : kk:mm:ss");
Date d2;
try {
    d2 = sdf1.parse("26/04/1986");
    long diff = d1.getTime() - d2.getTime();
    long millisYear = 1000 * 24 * 60 * 60 * 365L;
    System.out.println("number of years: " + diff / millisYear)
    System.out.println(sdf2.format(d1));
} catch (ParseException e) {
    e.printStackTrace();
}
```

```
output : number of years: 25
         27/04/2011 : 15:48:14
```

- other classes:

  java.util.GregorianCalendar

8.1 Date / Time handling

8.1.1 Traditional way

Working with dates is one of the tricky things in the Java world. If you want a quick way to work with a universal time that is not affected by time zones, you can use the **java.util.Date** class, but most methods are now deprecated.

- An instance of the Date class represents a single date and time. Internally the date and time is stored as a primitive long. Specifically, the long value holds the number of milliseconds between the date being represented and January 1, 1970.
- Two constructors can be used to get a new Date instance.
  - The default constructor *Date()* gives back the current datetime, or
  - you can construct your own Date by putting in the number of milliseconds

    ```
    new Date(1000000000L) -> 12 jan 1970
    ```

    Not so handy, don't you think?
- The *getTime()* method returns this number of milliseconds, with which one can calculate time differences.
- Other (deprecated) methods of the Date class are *getYear()*, *getMonth()*, *getSeconds()*, ...

Note: JodaTime

- manipulating dates and times can be facilitated by using the open source package JodaTime, which is not part of the Java SE standard. For more details you can consult http://www.joda.org/joda-time/.

8.1.2 Date formatting

For parsing strings as dates or to print out dates in a readable manner, you can make use of the **java.text.SimpleDateFormat** class. The constructor takes in a certain format (check the API for all possibilities). Using the instance you can parse or format to/from Date. The `parse()` method imposes a checked exception called *ParseException*.

Some examples are shown on the opposite page.

Other, more sophisticated classes to work with date and time are e.g. **java.util.GregorianCalendar**.

Java 8

```
LocalDate localDate =  LocalDate.now();
LocalTime localTime = LocalTime.now();
LocalDateTime localDateTime = LocalDateTime.now();
DateTimeFormatter fmt = DateTimeFormatter.ofPattern("MM/dd/yyyy");

try {
    LocalDate someDate = LocalDate.parse("12/31/2025", fmt);
    LocalDate futureDate = LocalDate.of(2020, 12, 31);
    someDate = LocalDate.parse("2025-12-31"); // this works
    someDate = LocalDate.parse("31/12/2020"); // this doesn't
} catch (DateTimeParseException e){
    System.out.println(e.getMessage());
}

Period diff = Period.between(localDate, futureDate);
System.out.printf("Difference is %d years, %d months and %d days",
        diff.getYears(), diff.getMonths(), diff.getDays());
```

8.1.3 java.time package

Java 8 offers the **java.time** package to facilitate date and time handling.

The existing classes for handling date and time, as described on the previous pages, suffer from a number of problems:

- They are not thread-safe, leading to potential concurrency issues.
- They exhibit some counter-intuitive API design. For example, days start at 0, while months start at 1.
- Developers have to write additional code to handle timezone logic.

As from Java SE 8, the JDK core contains a new date and time API, which has been rewritten from the ground up. The most commonly used classes are **LocalDate**, **LocalTime** and **LocalDateTime**. They represent the local Date/Time from the context of the observer. These classes are mainly used when timezones are not required to be explicitly specified in the context.

To get an instance of the current date/time, based on the system clock, we can use the *now()* method. For specific dates and times, a factory method called *of()* is available.

Another option is parsing a string in a specific format, which can be created by using a **DateTimeFormatter** object. The parse method will also work without it, as long as we make sure we provide a date/time in ISO format (e.g. “yyyy-MM-dd”).

For calculating the difference between two dates, use the *between()* method of the **Period** class.

Number formatting

- java.text.Numberformat or java.text.DecimalFormat

```
double d = 1.2367;
DecimalFormat df = new DecimalFormat("#.00");
System.out.println( df.format(d) );
```

