

Numerical Methods and Tools

- Introductory Java Course -

This document comes together with the course. Corresponding part has to be read before attending each session with Professors.

September 9, 2016



Contents

I	Basics: 8 hours + personal work	1
1	Introduction	3
1.1	Few words about Java	3
1.2	Limitations of Java	5
1.3	External references	5
2	The Netbeans IDE	7
2.1	What is Netbeans?	7
2.2	Creating a Java project with the Netbeans IDE	8
2.2.1	Setting Up the Project	8
2.2.2	Adding Code to the Generated Source File	12
2.2.3	Compiling and Running the Program	13
2.2.4	Building and Deploying the Application	14
2.3	Exporting and importing projects	15
2.3.1	Exporting HelloWorldApp as a zip file	15
2.3.2	Importing a project from a zip file	16
3	The Java Basics	17
3.1	Improved Hello World!	17
3.2	Java Naming Convention	19
3.3	Indenting code	20
3.4	Commenting Java code	20
3.5	Java keywords	21
3.6	Structuring Java codes into packages	22
4	Data types in Java	25
4.1	Primitive data types	26
4.2	Boolean expressions	28
4.3	Strings in Java	30
4.4	Arrays in Java	32
4.4.1	Static arrays	32
4.4.2	Dynamic arrays	34
4.4.3	Other collections of variables	37
4.5	New types: classes and objects	40

5	Processing data in Java	43
5.1	Conditional statements	43
5.1.1	If statement	43
5.1.2	Switch statement	45
5.2	Loop statements	47
5.2.1	While statement	47
5.2.2	Do-while statement	47
5.2.3	For statement	48
5.2.4	For-each statement	48
5.2.5	Complements	49
5.3	Methods rather than functions	51
5.4	Scope of variables	55
6	Problem: managing time series	59
6.1	Problem statement	59
6.2	Clues	60
II	Object Oriented Programming: 8 hours + personal work	61
7	Object oriented programming	63
7.1	Deeper into the object oriented paradigm	63
7.2	Polymorphism	67
7.3	Encapsulation	69
8	Core object oriented programming	73
8.1	Inheritance and overloading	73
8.2	Abstract classes and interfaces	82
8.3	Interfaces for structuring codes	86
8.4	Useful methods to manage arrays and collections	89
9	Exceptions and errors	91
9.1	First approach	91
9.2	Deeper into exceptions	93
9.3	Raising an Exception	95
10	Designing your own object models	97
10.1	Stick to reality	97
10.2	Using UML class diagrams as design tool	97
11	Problem: Matrices	101
11.1	Problem statement	101
11.2	Clues	102
12	Problem: Shapes	103
12.1	Problem statement	103

12.2 Clues	104
III Project: 12 hours + personal work	105
13 Programming Graphic User Interfaces	107
13.1 Philosophy of modern GUI toolkits	107
13.2 Java Swing	108
13.2.1 UI components	109
13.2.2 Layouts	121
13.2.3 Events	138
14 Plotting curves into Java	145
14.1 Introduction to JFreeChart	145
14.1.1 Collecting data	147
14.1.2 Creating chart	147
14.1.3 Displaying the chart	148
14.2 LineChartDemo2	148
14.3 TimeSeriesDemo8	151
15 Problem: occupancy estimators	155
15.1 Problem Statement	155
15.1.1 General objective	155
15.2 Clues	160
15.2.1 Reading a comma-separated values (CSV) from Java . . .	160
15.2.2 Possible architecture for the application	161
15.2.3 Dichotomic search algorithm	161
15.2.4 Simulated annealing algorithm	162

Part I

Basics: 8 hours + personal work

Chapter 1

Introduction

1.1 Few words about Java

Java is platform independent. This means that it will run on just about any operating system. So whether your computer runs Windows, Linux, Mac OS, it is all the same to Java! The reason it can run on any operating system is because of the Java Virtual Machine, which has been adapted to each operating system. Therefore, from a Java application point of view, all the machines looks similar: the virtual machine interprets the Java codes and transforms it to codes that can be understood by the physical machine with its operating system.

Java has the following characteristics:

- A Java source code (‘.java’) is precompiled into a Java byte code (‘.class’) that can be interpreted on a Java Virtual Machine
- Java syntax is inspired from C++
- Java uses a garbage collector, that clear automatically the no longer used variables, and avoid to forget the C memory dis-allocations.
- Java is object oriented i.e. variables and functions related to a same entity can be gathered into objects.
- Java Byte code is interpreted Just In Time. It is much more efficient than the static compilation because dynamic compilation optimized the compilation taking account the current context.
- The Java Virtual Machine yields the adage ‘Develop Once, Run Everywhere’.
- The Java programs are generally very stable because it is strongly typed (types of variables have to be explicitly defined) and because of the garbage collector.
- The Java language is widely used in Web development because a Web appli-

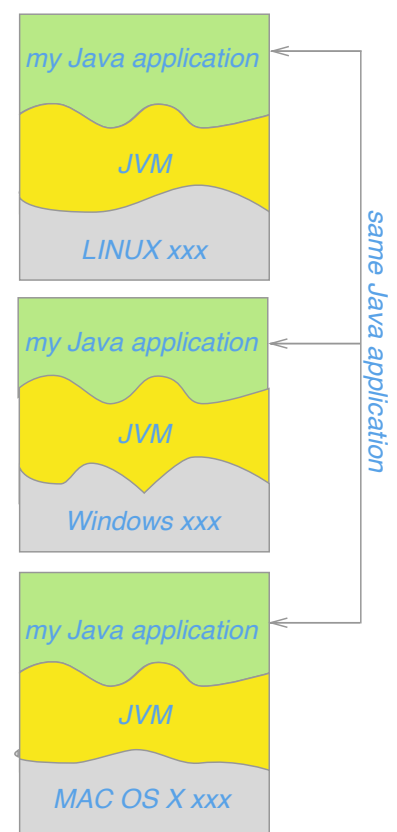


Figure 1.1 Role of the JVM

cation should be executable on any operating system.

- The Java language is very well documented. As a new Java developer, you should follow the way of your peers: refer when needed to the Java API documentation at <http://www.oracle.com/technetwork/java/api-141528.html>

Nowadays, Java is widely spread. Indeed, in 2002, Java was installed on 550 millions of computers and 75% of the developers was using Java as a first language. Today, 4.5 millions of developers work with Java.

The Virtual Machine is a program that processes all your code correctly. So you need to install this program (Virtual Machine) before you can run any Java code. Although the presence of a virtual machine, Java programs run fast.

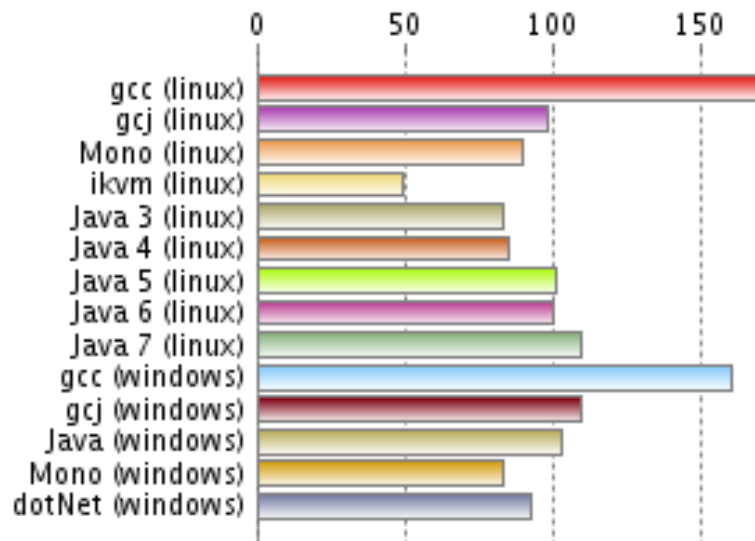


Figure 1.2 Comparison between C (GCC) and different versions of Java on different OS: score = $time_{ref\ java\ 6} / time$

Java is owned by a company called Oracle, so you need to head over to Oracle's website to get the Java Virtual Machine, also known as the Java Runtime Environment (JRE). Nevertheless, source code of Java itself is available i.e. Java is now open source. The official documentation is available at <http://java.com/en/download/manual.jsp>.

To write code and test it out, you need something called a Software Development kit. The one we are going to be using is called Java SE (for Standard Edition).

You write the actual code for your programs in a text editor or in an integrated development environment (IDE), such as Eclipse we are going to use. Basically, an IDE integrates a text editor, a debugger with many other facilities to support

the code development. The code is called source code, and is saved with the file extension '.java'. A program called 'javac' is then used to turn the source code into Java Byte Code. This is known as compiling. After 'javac' has finished compiling the Java Byte Code, it creates a new file with the extension .class (At least, it does if no errors are detected). Once the class file has been created, it can be run on the Java Virtual Machine. So, the different steps are:

- Create source code with the extension '.java'.
- Use Javac to create (compile) a file ending in .class
- Run the compiled class

Eclipse handles all the creation and compilation steps for you. Behind the scenes, though, it takes your sources code and creates the java file. It will launch Javac and compile the class file. Eclipse can then run your program inside its own software. This saves you the hassle of opening up a terminal window and typing long strings of commands.

1.2 Limitations of Java

Because of the virtual machine, Java program are not directly connected to the machine. Consequently, to develop drivers and operating systems but also real-time applications, code like C, C++ or Objective C are preferred.

Microsoft has developed an alternative to Java: the '.net' environment with the C# language but Java is more widespread, in particular, for Web applications.

Python language is considered as faster to learn and it supports different paradigms (structured programming like C and object oriented like Java). It is currently spreading in the scientific areas. It is less verbose than Java and more relevant for fast prototyping of solutions. It is less constraining than Java (no explicit type like integer, float, string... to be declared) but many companies consider this as a weakness: too much freedom may lead to codes of poor quality.

1.3 External references

To see how to install a JDK on Windows, watch <https://youtu.be/HI-zzrqQoSE?list=PLFE2CE09D83EE3E28>.

Mac OS X already contains a JDK. If you want to update the version, watch

Here are few markers of the Java history.

1990. Development of the Oak language at Sun Microsystems by James Gosling and Patrick Naughton. Oak is independent of the operating system (Platform-Independent). Oak has been used in home automation projects using infrareads (project *7) and for Video On Demand (project VOD)

1995. Oak becomes Java. Commercial launching. It has been used for Web applications (applet).

1996. java 1.0.1

1997. java 1.1

1998. Java 2 i.e. java SE 1.2. Launching of the Java Community Process (JCP). The JCP leans on the publications of Java Specification Requests that propose Java extensions. Propositions of JSR are presented and voted by the JCP Executive Committee (see <http://jcp.org>). A final JSR might then be published with a reference implementation.

1999. J2EE

2000. java 1.3

2002. java 1.4

2004. java 5.0

2006. java 6.0, Java turns into Gnu Public License

2009. Sun is bought by Oracle

2011. java 7.0

2014. java 8.0

Table 1.1 Few markers of the Java history

<https://youtu.be/qVulWqwfPy0>

A good reference for learning Java is the Java programming wikibook https://en.wikibooks.org/wiki/Java_Programming.

Books can also be found on the web such as ‘Java for dummies’ from Barry Burd (Wiley) or ‘Apprenez à programmer en Java’ from Cyrille Herby (site du Zéro).

An interesting tutorials about Java is, for instance, <http://www.homeandlearn.co.uk/java/java.html>. But other interesting tutorials can be found on the Web, depending on your current skills.

Documents are also available on Chamilo.

Chapter 2

The Netbeans IDE

2.1 What is Netbeans?

NetBeans is an integrated development environment written in Java. It allows applications to be developed from a set of modular software components called modules. The NetBeans IDE is primarily intended for development in Java, but also supports other languages, in particular PHP, C/C++ and HTML5. NetBeans is cross-platform and runs on Microsoft Windows, Mac OS X, Linux, Solaris and other platforms supporting a compatible JVM.

NetBeans began in 1996 as Xelfi, a Java IDE student project under the guidance of the Faculty of Mathematics and Physics at Charles University in Prague. In 1997, Roman Staněk formed a company around the project and produced commercial versions of the NetBeans IDE until it was bought by Sun Microsystems in 1999. Sun open-sourced the NetBeans IDE in June of the following year. Since then, the NetBeans community has continued to grow. In 2010, Sun (and thus NetBeans) was acquired by Oracle.

Among the interesting features of Netbeans, let's list:

- A debugger, let you place breakpoints in your source code, add field watches, step through your code, run into methods, take snapshots and monitor execution as it occurs. See for instance <https://www.youtube.com/watch?v=2Z9B8wYhKWw>.
- A profiler, provides expert assistance for optimizing application's speed and memory usage, and makes it easier to build reliable and scalable Java applications. See for instance <https://www.youtube.com/watch?v=DI4EFkzqCCg>.
- A Swing Graphic User Interface (GUI) builder, makes it possible to design

graphic user interfaces by dragging and positioning GUI components from a palette onto a canvas. The GUI builder automatically takes care of the correct spacing and alignment. See for instance <https://www.youtube.com/watch?v=8CI718A8OUI>

2.2 Creating a Java project with the Netbeans IDE

This tutorial ¹ provides a very simple and quick introduction to the NetBeans IDE workflow by walking you through the creation of a simple "Hello World" Java console application. Once you are done with this tutorial, you will have a general knowledge of how to create and run applications in the IDE.

2.2.1 Setting Up the Project

To create an IDE project:

- Start NetBeans IDE.
- In the IDE, choose File > New Project, as shown in the figure below.