```
output : 1.24
```

8.2 Number formatting

Number formatting works in a similar fashion as date/time formatting.

Basic classes to use are **java.text.NumberFormat** and **java.text.DecimalFormat**.

Again, the format is constructed with a pattern in the constructor. The *format()* method is then applied to a number.

Possible patterns can be found in the API documentation of the DecimalFormat class.

Some examples are shown on the opposite page.

CHAPTER 9. INPUT/OUTPUT

In this chapter we take a closer look at the *java.io* package. This package offers classes and interfaces to control **input and output streams**, containing raw bytes or more structured data. We discuss the structure of the package.

Optimisation of input/output through buffers and channels, and support of (advanced) file and path addressing is available in the *java.nio* package.

Objects can be **serialized** so they can be sent through a stream. The opposite is possible as well (deserializing them and using them in a Java application).

Utility classes for file manipulation are available as well.

Important note: Java 8

The new package *java.util.stream* contains the majority of interfaces and classes that provide functionality to streams and aggregate operations. These classes allow for efficient file access through lambda (functional) programming, which is described in a separate Chapter 10. Functional Programming on page 251.

Input/Output - available classes

- **complex and obscure class structure!**
  compatibility reasons
- **based on decorator pattern**

java.io

- **streams used for input/output of:**
  - bytes
  - characters
  - data
  - objects
- **optimisation of I/O through buffering**
- **scanning and formatting streams**
- **command line I/O - Console**
- **file access**

java.nio

- **file and directory access**
- **file attributes**
- **file systems**

Automatic Resource Management (ARM)

resource must implement *AutoCloseable* interface

try with resources

9.1 Overview of java.io package - available classes

A first glance at the *java.io* package may leave you somewhat disoriented. The shear number of interfaces and classes together with the class hierarchy can make it difficult to see the wood for the trees.

Nevertheless, the apparent complexity of input/output Java is not a coincidental result of lousy library design. In fact, the structure of the java.io package reflects a sound design principle that enables you to customize i/o streams in a manner which is called the ***Decorator* design pattern**. Starting from a few core objects reflecting input and output streams, the **behaviour** of these **streams** can be **modulated** in a most flexible way depending on the requirements of your application.

Several categories of i/o classes are included in the *java.io* package:

- core classes for input/output of:
  - byte streams - raw binary data
  - character streams - character data with automatic conversion to/from local character set
  - data streams - handling binary I/O of primitive data types and Strings
  - object streams - (de)serialization to/from objects
- classes for opimisation of I/O through buffering
- scanning and formatting classes - break inputstream into a sequence of tokens
- command line I/O and *Console* class
- classes to handle file access.

The *java.nio.file* package defines additional interfaces and classes to access files, file attributes, and file systems.

In the following paragraphs these parts will be introduced and their combined use will be illustrated.

Note: Automatic Resource Management

The exception handling mechanism for I/O is improved with the introduction of so called **Automatic Resource Management** (ARM). The basic idea is to control the scope of a resource to a block, where it is used, and automatically close the resource when control exits the block. See Automatic Resource Management (ARM) on page 187.

In order to accomplish this automatic closing, the resource (file, stream, ...) must implement the `AutoCloseable` interface. Then the resource(s) is/are specified in the try clause within parenthesis, and the compiler detects automatically the scope.

The usage of automatic resource management is shown in the following examples.

access per byte -> int (0 - 255)

- **abstract superclass java.io.InputStream**
  - *read()* end-of-file = -1
  - *skip()*
  - *mark()*
  - *reset()*
  - *close()*
- **abstract superclass java.io.OutputStream**
  - *write()*
  - *flush()*
  - *close()*

Concrete implementations

- **FileInputStream / FileOutputStream**

  used for file I/O
- **BufferedInputStream / BufferedOutputStream**

  decorator for efficient buffering

9.2 Byte streams

All byte streams are subclasses of two common, abstract classes. One class that offers basic reading support, *java.io.InputStream*, and one that offers basic byte writing functionality, *java.io.OutputStream*.

9.2.1 InputStream

read() methods read bytes as **int in the range 0 to 255**; If the end of the stream is reached, the methods **return -1**.

A *skip(long)* method allows to skip over and discard a specified number of bytes of data from the stream.

The *available()* method returns the number of bytes that can be read. If there are **no bytes available, reading will block** execution until data become available.

When finished reading, the input stream should be closed. This can be done by calling *close*(), or using the **try with resources** clause (see example on next page).

The method *markSupported()* indicates whether a stream supports *mark(int)* and *reset()*. If supported, these methods can be used to reread a stream starting from a marked position.

9.2.2 OutputStream

***write*()** methods writes the specified byte(s) to the output stream.

flush() is used to force writing any buffered output bytes;

close() will close the output stream and release any system resources associated with the stream - a closed stream cannot be reopened. Again this can be accomplished through a try with resources clause.

9.2.3 Using byte streams

To show how byte streams work, we'll take a look at ***FileInputStream*** and ***FileOutputStream***, the **byte streams** used for file I/O. Construction is done, based on the file (object or name), e.g.

*FileInputStream(**File f**)* or *FileInputStream(**String fileName**)*

Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

Reading and writing is done with the methods provided in the abstract superclasses.

Using byte streams

```
public class ByteStreamDemo {
   public static void main(String[] args) {

    // automatic resource management
    try ( InputStream in =
              new BufferedInputStream(new FileInputStream("in.txt"));
        OutputStream out =
            new BufferedOutputStream(new FileOutputStream("out.txt"));)
        {
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } catch (IOException ioexc) {
            System.out.println(ioexc.getMessage());
        }
    }
}
```

9.2.4 Buffering

So far, we used unbuffered I/O. This means that each read- or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request may trigger disk access, network activity, or some other operation that is relatively expensive.

The use of **buffered I/O streams reduces** this kind of **overhead** significantly. Buffered input streams read data from a **memory area** known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

An unbuffered stream can be converted into a buffered stream by "**wrapping**" it (**decorator pattern**), by passing the former into the constructor of the latter, as shown in the example on the opposite page. Moreover, the resource creation is specified inside the try (ARM) clause, which makes them eligible for automatic closing.

The ***InputStream*** is constructed to read from the file in.txt and an ***OutputStream*** is created that will write to file out.txt.

Subsequently, the bytes from the input file are, one by one, read and written to the output file.

Both streams are closed to release the underlying file resources. This is done in a try with resources clause

The file access process can cause **exceptions** at execution time. The following checked exceptions should be caught:

- *FileNotFoundException* when the input file doesn't exist or can't be read

  FileNotFoundException is a subclass of *IOException*

- correspondingly, if the output file is a directory or can't be opened for writing a *FileNotFoundException* is thrown - but if the output file doesn't exist, an attempt to create a new file is made;

- the *read()*, *write()* and *close()* methods can cause an *IOException.*

Character streams

access per char -> 16-bit Unicode

- **abstract superclass java.io.Reader**
  - *same methods as InputStream*
  - *reset()*
- **abstract superclass java.io.Writer**
  - *same methods as OutputStream*

Concrete implementations

- **FileReader / FileWriter**

  used for file I/O
- **BufferedReader / BufferedWriter**

  decorator for efficient buffering
- **InputStreamReader / OutputStreamWriter**

  conversion byte stream to/from character stream