¹available at <https://netbeans.org/kb/docs/java/quickstart.html>

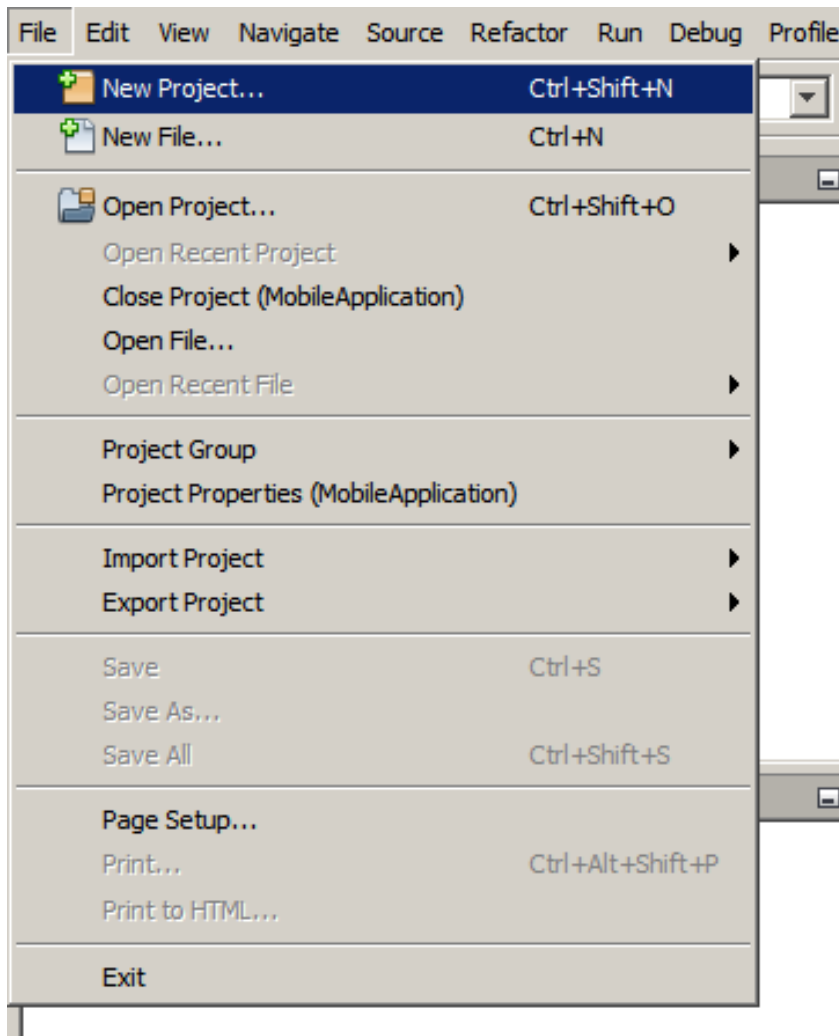


Figure 2.1

- In the New Project wizard, expand the Java category and select Java Application as shown in the figure below. Then click Next.

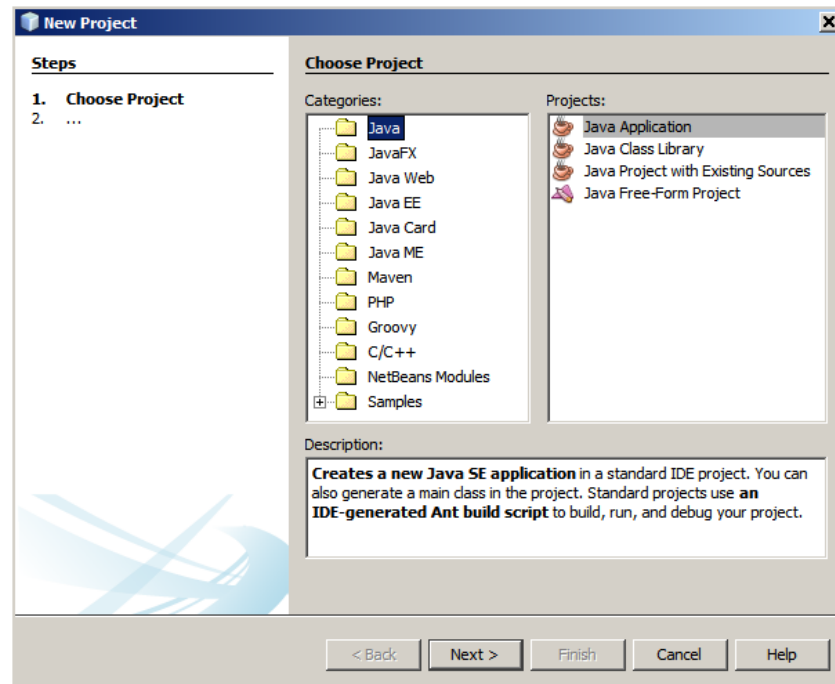


Figure 2.2

- In the Name and Location page of the wizard, do the following (as shown in the figure below):
 - In the Project Name field, type HelloWorldApp.
 - Leave the Use Dedicated Folder for Storing Libraries checkbox unselected.
 - In the Create Main Class field, type helloworldapp.HelloWorldApp.

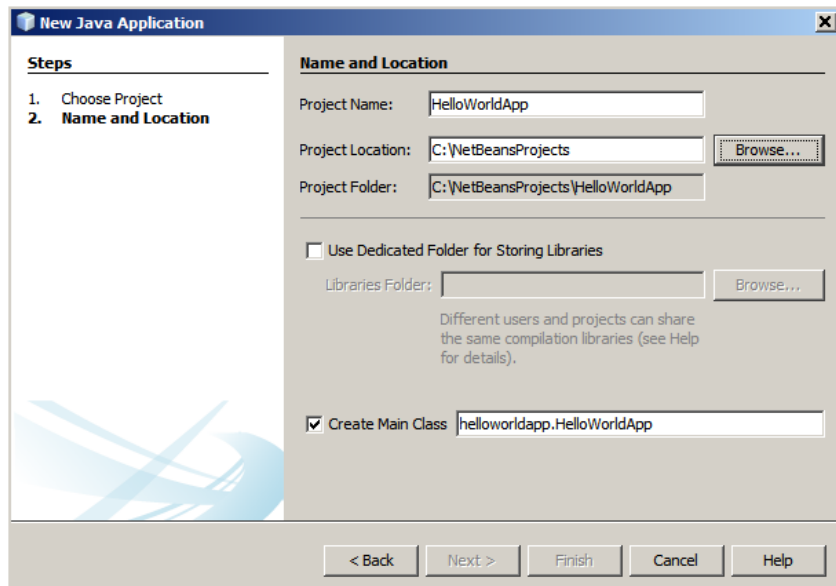


Figure 2.3

- Click Finish. The project is created and opened in the IDE. You should see the following components:
 - The Projects window, which contains a tree view of the components of the project, including source files, libraries that your code depends on, and so on.
 - The Source Editor window with a file called HelloWorldApp open.
 - The Navigator window, which you can use to quickly navigate between elements within the selected class.

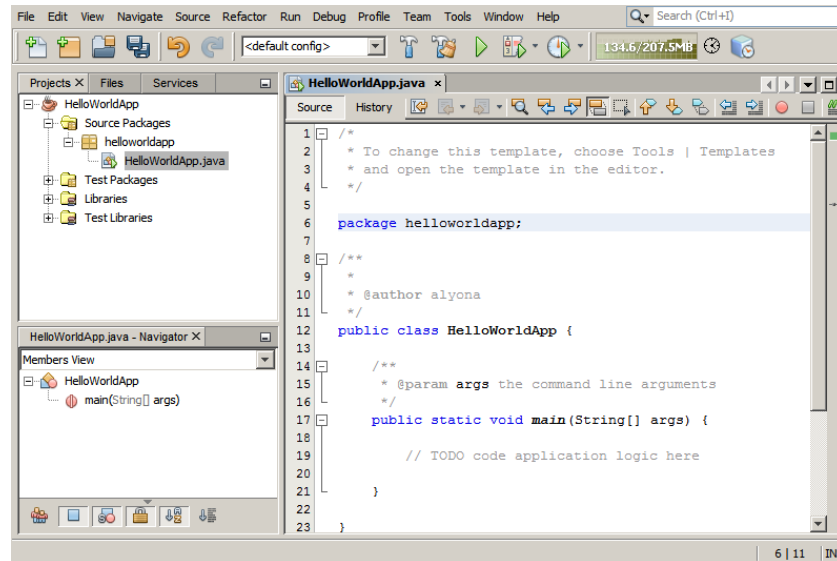


Figure 2.4

2.2.2 Adding Code to the Generated Source File

Because you have left the Create Main Class checkbox selected in the New Project wizard, the IDE has created a skeleton main class for you. You can add the "Hello World!" message to the skeleton code by replacing the line:

```
// TODO code application logic here
```

with the line:

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

Save the change by choosing File > Save. The file should look something like the following code sample.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package helloworldapp;

/**
 *
 * @author <your name>
 */
public class HelloWorldApp {
```

```
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    System.out.println("Hello World!");
}
}
```

2.2.3 Compiling and Running the Program

Because of the IDE's Compile on Save feature, you do not have to manually compile your project in order to run it in the IDE. When you save a Java source file, the IDE automatically compiles it.

Information

The Compile on Save feature can be turned off in the Project Properties window. Right-click your project, select Properties. In the Properties window, choose the Compiling tab. The Compile on Save checkbox is right at the top. Note that in the Project Properties window you can configure numerous settings for your project: project libraries, packaging, building, running, etc.

To run the program:

- Choose Run > Run Project.

The next figure shows what you should now see.

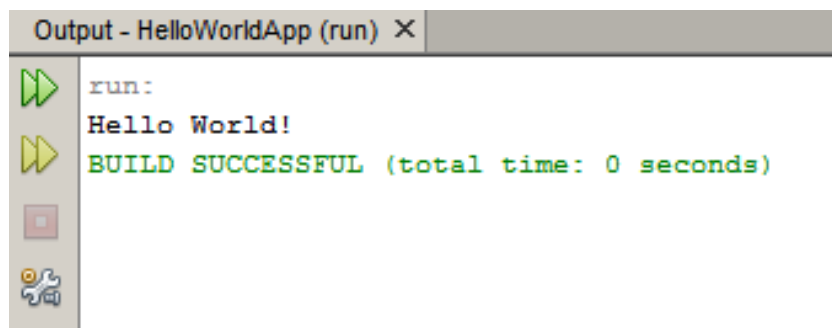


Figure 2.5

Congratulations! Your program works!

If there are compilation errors, they are marked with red glyphs in the left and right margins of the Source Editor. The glyphs in the left margin indicate errors for the corresponding lines. The glyphs in the right margin show all of the areas of the file that have errors, including errors in lines that are not visible. You can mouse over an error mark to get a description of the error. You can click a glyph in the right margin to jump to the line with the error.

2.2.4 Building and Deploying the Application

Once you have written and test run your application, you can use the Clean and Build command to build your application for deployment. When you use the Clean and Build command, the IDE runs a build script that performs the following tasks:

- Deletes any previously compiled files and other build outputs.
- Recompiles the application and builds a JAR file containing the compiled files.

To build your application:

- Choose Run > Clean and Build Project.

You can view the build outputs by opening the Files window and expanding the HelloWorldApp node. The compiled bytecode file HelloWorldApp.class is within the build/classes/helloworldapp subnode. A deployable JAR file that contains the HelloWorldApp → .class is within the dist node.

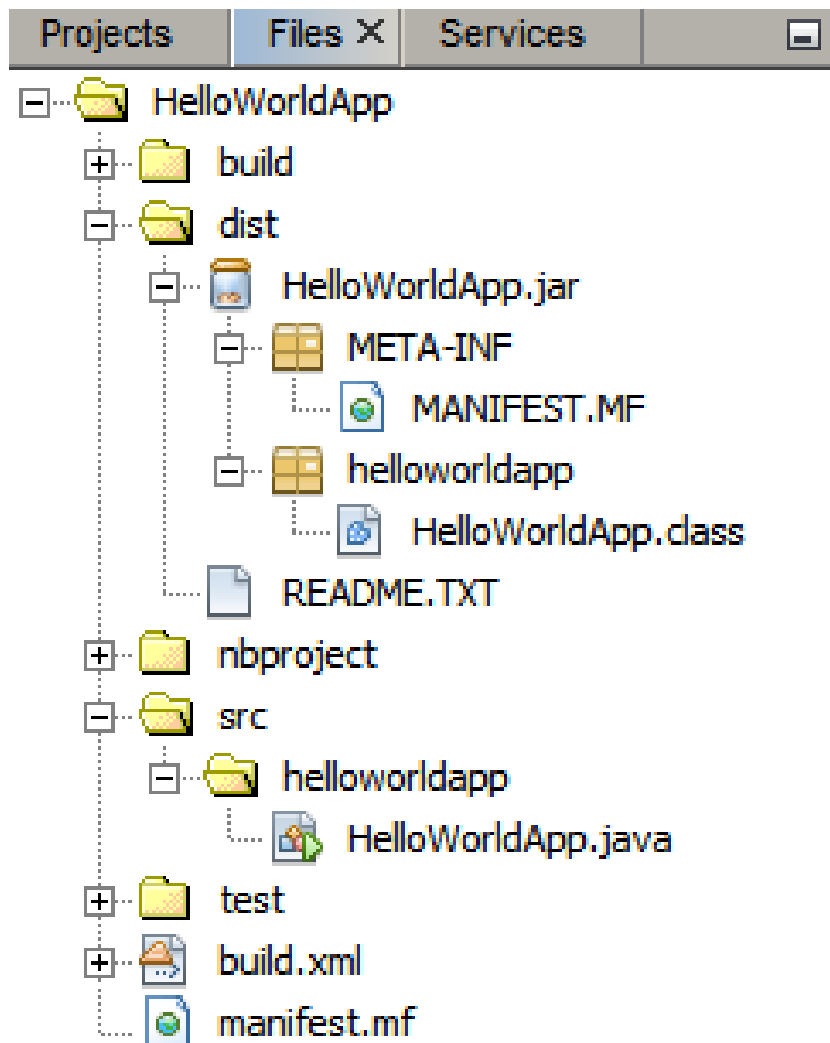


Figure 2.6

You now know how to accomplish some of the most common programming tasks in the IDE.

2.3 Exporting and importing projects

2.3.1 Exporting HelloWorldApp as a zip file

- Select the root node HelloWorldApp in the ‘Projects’ tab
- File > Export Project > To ZIP...
- save as a zip file named ‘HelloWorldApp’ where you want

2.3.2 Importing a project from a zip file

- File > Import Project > From ZIP...
- in ZIP file, browse to select the zip file containing the project: 'HelloWorldApp.zip' for instance
- in Folder, select the folder where the project is going to be created.



What you should remember from this chapter?

- How to create an Netbeans project?
- How to import and export a project as ZIP file

Chapter 3

The Java Basics

3.1 Improved Hello World!

You are going to write your first code.

Exercise 3.1

Using the Netbeans platform, create a project ImprovedHelloWorld. Inside the ‘Source Packages’ of your new project, create a package¹ ‘ense3’. Then create a ‘MyHelloWorld’ class i.e. a file that will contain the following Java code:

```
package ense3;

public class MyHelloWorld {

    public static void main(String[] args) {
        for(int i=0; i<5; i++)
            System.out.println("Hello World "+i+"!");
    }

}
```

Run the code and observe.

Once your code has run, double click in the left margin of the code view, next to the line `System.out.println("Hello World "+i+"!")`, to create a break point. Then, run step by step the code from the break point in debug mode. Observe the evolution of the variable *i*.

Let's now analyze the code of the ‘HelloWorld’ program.

¹a folder recognized by Java code

The first line `package ense3;` specifies to the Java virtual machine that the program is in the 'ense3' sub-folder of the root folder 'src' (displayed as 'Source Packages' in Netbeans) for the code. Even if that information could be guessed by analyzing folders, in Java, it has to be explicitly specified.

Then the name of the program (or `class` in Java) is specified in `public class MyHelloWorld` → `{ ... }`. Note that the file name should be exactly the same than the class name. Class 'MyHelloWorld' should be saved into a file 'MyHelloWorld.java'. Once the class has been created by Netbeans, you cannot change its name without renaming the file.

✎ Fortunately, Netbeans has refactoring capabilities that ease the renamings (right click on what you want to rename, Refactor > Rename...)

The 'public' keyword indicates that the class can be accessed from all the (upcoming) classes of the projects.

`public static void main(String[] args) { ... }` is a function bound to the class (program) 'MyHelloWorld'. In Java, conversely to C, functions are named methods. 'public' keyword has the same meaning that before. 'static' means the method is related to the class. 'void' means that the method does not return any value. `String[] args` indicates that the method requires a array of String as argument. 'args' could be named with any other labels provided it remains an array of String. It may be used to access arguments passed through the command such as `java HelloWorld arg1 arg2`. These arguments are generally not used but are still compulsory.

The method `public static void main(String[] args) { ... }` is special. Indeed, if its name, the type of arguments and of the return or the keywords are changed, the class could not be run anymore. In Java, only methods written in this way can be executed at launch. Other methods can be called but not at launch.

Inside the method, you will recognize the 'for' loop, with which you are now familiar because of C language. Let's focus on the command to print a String: `System.out` → `println("Hello World "+i+"!")`. To get a better understanding of the reason for such a complex writing `System.out.println`, you can use the official Java documentation. It must become usual for you to go to this web site to check the details of class. In the lower left corner, you can find the class 'System': <http://docs.oracle.com/javase/7/docs/api/java/lang/System.html>. You can see that the standard input (read from keyboard) and output (print to console) streams are bound to the System class. Printing on console requires to work with the output stream 'out'. Clicking on 'out' shows that its type is 'PrintStream' <http://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html>. Different 'println' methods (but also others) are available. `void println(String x)` is the method we are using in 'MyHelloWorld' class.

• It is very important to understand that all the Java classes are well documented in <http://docs.oracle.com/javase/7/docs/api/>

3.2 Java Naming Convention

Good developers follow naming convention to improve the readability of their codes. Oracle, that markets Java, proposes Java naming convention at <http://www.oracle.com/technetwork/java/codeconventions-135099.html>. We request you to use full English descriptions for names and to avoid using abbreviations. For example, use names like `firstName`, `lastName`, and `middleInitial` rather than the shorter versions `fName`, `lName`, and `mi`.

variable. Choose meaningful names that describe what the variable is being used for. Avoid generic names like `number` or `temp` whose purpose is unclear. Compose variable names using mixed case letters starting with a lower case letter. For example, use `salesOrder` and not `SalesOrder` or `sales_order`.

constant. Use ALL_UPPER_CASE for your named constants, separating words with the underscore character. For example, use `TAX_RATE` and not `taxRate` or `TAXRATE`.

class. Follow the variable naming convention but starts with an upper case letter like `'MyClass'`. Class names should refer to a type of things that can be manipulated. For instance, `'Shape'` or `'Matrix'`.

method. Begin method names with a strong action verb (for example, `'deposit'` or `'addInterest'`). It follows the variable naming convention but it starts by a lower case letter. Try to come up with meaningful method names that succinctly describe the purpose of the method.

getter and setter methods. Use the prefixes `get` and `set` for getter and setter methods. Getter methods merely return the value of a variable; setter methods change the value of a variable. For example, use the method names `getBalance` and `setBalance` to access or change the instance variable `balance`. If the method returns a boolean value, use `is` or `has` as the prefix for the method name. For example, use `isOverdrawn` or `hasCreditLeft` for methods that return `true` or `false` values.

arguments in a method. It should follow the same naming conventions as with variables, i.e. use mixed case, begin with a lower case letter, and begin each subsequent word with an upper-case letter. Consider using the prefix `a` or `an` with parameter names. This helps make the parameter distinguishable from other types of variables.

Variable and constant names are called identifiers.

3.3 Indenting code

Here is a proper way to indent a code (dots stand for space character) and to improve its readability:

```
public class MyClass {  
  
    ...public void greetUser(int currentHour) {  
        .....System.out.print( "Good");  
        .....if (currentHour < AFTERNOON) {  
            .....System.out.println( " Morning");  
            .....}  
        .....else if (currentHour < EVENING) {  
            .....System.out.println( "Afternoon");  
            .....} else {  
            .....System.out.println( "Evening");  
            .....}  
        ...}  
  
    }
```

Note that Netbeans can automatically reindent properly your code (Source > Format).

3.4 Commenting Java code

Comments provide readers with the information helpful to understand your program. Use comments to provide overviews or summaries of chunks of code and to provide additional information that is not available in the code itself, but avoid over-commenting your code. The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Use single-line comments, also called inline comments, to provide brief summary comments for chunks of code. Begin single-line comments with a double slash (//) that tells the compiler to ignore the rest of the line.

```
// Compute the exam average score for the midterm exam  
sumOfScores = 0;  
for (int i = 0; i < scores.length; i++)  
    sumOfScores = sumOfScores + scores[i];  
average = float(sumOfScores) / scores.length;
```

Trailing comments are used to provide an explanation for a single line of code. Begin trailing comments with a double slash (//) and place them to the right of the line of code they reference. Trailing comments are used to explain tricky code,

specify what abbreviated variable names refer to, or otherwise clarify unclear lines of code. In general, avoid the use of trailing comments. Instead rewrite tricky or unclear code, use meaningful variable names, and strive for self-documenting code.