```
public class CharacterStreamDemo {
   public static void main(String[] args) throws IOException {
       try (Reader in = new BufferedReader(new FileReader("in.txt"));
           Writer out = new BufferedWriter(new FileWriter("out.txt")))
       {
           int c;
           while ((c = in.read()) != -1) {
               out.write(c);
           }
       } catch (IOException ioexc) {
           System.out.println(ioexc.getMessage());
       }
   }
}
```

9.3 Character streams

The previous example seemed like a decent program to copy the file in.txt, but in fact it is a kind of low-level I/O that you should avoid. Since it copies a text file, it should use character streams to achieve its goal. Java also offers streams for more complex data types, but, internally, all streams build on byte streams.

The Java platform uses **16-bit Unicode** characters. Character streams are used to automatically translate between raw bytes and Unicode characters. These streams will even automatically adapt to the local character set.

All character streams descend from two common, abstract ancestor classes *Reader* and *Writer*. Subclasses provide higher efficiency and/or additional functionality.

9.3.1 Reader

Most methods are equivalent to the InputStream, a character is read as int.

The method *ready()* indicates that a stream contains data to be read. Reading from an input stream that is **not ready** will **block** the execution of your program.

9.3.2 Writer

The abstract ***Writer*** class is the character counterpart of *OutputStream*, and uses the same methods.

Strings can be written as well, using:

void *write*(String str) or *void write*(String str, int off, int len)

9.3.3 Using character streams

Using character streams is almost identical to using byte streams. The only difference is that the read data is interpreted as 16-bit Unicode characters, which means that for each character read, the stream actually reads two bytes of data.

The example on the left demonstrates the use of character streams. This version of our program is almost identical to the one using byte streams. However, in this case `int c` will be used to hold **16-bit Unicode characters instead of raw 8-bit** values.

Note: InputStreamReader and OutputStreamWriter

Often character streams need to “wrap” byte streams (decorator pattern). In fact, e.g. the implementation of *FileReader* itself uses a *FileInputStream*. Two general-purpose streams specialize in this **conversion between characters and bytes**: ***InputStreamReader*** and ***OutputStreamWriter***. These classes are to be used in case there is only a byte stream available that meets your needs.

additional methods and features

```
public class CopyLines {
   public static void main(String[] args) throws IOException {
       try ( LineNumberReader  in =
                              new LineNumberReader(new FileReader("in.txt"));
             PrintWriter out = new PrintWriter(new FileWriter("out.txt")); )
         {
          String nextLine;
          int i;
          while ((nextLine = in.readLine()) != null) {
             i = in.getLineNumber();
             out.println( i + " " +  nextLine);
          }
       }
       catch(IOException ioexc) {
          System.out.println(ioexc.getMessage());
       }
    }
}
```

```
output:
1 first line
2 next line
3 last line
```

9.3.4 Line oriented input/output

Usually character data are not processed character by character. A unit of character data that is widely used, is of course a line. A couple of stream classes will help you use line-oriented I/O. An additional benefit is that these classes make an abstraction of the actual line end used on different operating systems.

All classes that help performing **line I/O** implement **buffering**, it would of course make no sense if they didn't:

- the ***BufferedReader*** class adds a *readLine()* method to our set of read methods - additionally, it also improves performance by implementing buffered reading;

  readLine() returns a String, or null if the end of the stream has been reached.

- the ***LineNumberReader*** class, which is a subclass of the former, adds a line count feature

  Often these reader features are sufficient, but if more complex processing is needed, *java.util.Scanner* will help you out;

- the ***PrintWriter*** class is used to output and format all kinds of data and allows, amongst others, to write lines ending with the local systems end-of-line characters through its *println()* method

  The default behaviour of a *PrintWriter* is to automatically *flush()* its contents after each line written.

Example:

On the opposite page is a listing of the program to copy a text file, implemented using these more sophisticated classes. The main difference is that, instead of reading the original byte by byte or one character at a time, this version processes the input file line by line.

In addition, the output in the example precedes each line with its line number.

"structured" data access

- **interface DataInput**
  - read binary data as primitive data
  - *readInt()*, *readBoolean()*, *readFloat()*, ...
  - end-of-file causes *EOFException*
- **interface DataOutput**
  - write primitive data as binary data
  - *writeInt()*, *writeBoolean()*, *writeFloat()*, ...

general-purpose implementations (decorators):

DataInputStream and DataOutputStream

```
// decorator of bytestream
DataInputStream in =
        new DataInputStream( new BufferedInputStream(
                                    new FileInputStream("input.bin") ) );
```

9.4 Data streams

To support binary I/O of primitive data types, Java provides several classes that can be used to read and write records that do **not** necessarily need to be **human readable**, for example to manipulate bitmap images.

9.4.1 Data stream interfaces

The interfaces ***DataInput*** and ***DataOutput*** define a range of read and write methods for all Java's primitive types, such as: *readBoolean(), readInt(), readFloat(),* ... and their *write* equivalents.

An important characteristic of these read methods is that they will throw an end-of-file exception (*EOFException*) to signal that the end of the data has been reached.

9.4.2 Data stream classes

The most common pair of streams that implement these interfaces are the classes ***DataInputStream*** and ***DataOutputStream*** that are **used as decorators** for other byte streams, typically constructed as shown on the opposite page.

Scanner

- tokenizing: break input into individual tokens
- constructed using files, streams or strings
- tokens automatically converted to primitive types

Example

```
/*  file in.txt contains whole numbers separated by  space hash space
        e.g.  4 - 9 - 3 - 8 - 5 - 12 - 7
*/
Scanner sc = new Scanner(new File("in.txt")).useDelimiter(" - ");
    boolean control = sc.hasNextInt();
    if (control == true)
        System.out.println("We found the following even numbers: ");
    while (sc.hasNextInt()) {
        int num = sc.nextInt();
        if (num % 2 == 0)    //  only even numbers will be printed
            System.out.print(num + "\t");
    }
```

```
output:
4        8        12
```

Formatting API

```
LineNumberReader  in =
                    new LineNumberReader(new FileReader("in.txt"));
PrintWriter out = new PrintWriter(new FileWriter("out.txt")); )
while ((l = in.readLine()) != null) {
        i = in.getLineNumber();
        out.format("% 4d\t%s\n", i, l);
     }
```

```
output:
   1        first line
   2        next line
   3        last line
```

9.4.3 Scanner and formatter

Programming I/O often involves translating to and from the neatly formatted data humans like to work with. To assist you with these chores, the Java platform provides

- the **Scanner** breaks input into individual tokens associated with bits of data.

  The example on the opposite page shows some possible methods to tokenize the input of a file, based on the delimiter " - ", and the type of token "integer".

  Another interesting scanner method is *findInLine(Pattern pattern)* which attempts to find the next occurrence of the specified pattern ignoring delimiters.

- The **formatting API** assembles data into nicely formatted, human-readable form. This formatting is available for PrintStream and PrintWriter classes.

  - *print()* and *println()* format individual values in a standard way.

  - *format()* formats (almost) any number of values based on a format string, with many options for precise formatting.

  An example of the *format()* is shown on the opposite page (based on the example explaining Using character streams on page 240).

  - The line number is written as four characters, written with leading spaces `% 4d`

  - followed by a tabulator character `\t`

  - and the original line `%s`

  - finishing with a system dependent line separator `\n`

The use of *scanner* and *formatter* will be discussed in more detail in the ABIS Java advanced course.

Object streams

lightweight ‘persistence’ through serialization

- **serializing = writing object to byte stream**
- **deserializing = reading object from byte stream**

- **implement `Serializable` (marker interface)**
- **referenced objects are**
  - **serializable**
  - **or marked *`transient`***

ObjectOutputStream

- ***writeObject(Object o)***

ObjectInputStream

- ***readObject()***

Note: compatibility issues between different JVMs

override SerialVersionUID

9.5 Object streams

Like data streams wrap a byte stream to read and write primitive data types, Java offers object streams that wrap a byte stream to read and write objects. Of course converting an object in a stream of bytes is a little more complex, this process is called ***serialization***.

There are a few requirements for an object to be serializable, otherwise a *NotSerializableException* is thrown:

- the object should **implement the marker interface *java.io.Serializable***
- serializing an object includes serializing all fields of the object (the whole object graph):
  - these fields should themselves be serializable, **or**
  - they should be explicitly excluded from the serialization process by adding the modifier ***transient*** to the field definition.

Note: static fields are never serialized.

9.5.1 ObjectInputStream and ObjectOutputStream

The object stream classes ***ObjectInputStream*** and ***ObjectOutputStream*** implement *ObjectInput* and *ObjectOutput*, which are subinterfaces of *DataInput* and DataOutput. That means that all the primitive data I/O methods covered in *Data Streams* are also implemented in object streams.

- To write an object to the output stream, use the *writeObject()* method. Even its private data members are written to the stream!
- To read an object, the stream offers *readObject()*. This method attempts to reconstruct the object and its attributes.

  A *ClassNotFoundException* is thrown when no compatible class exists in the classpath of the program that reads the object.

  The resulting object reference is of type *Object*, therefore it is necessary to cast the object to its actual type to use it.

Object serialization is used in communicating applications to exchange objects, for example the Java **Remote Method Interface** (RMI). This will be discussed in the ABIS Java Advanced course.

9.5.2 What is a compatible class?

When an object is ‘reconstructed’ from a serialised format, the actual available class in the JVM is used. If this class is a revised version, you can force the compatibility (at your own risk), by providing an additional attribute:

```
private static final long SerialVersionUID = 1L;
```

Keep the same ‘version’ number for compatible classes.

standard (byte) streams

- ***System.in***
  - **default: read from keyboard**
- ***System.out***
  - **default: output to display**
- ***System.err***
  - **default: output to display**

Console

- **character stream**
- **password support**

```
// obtain console
Console c = System.console();
if (c == null) {
    System.err.println("No console.");
    System.exit(1);
  }

  String login = c.readLine("Please, enter your login id: ");
    // password will not be echoed
  char [] oldPassword = c.readPassword("Enter your password: ");
```

9.6 Command line I/O

A program is often run from the command line and interacts with the user in the command line environment. The Java platform supports this through the standard (byte) streams.

Standard Streams are a feature of many operating systems. By **default**, they read **input** from the **keyboard** and write **output** to the **display**. They also support I/O on files and between programs, but that feature is controlled by the command line interpreter, not the program.

The Java platform supports three *Standard Streams*, all **byte streams**:

- *Standard Input*, accessed through ***System.in***;
- *Standard Output*, accessed through PrintStream ***System.out***;
- *Standard Error*, accessed through PrintStream***System.err***.

These objects are defined automatically and do not need to be opened.

Standard Output and *Standard Error* are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages.

> *System.out* and *System.err* are defined as *PrintStream* objects. Although it is technically a byte stream, *PrintStream* utilizes an internal character stream object to emulate many of the features of character streams.
>
> By contrast, *System.in* is a byte stream with no character stream features. To use *Standard Input* as a character stream, wrap *System.in* in *InputStreamReader*.

```
InputStreamReader cin = new InputStreamReader(System.in);
```

Console

This command line access, and more specific the Standard Input, can also be realised via the class **Console**:

- provides input and output streams that are true character streams, through its reader and writer methods.
- is particularly useful for secure password entry. Passwords are not echoed on the screen, and they are available as a character array instead of a String, (meaning that the char array can be overridden/emptied after usage).

In order to use the Console, the program must attempt to retrieve the Console object by invoking *System.console()*.

CHAPTER 10. FUNCTIONAL PROGRAMMING

In this chapter, we will discuss some important new Java 8 features, such as lambda expressions, which represent Java's first step into functional programming. Functional programming has gained importance lately because it is well suited for concurrent and event-driven ("reactive") programming.

We will also discuss functional interfaces as well as Java's new stream operations.

The need for lambda

- from (anonymous) inner classes to lambda expressions: reduce coding, make it more concise
- processing collections more effectively
- better support for multi-core processing

-> allow for functional programming

What is a lambda expression?

- a method without name, access-modifier or return value declaration
- represents (abstract method implementation of) functional interface
- target typing (+ type inference) provided by compiler

Examples:

```
(int x, int y) -> x + y
(x, y) -> x * y
(x, y) -> { return x+ y; }
s -> { System.out.println(s); }
() -> System.out.println("Hello")
```


10.1 Lambda: why, what, how?

10.1.1 Concepts

A lambda expression is a block of code that can be passed around as if it were a variable. It has parameters as well as a body, similar to traditional methods, but it is **anonymous**. Furthermore, it doesn't have an access modifier. It represents a (abstract method implementation of a) **functional interface**. Target typing is provided by the compiler.

Before Java 8, we needed to create an object and pass the object around. As of Java 8, we can treat functionality as a method argument.

Please note that lambdas only work with functional interfaces, which are interfaces with **only one abstract method**. Many functional interfaces are available in Java 8, like *ActionListener* and *Runnable*. More can be found in the package *java.util.function*. Furthermore, lambdas are used extensively in the *Stream* API, as we will see later on.

10.1.2 Syntax

Lambda syntax is pretty straightforward, as can be seen in the examples: a **comma-separated parameter list** (with optional data types) is followed by an **arrow token** (->), followed by the **body**, which consists of a single expression or a statement block. When there is just a single parameter, the brackets are optional. A return statement is also optional when the return type of the method is void.

Creating our own functional interface:

```
public interface PrintInfo {
    public void print();
}
PrintInfo prt = () -> System.out.println("Just a Lambda test print.");
prt.print();
```

Using Java 8 built-in interface:

```
List<String> welcomeList = Arrays.asList(“Hello”, “Hi”, “Welcome”);
Consumer<String> prt = s -> System.out.println(s);
welcomeList.forEach( prt ) ;

// alternative
welcomeList.forEach( s -> { System.out.println(s); } ) ;

// or
welcomeList.forEach( s -> System.out.println(s) ) ;
```


10.2 How to use lambdas

We can either create our own functional interfaces, or use the built-in ones.

Lambda supports “target typing”, which infers the object type from the context in which it is used. As a consequence, there are some restrictions when using lambdas:

- the type and number of the lambda expression **parameters** should be compatible with abstract method parameters.
- their **return types** must be compatible.
- any **exception** thrown by the lambda expression must be acceptable to the method.

Some examples of (existing) single function interfaces:

- `Runnable`

  void run()

  corresponding lambda: *() -> doSomething()*

- `Comparator`

  i*nt compare(x, y)*

  corresponding lambda: *(o1, o2) -> o1.function().compareTo(o2.function())*

- `ActionListener`

  void actionPerformed(ActionEvent e)

  corresponding lambda: *event -> doSomething()*

Method references

```
// Static method reference:
MyConverter< String , Double > conv = Double::valueOf;
Double target = conv.convert("-0025.300");
System.out.println( target );
```

```
// Object method reference:
class DemoStart {
  String startsWith(String str) { return String.valueOf(str.charAt(0)) ; }
}
DemoStart ds = new DemoStart() ;
MyConverter<String, String> convs = ds::startsWith ;
String targets = convs.convert("Hello"):
System.out.println( targets );
```

```
// Constructor reference:
// assume class Person with constructors:
// Person()
// Person(String firstName, String lastName)

@FunctionalInterface
interface PersonFactory<P extends Person> {
P create(String fname, String lname);
}

PersonFactory<Person> persFact = Person::new ;
Person p1 = persFact.create("James", "Bond");
System.out.println("Name : " + p1.getFirstName() + " " +
p1.getLastName();
```


10.3 Method references

We've used lambdas to create anonymous methods. Sometimes, however, a lambda expression does nothing more than call an existing method. In those cases, it's often clearer to refer to the existing method by name.

Hence, method references are compact, easy-to-read lambda expressions for methods that already have a name.

Because Java 7 and older did not have any syntax for methods themselves to be passed as an argument (you can only pass method results, but never method references), the **::** syntax was introduced in Java 8.

type	method reference	lambda equivalent
static	Class::method	obj -> Class.method(obj)
object	obj::method	() -> obj.method()
constructor	Class::new	() -> new Class()

```
public class Streams {

    public static void main(String[] args) {
        List<String> myList = Arrays.asList ("alex", "amy", "bob", "chet",
                                             "chat", "mike");
        myList.stream()
        .filter(s -> s.startsWith("c"))
        .filter(s -> !"chat".equals(s))
        .map(String::toUpperCase).sorted()
        .forEach(System.out::println);
    }
}
```

```
CHET
```

```
// stream operations can be used for file I/O, as well:
String fileName = "c://readme.txt";
        Stream<String> stream = Files.lines(Paths.get(fileName));
        stream.forEach(System.out::println);
        stream.close();
```

10.4 Streams

Streams allow us to perform tasks on collections of elements. The Java 8 Streams concept is based on converting *Collections* to a *Stream*, processing the elements in parallel and then gathering the resulting elements into a Collection.

Stream operations are either *intermediate* or *terminal*. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result.

In the example on the opposite page, we use some of the following stream operations:

Intermediate Stream operations: these perform specific tasks on the stream's elements, and always return a new stream. This allows for chaining multiple operations without using semicolons.

.*filter*: returns a stream consisting of the elements of this stream that satisfy the given conditions

.*map*: each element of the original stream is mapped to a new value (possibly of a different type)

.*distinct*: returns a stream with only the unique elements

.*limit*: only a specified number of elements are returned (from the beginning of the original stream)

.*sorted*: the elements are sorted in natural order unless you provide a custom Comparator

Terminal Stream operations: the stream's intermediate operations are processed, and a result is produced.

.*forEach*: performs processing on every element in a stream

.*count*: returns the numbers of elements in a stream

APPENDIX A. JAVA REFERENCE INFORMATION

A.1 Reserved words

The Java language is using a number of reserved words, so do not use them as identifiers.

- Keywords marked with (*) are reserved but currently unused by Java.

Words marked with (+) are not really keywords but literals, so you cannot use them as names in your programs.

abstract	assert	boolean	break	byte
case	catch	char	class	const (*)
continue	default	do	double	else
enum	extends	false (+)	final	finally
float	for	goto (*)	if	implements
import	instanceof	int	interface	long
native	new	null (+)	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true (+)	try
void	volatile	while		

A.2 Use of modifiers

Modifier	Used on	Meaning
visibility (access)		
none (package)	class interface member	accessible only in its package
private	class	for inner (nested) classes, accessible only in outer class, which contains it
	member	accessible only in its class, which defines it
protected	class	for inner (nested) classes, accessible only in outer class, which contains it
	member	accessible only within its package and its subclasses (even in another package)
public	class interface member	accessible anywhere
restrictions		
abstract	class	cannot be instantiated, because unimplemented methods must first be implemented in subclasses
	method	no body, only signature - the enclosing class must be abstract
final	class	cannot be subclassed
	method	cannot be overridden nor dynamically looked up
	field	cannot change its value (compile time constant)
static	class	makes an inner class top-level class
	method	a class method, invoked through the class name
	field	a class field, invoked through the class name
	initializer	runs when the class is loaded
synchronized	method	- for a static method, a lock for the class is acquired before executing the method - for a non-static method, a lock for the specific object instance is acquired
specials		
transient	field	field will not be serialized with the object
native	method	platform-dependent - no body, only signature
volatile	field	accessible by unsynchronized threads - rarely used

Possible combinations of modifiers

Modifier	Class	Field	Method	Constructor
private	no yes (for inner classes	yes	yes	yes
protected	no	yes	yes	yes
public	yes	yes	yes	yes
abstract	yes	no	yes	no
final	yes	yes	yes	no
static	no	yes	yes	no
synchronized	no	no	yes	no
transient	no	yes	no	no
native	no	no	yes	no
volatile	no	yes	no	no

Declarations:

- class