```
ss = s1 + s2 + s3; //add the three scores into the sum
a = float(ss) / x; //calculate the mean of the scores
```

In addition, the C style comments `/* ... */` may be used temporarily for commenting out blocks of code during debugging

3.5 Java keywords

Keywords are special tokens in the language, which have reserved use in the language. Keywords may not be used as identifiers in Java. You cannot declare a field whose name is a keyword, for instance.

Examples of keywords are the primitive types (see next section), `int` and `boolean`; the control flow statements `for` and `if`; access modifiers such as `public`, and special words which mark the declaration and definition of Java classes, packages, and interfaces: `class`, `package`, `interface`.

You can find all the Java language keywords with explanations at https://en.wikibooks.org/wiki/Java_Programming/Keywords. The keywords in blue are similar to C language:

<code>abstract</code>	<code>assert</code>	<code>boolean</code>	<code>break</code>
<code>byte</code>	<code>byvalue</code>	<code>case</code>	<code>catch</code>
<code>char</code>	<code>class</code>	<code>const</code>	<code>continue</code>
<code>default</code>	<code>do</code>	<code>double</code>	<code>enum</code>
<code>else</code>	<code>extends</code>	<code>false</code>	<code>finally</code>
<code>float</code>	<code>for</code>	<code>goto</code>	<code>if</code>
<code>implements</code>	<code>import</code>	<code>instanceof</code>	<code>int</code>
<code>interface</code>	<code>long</code>	<code>native</code>	<code>new</code>
<code>null</code>	<code>package</code>	<code>private</code>	<code>protected</code>
<code>public</code>	<code>return</code>	<code>short</code>	<code>static</code>
<code>super</code>	<code>switch</code>	<code>synchronized</code>	<code>this</code>
<code>thread</code>	<code>safe</code>	<code>throw</code>	<code>transient</code>
<code>true</code>	<code>try</code>	<code>void</code>	<code>while</code>

3.6 Structuring Java codes into packages

Basically, a package is a folder of the hard drive that contains classes i.e. Java files. An example of package organization is given by:

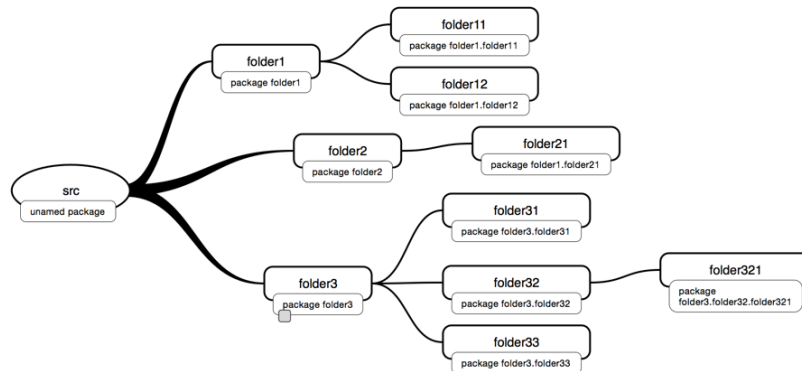


Figure 3.1 A typical Java program decomposed into packages

All the Java classes inside the ‘folder32’ must start with:

```
package folder3.folder32;
```

...

If a class ‘ClassA’ in folder21 has to use a class ‘ClassB’ from folder12, the file ‘ClassA.java’ will look like:

```
package folder3.folder32;
```

```
import folder1.folder12.ClassB;
```

....


All the classes from ‘folder21’ can be imported at once: `import folder2.folder21.*`

Last but not least, because the root ‘src’ folder has not name, it cannot be imported in other classes. It is therefore a bad habit to put classes in the root folder.

🔔 Important

What you should remember from this chapter?

- What is the basic structure of a Java code?
- How to create a package and a class?

- 
- How to run and debug a Java code?
 - How to get further information about a class at the official Java Doc web site?
 - What are the main naming conventions?
 - How to comment a code?

Chapter 4

Data types in Java

Data types are closed to the one you met in C language. Nevertheless, in Java, they are additional data types.

Common data types are:

- primitive data types
- Strings, for texts
- arrays
- objects, for new types

Important

Primitive data types are passed by values and all the other types by reference. Consider the following code:

```
public class PrimitiveCompoundType {  
  
    public static void display(int aValue, int[] anArray) {  
        aValue = aValue + 1;  
        anArray[0] = anArray[0] + 1;  
        System.out.println("i: " + aValue);  
        System.out.println("array: ");  
        for(int i=0; i<anArray.length; i++)  
            System.out.print(anArray[i] + " ");  
        System.out.println();  
    }  
  
    public static void main(String[] args) {  
        int value = 1;  
        int[] array = { 1,2,3,4};  
        display(value, array);  
        display(value, array);  
    }  
}
```

```
}
```

It yields:

```
i: 2
array:
2 2 3 4
i: 2
array:
3 2 3 4
```

The value of primitive type is not change for the variable 'value' whereas the first value of the array has changed. You will get the same result with all the primitive types. All the other compound types would be impacted by a change with one exception with the 'String' type, which is immutable. It means that, even if it is a compound type, it cannot be changed from within a function.

Exercise 4.1

Try to test the modification `aText = aText + "!"` of a String from within a function in adapting the previous code.

In object oriented programming, it is also possible to create new data types: the Object Data Types.

4.1 Primitive data types

Definition

Primitive types are the most basic data types available within the Java language; these include boolean, byte, char, short, int, long, float and double. These types serve as the building blocks of data manipulation in Java. Such types serve only one purpose — containing pure, simple values of a kind.

Because primitive data types are defined into the Java type system by default, they come with a number of predefined operations. You can not define a new operation for such primitive types. In the Java type system, there are three further categories of primitives:

Numeric primitives: short, int, long, float and double. These primitive data types hold only numeric data. Operations associated with such data types are those of simple arithmetic (addition, subtraction, etc.) or of comparisons (is greater than, is equal to, etc.)

Textual primitives: byte and char. These primitive data types hold characters (that can be Unicode alphabets or even numbers). Operations associated with such types are those of textual manipulations (comparing two words, joining characters to make words, etc.). However, byte and char can also support arithmetic operations.

Boolean and null primitives: boolean and null.

All the primitive types have a fixed size. Thus, the primitive types are limited to a range of values. A smaller primitive type (byte) can contain less values than a bigger one (long).

Category	Types	Size (bits)	Minimum Value	Maximum Value	Precision	Example
Integer	byte	8	-128	127	From +127 to -128	byte b = 65;
	char	16	0	$2^{16}-1$	All Unicode characters	char c = 'A'; char c = 65;
	short	16	-2^{15}	$2^{15}-1$	From +32,767 to -32,768	short s = 65;
	int	32	-2^{31}	$2^{31}-1$	From +2,147,483,647 to -2,147,483,648	int i = 65;
	long	64	-2^{63}	$2^{63}-1$	From +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808	long l = 65L;
Floating-point	float	32	2^{-149}	$(2-2^{-23}) \cdot 2^{127}$	From 3.402,823,5 E+38 to 1.4 E-45	float f = 65f;
	double	64	2^{-1074}	$(2-2^{-52}) \cdot 2^{1023}$	From 1.797,693,134,862,315,7 E+308 to 4.9 E-324	double d = 65.55;
Other	boolean	1	--	--	false, true	boolean b = true;
	void	--	--	--	--	--

Figure 4.1 Size of primitive data types

Data conversion (casting) can happen between two primitive types. There are two kinds of casting:

Implicit: casting operation is not required; the magnitude of the numeric value is always preserved. However, precision may be lost when converting from integer to floating point types

Explicit: casting operation required; the magnitude of the numeric value may not be preserved

For instance:

```
int i1 = 65;
long l1 = i1; // Implicit casting (int is converted to long, casting is not needed)
long l2 = 656666;
int i2 = (int) l2; // Explicit casting (long is converted to int, casting is needed)
```

The following table shows the conversions between primitive types, it shows the casting operation for explicit conversions:

	from byte	from char	from short	from int	from long	from float	from double	from boolean
to byte	-	(byte)	(byte)	(byte)	(byte)	(byte)	(byte)	N/A
to char		-	(char)	(char)	(char)	(char)	(char)	N/A
to short		(short)	-	(short)	(short)	(short)	(short)	N/A
to int				-	(int)	(int)	(int)	N/A
to long					-	(long)	(long)	N/A
to float						-	(float)	N/A
to double							-	N/A
to boolean	N/A	N/A	N/A	N/A	N/A	N/A	N/A	-

Figure 4.2 Conversions between primitive types

Unlike C, C++ and similar languages, Java can't represent false as 0 or null and can't represent true as non-zero. Java can't cast from boolean to a non-boolean primitive data type, or vice versa.

See https://en.wikibooks.org/wiki/Java_Programming/Primitive_Types for more.

4.2 Boolean expressions

Boolean values are values that evaluate to either true or false, and are represented by the boolean data type. Boolean expressions are very similar to mathematical expressions, but instead of using mathematical operators such as "+" or "-", you use comparative or boolean operators such as "==" or "!=".

Java has several operators that can be used to compare variables. For example, how would you tell if one variable has a greater value than another? The answer: use the "greater-than" operator.

Here is a list of the comparative operators in Java:

- >: greater than
- <: lower than
- >=: greater equal than
- <=: lower equal than
- ==: equal to
- !=: not equal to

To see how these operators are used, look at this example:

```
int a = 5, b = 3;
System.out.println(a > b); // Value is true because a is greater than b
System.out.println(a == b); // Value is false because a does not equal b
```

```
System.out.println(a != b); // Value is true because a does not equal b
System.out.println(b <= a); // Value is true because b is less than a
```

It comes out:

```
true
false
true
true
```

Comparative operators can be used on any primitive types (except boolean), but only the "equals" and "does not equal" operators work on objects. This is because the less-than/greater-than operators cannot be applied to objects, but the equivalent operators can. Specifically, the == and != operators test whether both variables point to the same object.

The Java boolean operators are based on the operations of the boolean algebra. The boolean operators operate directly on boolean values.

Here is a list of four common boolean operators in Java:

- !: Boolean NOT
- &&: Boolean AND
- ||: Boolean inclusive OR
- ^: Boolean exclusive XOR

The boolean NOT operator (!) inverts the value of a boolean expression. The boolean AND operator (&&) will result in true if and only if the values on both sides of the operator are true. The boolean inclusive OR operator (||) will result in true if either or both of the values on the sides of the operator is true. The boolean exclusive XOR operator (^) will result in true if one and only one of the values on the sides of the operator is true.

To show how these operators are used, here is an example:

```
boolean iMTrue = true;
boolean iMTrueToo = true;
boolean iMFalse = false;
boolean iMFalseToo = false;

System.out.println("NOT operand:");
System.out.println(!iMTrue);
System.out.println(!iMFalse);
System.out.println(!(4 < 5));
System.out.println("AND operand:");
System.out.println(iMTrue && iMTrueToo);
System.out.println(iMFalse && iMFalseToo);
System.out.println(iMTrue && iMFalse);
```

```
System.out.println(iMTrue && !iMFalse);
System.out.println("OR operand:");
System.out.println(iMTrue || iMTrueToo);
System.out.println(iMFalse || iMFalseToo);
System.out.println(iMTrue || iMFalse);
System.out.println(iMFalse || !iMTrue);
System.out.println("XOR operand:");
System.out.println(iMTrue ^ iMTrueToo);
System.out.println(iMFalse ^ iMFalseToo);
System.out.println(iMTrue ^ iMFalse);
System.out.println(iMFalse ^ !iMTrue);
```

The result is given by:

NOT operand:

false

true

false

AND operand:

true

false

false

true

OR operand:

true

false

true

false

XOR operand:

false

false

true

false

4.3 Strings in Java

Strings, which are widely used in Java programming, are sequences of characters. In the Java programming language, strings are objects.

The Java platform provides the String class to create and manipulate strings.

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!". It is stored into memory just

like with C language. The reference (pointer) is bound to the variable label but, conversely to C, it is almost transparent for the programmer.

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the `length()` method, which returns the number of characters contained in the string object.

Below the program gives an example of usage for ‘`length()`’:

```
public class StringDemo {  
  
    public static void main(String args[]) {  
        String palindrome = "Dot saw I was Tod";  
        int len = palindrome.length();  
        System.out.println( "String Length is : " + len );  
    }  
}
```

The String class includes a method for concatenating two strings:

```
String string1 = "My name is ";  
String string2 = "Jean";  
String string3 = string1.concat(string2);  
String string4 = string1 + string2;
```

‘string3’ and ‘string4’ point both to a String “My name is Jean”

Comparing strings is not as easy as it may first seem. Be aware of what you are doing when comparing String’s using `==`:

```
public class StringTest {  
  
    public static void main(String[] args) {  
        String greeting = new String("Hello World!");  
        if (greeting == "Hello World!") {  
            System.out.println("Match found.");  
        } else {  
            System.out.println("Match not found.");  
        }  
        if (greeting.equals("Hello World!")) {  
            System.out.println("Match found.");  
        } else {  
            System.out.println("Match not found.");  
        }  
    }  
}
```

It yields:

Match not found.

Match found.

The difference between the ‘==’ and ‘equals’ comparisons is that the ‘==’ checks to see whether objects are same i.e. in the same position in memory (similarly to C, pointing to the same object). The ‘equals’ test compares the actual content of strings.

For more information, visit https://en.wikibooks.org/wiki/Java_Programming/API/java.lang.String.

4.4 Arrays in Java

There are two kinds of arrays: the static ones, whose size cannot be changed and the dynamic ones. Other sequences such as Set or Map are also available.

4.4.1 Static arrays

Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

```
dataType[] arrayRefVar; // preferred way.  
dataType arrayRefVar[]; // works but not preferred way.
```

You can create an array by using the new operator with the following syntax:

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things:

- It creates an array using new dataType[arraySize];

- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

- It creates an array using `new dataType[arraySize];`
- It assigns the reference of the newly created array to the variable arrayRefVar.