```
[public] [abstract] [final] class MyClass [extends MySuperClass]
                    [implements MyInterface1 [,MyInterface2]* ]
```

- member variables

```
[access level] [static] [final] [transient] [volatile] type myVar
```

- methods

```
[access level] [static] [abstract] [final] [native] [synchronized]
returnType myMethod([parameter list])
           [throws Exception1 [, Exception2]* ]
```

A.3 Class structure

A Java class definition contains a lot of sections which can be structured and used according to the following scheme.

Table 14: Java class structure

Description	Coding
class-location	`package` package-name`;`
class-import	`import` imported-class-location`;`
class-signature	(modifiers) `class` class-name `extends`/`implements`-clause
class-body	`{`
-> variables	(modifiers) type identifier (initialization)`;`
- primitive variable	(modifiers) primitive-data-type identifier (initialization)`;`
- reference variable	(modifiers) class-name identifier (initialization)`;`
-> constructors	(access modifier) class-name (type argument,...) {statements}
-> methods	(modifiers) type method-name (type argument,...) {statements}
- accessor methods	`public` return-type `get`...`()` `{return ...;}`
	`public void set`...(type argument) {var = argument;}
- business methods	(modifiers) return-type method-name (type argument, ...) {statements}
-> inner class	(modifiers) `class` class-name { variables, constructors + methods }
	}

An example (for the class Person) of this structuring:

```
package be.abis.companies;

import be.abis.products.*;

// class-signature
public class Person implements CourseParticipant
  // class-body begins here
  {
/* VARIABLES */
  // variables of primitive data type
  private int nr;                 // object variable
  private static int COUNT = 0;   // class variable
  // variables of object type
  private String firstName, lastName;
  private Address homeAddress;


/* CONSTRUCTOR(s) */
   public Person(String lName) { this.setLastName(lname);  }


/* METHODS */
  /* accessor methods */
   public String getLastName()  { return lastName; }
   public void setLastName(String newName)
      { lastName = newName; }


  /* business methods */
   public void attendCourse( ) { ... }
   // overloaded method
   public void attendCourse(Session curSession) { ... }


  // end of class-body
  }
```

APPENDIX B. EXERCISES

B.1 Java introduction and syntax

Create a new empty project called "**JavaProgramming**" in IntelliJ. In this, add a new java module "**Syntax**". Make sure that the language level is Java 8.

1. Add a class with a main method, called "Exercise1". In this, represent the use case stated below (we won't use "real" classes yet).

   We want to mimic the info linked with an instructor at ABIS. Each instructor has a **firstName**, an **age**, a **gross (monthly) salary**, and a **field indicating whether the instructor is a senior** in his job or not.

   Choose correct names and data types for these variables. Fill them in with some values, and use a *System.out.println* to print a sentence like:

   Bob is 43 years old. Gross salary is 3216.54. Senior: false

2. Calculate the net salary of the instructor, by deducting a tax of 48% from his gross salary.

3. Make the tax deduction dependent on the gross total income per year. You can find an overview of the rates at https://taxsummaries.pwc.com/belgium/individual/taxes-on-personal-income

4. Change the last part of the print statement (see exercise B1.1), such that it says:

   *Bob is **[not]** a senior.* Do this via the conditional operator.

5. Assume that the instructor started working in the company at 23 with a monthly gross salary of 2700.0, and that he got a raise of 5% each 2 years. Print out the salary history of the instructor in following format:

```
Bob's net salary at 23 is 1485.0000000000002
Bob's net salary at 25 is 1559.2500000000002
Bob's net salary at 27 is 1637.2125
Bob's net salary at 29 is 1719.0731250000001
Bob's net salary at 31 is 1805.0267812500003
Bob's net salary at 33 is 1895.2781203125003
Bob's net salary at 35 is 1809.1291148437504
Bob's net salary at 37 is 1899.585570585938
Bob's net salary at 39 is 1994.5648491152351
Bob's net salary at 41 is 2094.293091570997
Bob's net salary at 43 is 2199.007746149547
```

   Do this using a while loop. The current monthly salary should now be based on this calculation.

6. Modify the exercise such that the program takes 3 arguments (the start age, the current age and the gross start salary) from the command line. Calculate the salary as described in exercise one.
   You can use:
   `Integer.parseInt("aString")` to convert a String into an integer.
   `Double.parseDouble("aString")` to convert a String into a double.

7. Anyone who works for more than 35 years in the company, doesn't receive a raise of salary any more.

8. An instructor is linked to a number of domains. Use an array to add them, and print out the domains using the enhanced for loop. Also print out the number of domains.

9. Add a gender to the instructor (male, female or other).
   Use a switch statement to choose the correct pronouns for each case, and introduce them into the print statements you have used before. Make sure to capitalize the pronoun if necessary.

10. Add an enum *Gender* which represents the choices stated above. Adjust the switch statement to use this enum.

B.2 Java building blocks

Create a new Java module "**CourseAdmin**" in your project. Before implementing each exercise, draw a UML class diagram to represent the current situation, and show it to the instructor. All following exercises will build further onto this one.

1. Create a class **Course**. Each course object is described by **a title, a number of days and a price per day**.

   Add 2 methods, one called *printInfo()*, that prints out the course's data, and a second one called *calculateTotalPrice()*, which calculates the total price.

   In a Test class (with a main method), create an object of type Course, and call the 2 methods created above.

2. Make all variables private, and check what happens in the Test class. Add a constructor and accessor methods in the Course class.

   Use the constructor in the Test class.

3. Create a class **Company**. For the moment, a company is only represented by its **name**.

   Generate the constructor and accessor methods.
   Add a method *printInfo()* again, and create a company object in the Test class which calls it.

4. Create a class **Person.** Each person has a **first name**, a **last name**, and is possibly linked to a **company**.

   Add 2 overloaded constructors in the class, one without the company, and one with the company.

   Add a *printInfo()* method again, but make sure to check whether the person is linked to a company or not.

   In the Test class, create 2 persons by using the different constructors, and call the printInfo() method for both of them.
   The outcome should look like:
   Mary Jones works for Abis.
   John Doe is not linked to a company for the moment.

5. Define a static variable **counter** inside the **Person** class that keeps track of how many persons are instantiated, and another attribute **personNumber**, that keeps track of the (unique) number of the person object itself. Add the personNumber in the *printInfo()* method.

   Create an array of persons in the Test class. Print out the number of persons which were created, and loop through the array while calling the *printInfo()* method for each person.

6. Create an overloaded version of the *calculateTotalPrice()* method in the Course class, such that you can pass a reduction percentage as argument.

7. Add a method *attendCourse(Course c)* in the Person class. Implement it with a printout saying: *Bob is attending a Java course.*

8. Add a **list** (array) of **hobbies** (represented as string objects) to each person. Don't forget to create the accessor methods.

   Add a method *addHobby(String hobby)* in the Person class, and also foresee a method *addHobbies()* in case you want to add multiple hobbies at once. Use a varargs to do this.

9. Change the visibility of the *calculateTotalPrice()* method in the **Course** class to private, such that you can only call it from inside the class itself (e.g. from inside the *printInfo()* method).
   Check whether you can indeed not call it from the Test class anymore.

10. Split your application into 2 packages: **be.abis.courseadmin.model** and **be.abis.courseadmin.test**. Correct imports if necessary.

    Change the visibility of the *calculateTotalPrice()* method to package visibility. Call it from the Test class and from inside the *followCourse()* method in the Person class.

B.3 Java applications

1. Take a look at the Java SE API documentation, available

   - on the shared drive L:\Courses Library\JavaDoc\JDK1.8\docs, or

   - at https://docs.oracle.com/javase/8/docs/

   What does the *System.out.println()* statement actually mean? In which packages are the relevant objects, variables and methods defined?

2. Document your *Person* class. Make sure that you are documented as the author of the class. Generate the HTML documentation in your C:\temp directory, with the help of the Javadoc tool.

3. Create a jar file ("artifact") of your application, and run it via the command line.

B.4 Inheritance - polymorphism

1. A course is organised in **sessions**. ABIS provides two kinds of sessions:

   - **public sessions**: **organized by ABIS**;

   - **company sessions**: **organized by another company** - a company session has a **number of participants** (which is not always known when creating the session).

   However, what they have in common is that in each session a certain **course** is taught (use the provided *Course* class for this), has a **start date** (of type String for now), and a certain **location** (*Company*). Also the **instructor** (*Person*) is known.

   - provide for each class one or more **constructors** to set the variables

   - add a printInfo() method again, which prints out something like:

     The Java course will start on 24/10/2023. It will take place at ABIS. Your instructor is Bob Janssens of TTL.

     In case of a company session, add following sentence:

     This session is offered to you by IBM. There will be 5 participants.

     In case the number of participants is not known yet, don't add it to the print.

   - create at least 1 object of each session class in your Test class and print the information.

2. In the Test class, create an array of Session objects, in which you add the objects you created in the previous exercise. Use a for loop to do the printing. Can you see the polymorphism which is taking place?

3. Another service provided by ABIS, besides sessions, is **Consultancy**. One of the common methods in the classes **PublicSession**, **CompanySession** and **Consultancy** is *calculatePrice()*, but the implementation differs.

   Define a method *calculatePrice()* (with return type double) in an superclass **Service**, and see what happens in the subclasses. You do not have to provide a specific implementation in the *calculatePrice()* method, just return a hard-coded value.

   Create an array of services, fill it with the different types of services, and call the method *calculatePrice()* on each service object to see what happens. Let each object also print out of which class it is.
   Example output:
   The price for the CompanySession is 400.0.

4. In the case that the service is a session, also call the *printInfo()* method while looping through the array.

5. Add *toString()* methods in the classes Company, Course and Person. These could replace the *printInfo()* methods, or return a simplified output.
   Call them in the Test class by using a System.out.println of the given object, or integrate them into some of the other *printInfo()* methods.

6. Try to make the *calculatePrice()* method in the Service class protected. What happens?
   Try the same in the Session class. Explain the error message.

7. Experiment with the **final** keyword. Make the *calculatePrice()* method in the Consultancy class final. Create a subclass called **TopConsultancy**, and try to override the method. Also add a method *doExtras()* in the class.
   Try to make the Consultancy class final now. What happens?

8. Make the Service class and its *calculatePrice()* method **abstract** and check what happens.

9. Add an abstract method *getOrganizer()* in the Session class.

10. Add a **companyNumber** property in the Company class.

    Create a package **be.abis.courseadmin.repository**. In this, add a **CompanyRepository** interface with following methods:

    ```
    Company findCompany(int id);
    Company findCompany(String name);
    void addCompany(Company c);
    void updateCompany(Company c);
    void deleteCompany(int id);
    ```

    Then create a class **MemoryArrayCompanyRepository** which implements this. The class should contain a list (array) of 5 pre-created companies. Add those via the constructor. Implement the 2 “find” methods.

    Use the class in everywhere where you need a Company object. Use the “coding to interfaces” design principle.

11. Declare two interfaces **Instructor** and **CourseParticipant** that respectively declare the methods *teachCourse(Course c)* and *attendCourse(Course c)*.

    Modify the *Person* class to implement these interfaces (as roles a person can play).

    In the Test class, create a new person, and let him “switch hats”. One time he will play the role of instructor, another time he should play the role of course participant.

    Test objects of the modified class in your program.

12. Revisit the **Gender** enum we created in exercise B1.10. Put it in a package **be.abis.courseadmin.enum**.
    Add the different **pronouns** + an **abbreviation for the gender** as part of the constructor.

    Add a class **StringUtils** in a package **be.abis.courseadmin.util**. Define a static method *capitalize(String s)* in it to do the capitalization if needed.

    Use the gender in the Person class (make it required in the constructor), and rewrite the *printInfo()* method such that it uses some of the (capitalized) pronouns.

B.5 Exception handling

1. Implement the *calculatePrice()* method in **CompanySession**.
   This method calculates the price a company has to pay for a session as follows:

   total price = total persons * days * daily price.

2. When a company wants to order a session, they first check this total price. If the price is higher than a certain value, a **PriceTooHighException** should be thrown. Create the exception class in a package called **be.abis.courseadmin.exception**.

   Call the method in the Test class, and use a try-catch block to handle the exception. Try out different possibilities in the catch-block.

   Add a finally block and print a message to prove you have entered the clause.

3. Create a method *requestPriceOfferForCompanySession(Course c, int numberOfParticipants)* in the Company class. Let it call the *calculatePrice()* method. Handle the exception by declaring it in the method signature.

4. If the price for the offer is too low (according to the instructor's company), a **PriceTooLowException** should be thrown. Use different types of catch blocks in the main to test.

5. Create a superclass **PriceException** of previously created exceptions, and throw this from the "request" method.

6. Add a **CompanyNotFoundException** which is thrown from the "find" methods in the **MemoryArrayCourseRepository** class.

B.6 Collections

1. Change the hobbies array linked to person into a **HashSet** and adjust the "add" methods. Make sure to use the **Set** interface as the reference type each time. What happens if you introduce a same hobby twice?

2. Add a new implementation of the CompanyRepository which is called **MemoryListCompanyRepository**. This class uses an ArrayList (interface = List) to store the companies instead of an array. Make sure to implement all methods this time.

   Switch to this implementation each time you need company objects.

3. Now we are going to implement the enrolment process for public sessions. For a public session, each person attending the course is enrolled separately.

   In the class **PublicSession**, add a *List<CourseParticipant> enrolments* to achieve this.

   Implement following methods:

   - *addEnrolment()* to do 1 enrolment
   - *addEnrolments()* using a varargs in case you want to do multiple enrolments at once
   - *cancelEnrolment()* to remove a person from the list

4. Add a method *printParticipantList()* in the PublicSession class to print out the final participant list (call *printInfo()* for each person).

5. Create a method *findAbisParticipants()*, which returns a List of all participants of a given session (if any) which work for Abis.

6. Add another method removeAbisParticipants(), which removes all participants working for Abis from the list. Use an iterator to achieve this.

7. Switch the HashSet used in exercise B6.1 into a **TreeSet** an check what happens when you print out the hobbies of a Person.

8. Create an interface **CompanyService** in a package **be.abis.courseadmin.service.**
   Add a method *sortAllCompaniesByName()* in it.
   Implement the interface by using a class called **AbisCompanyService**. Link this class with the CompanyRepository.
   You will need to implement **Comparable** in the Company class to achieve this.

9. We also want to add a method *sortAllCompaniesByCompanyNumber()*.
   Use a static inner class in the **Company** class for the sorting. This class will need to implement the **Comparator** interface.

B.7 Utilities

1. Replace the String definition of a session's startDate with a **LocalDate** object. Use the **DateTimeFormatter** class in order to keep the format "correct" (e.g. 21/10/2022 or 5/1/2023) when you print or parse the date.

   Add a **DateUtils** class which is linked to your standard formatter.

2. Provide some formatting for the session's total price. Keep only 2 decimals, and use a "," as the decimal separator. Formatting can be done with the class **DecimalFormat**.

B.8 File I/O

1. Add a new implementation of the **CompanyRepository** interface, called **FileCompanyRepository**. In this case, the companies should be read in from a file called companies.txt (which you put under the c:\temp\javacourses directory).

   The file should only contain the company names. The companyNumber should be generated like you did for the persons in ExB2.5.

   Make sure to implement all methods again. Throw a **CompanyAlreadyExistsException** when adding a new company in case the name is already on the list.

   Don't forget to switch the implementation of the CompanyRepository everywhere where it is used.

   Delete the MemoryArrayCompanyRepository class.

B.9 Functional programming

1. Create a (functional) interface called **Calculator**, with a method *performCalculation(),* which takes in 2 arguments: one of type *double*, one of type *int*.

   First implement this in the Test class by using a lambda expression and try out.

   Also replace the implementation of the *calculateTotalPrice(int reduction)* method in the Course class, by using a lambda expression to do the price reduction.

2. Add a class **CalculatorUtils**, with a static method *double callCalculator(Calculator c, double d, int i)*.
   Replace the implementation you did in previous exercise by calling this method. Make sure to pass the lambda directly in the method call.

3. Add an **EnrolmentService** interface, with implementation class **AbisEnrolmentService**. Foresee a number of *sort()* methods in this to sort the List<Participant> of the PublicSession (on personNumber, lastName, firstName, combination of lastName+firstName,...). The sort on personNumber should be considered as the "natural" sorting. Implement the other Comparators by using lambdas instead of inner classes.

4. Print out the (sorted) partipant list of a PublicSession (person's names + company name will do for now) by using the *forEach()* method on the list. Try to use a method reference for this.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M