```
dataType[] arrayRefVar = new dataType[arraySize];
```

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

Here is a complete example of showing how to create, initialize and process arrays:

```
public class TestArray {

    public static void main(String[] args) {
        double[] myList = {1.9, 2.9, 3.4, 3.5};

        // Print all the array elements
        for (int i = 0; i < myList.length; i++) {
            System.out.println(myList[i] + " ");
        }
        // Summing all elements
        double total = 0;
        for (int i = 0; i < myList.length; i++) {
            total += myList[i];
        }
        System.out.println("Total is " + total);
        // Finding the largest element
        double max = myList[0];
        for (int i = 1; i < myList.length; i++) {
            if (myList[i] > max) max = myList[i];
        }
        System.out.println("Max is " + max);
    }
}
```

It leads to:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

JDK 1.5 introduced a new for loop known as for-each loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

```
public class TestArray {
```

```

public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (double element: myList) {
        System.out.println(element);
    }
}

```

It results in:

```

1.9
2.9
3.4
3.5

```

Multi-dimensional arrays can be defined just like in C language:

```

String[][] twoDimArray = {{ "a", "b", "c", "d", "e"},
                           { "f", "g", "h", "i", "j"},
                           { "k", "l", "m", "n", "o"} };

int[][] twoDimIntArray = {{ 0, 1, 2, 3, 4},
                           {10, 11, 12, 13, 14},
                           {20, 21, 22, 23, 24}};

String[] oneDimArray = { "00", "01", "02", "03", "04" };
String[][] twoDimArray2 = {oneDimArray,
                           { "10", "11", "12", "13", "14"},
                           { "20", "21", "22", "23", "24"} };

String[][] weirdTwoDimArray = {{ "10", "11", "12"},
                                null,
                                { "20", "21", "22", "23", "24"} };

```

Referring to “21” in the last array can be done with `twoDimArray2[2][1]` and to “03” with `twoDimArray2[0][3]`. Indeed, a multi-dimensional array is a array of arrays. Note that row can be missing and that the size of inner arrays can be different, just like in the ‘weirdTwoDimArray’.

To learn more, see https://en.wikibooks.org/wiki/Java_Programming/Arrays.

4.4.2 Dynamic arrays

In Java, the easier way to create a dynamic array (dimensions can be dynamically modified) is to use `java.util.ArrayList`. Here is an example:

```
import java.util.ArrayList;
```



```

public class DynamicArray {

    public static void main(String[] args) {
        ArrayList dynamicSerie = new ArrayList();
        double[] staticSerie = { 1.3, 2.1, 4.5, 3.2};
        for (double value: staticSerie)
            dynamicSerie.add(value);
        dynamicSerie.add(5.6);
        System.out.println(dynamicSerie);
        double sum = 0;
        for(int i=0;i<dynamicSerie.size();i++)
            sum = sum + (double)dynamicSerie.get(i);
        System.out.println(sum);
    }
}

```

It yields:

```

[1.3, 2.1, 4.5, 3.2, 5.6]
16.700000000000003

```

The cast `(double)` is compulsory in the line `sum = sum + (double)dynamicSerie.get(i);` → ↵. Indeed, a dynamic array can contain any kind of types (or objects) but Java should explicitly mention the type to be able to detect consistency problem before execution, just using syntax analysis.

It is nevertheless possible to specify what a dynamic table will contain. The following code is equivalent to the previous one but casting is not required here:

```
import java.util.ArrayList;
```

```

public class DynamicArray {

    public static void main(String[] args) {
        ArrayList<Double> dynamicSerie = new ArrayList<Double>();
        double[] staticSerie = { 1.3, 2.1, 4.5, 3.2};
        for (double value: staticSerie)
            dynamicSerie.add(value);
        dynamicSerie.add(5.6);
        System.out.println(dynamicSerie);
        double sum = 0;
        for(int i=0;i<dynamicSerie.size();i++)
            sum = sum + dynamicSerie.get(i);
        System.out.println(sum);
    }
}

```

Note that double is starting by an upper case because primitive type cannot be used here but the corresponding object type. The following correspondence holds for primitive types:

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

Regarding `java.util.ArrayList`, the following methods are available (for more see official JavaDoc <http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>):

add(E e): boolean Appends the specified element to the end of this list.

add(int index, E element): void Inserts the specified element at the specified position in this list.

clear(): void Removes all of the elements from this list.

clone(): Object Returns a shallow copy of this `ArrayList` instance.

contains(Object o): boolean Returns true if this list contains the specified element.

get(int index): E Returns the element at the specified position in this list.

indexOf(Object o): int Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

isEmpty(): boolean Returns true if this list contains no elements.

lastIndexOf(Object o): int Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

remove(int index): E Removes the element at the specified position in this list.

remove(Object o): boolean Removes the first occurrence of the specified element from this list, if it is present.

size(): int Returns the number of elements in this list.

toArray(): Object[] Returns an array containing all of the elements in this list in proper sequence (from first to last element).

toArray(T[] a): T[] Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Note that 'E' stands for the type specified at the `ArrayList` creation `new ArrayList<E>`. If 'E' is 'Double', replace 'E' by 'Double' everywhere. If no type is specified, by default, E is an 'Object'.

You can find more information about `ArrayList` at https://en.wikibooks.org/wiki/Java_Programming/ArrayList

4.4.3 Other collections of variables

Two other kinds of collections exist: the sets in accordance with mathematical concept of ‘set’, and the maps, which consist of a collections of couples of values ‘(key,value)’ where value can be obtained thanks to the ‘key’.

Set

The following example illustrates the concept of ‘Set’:

```
import java.util.*;
public class SetExample {

    static public void main(String[] args) {
        HashSet<String> set = new HashSet<String>();
        set.add("Bernard");
        set.add("Elisabeth");
        set.add("Renée");
        set.add("Elisabeth");
        set.add("Clara");
        System.out.println(set);
        TreeSet<String> sortedSet = new TreeSet<String>(set);
        System.out.println(sortedSet);
    }
}
```

In the console, we get:

```
[Renée, Clara, Elisabeth, Bernard]
[Bernard, Clara, Elisabeth, Renée]
```

We can see that even if “Elisabeth” element has been added twice, it is considered as the same element. That’s why it appears only once. Conversely to ‘HashSet’, ‘TreeSet’ returns its elements in a sorted way.

Here are the main methods related to ‘Set<E>’:

add(E e): boolean Adds the specified element to this set if it is not already present.
clear(): void Removes all of the elements from this set (optional operation).
contains(Object o): boolean Returns true if this set contains the specified element.
equals(Object o): boolean Compares the specified object with this set for equality.
isEmpty(): boolean Returns true if this set contains no elements.
remove(Object o): boolean Removes the specified element from this set if it is present.

size(): int Returns the number of elements in this set (its cardinality).
toArray(): Object[] Returns an array containing all of the elements in this set.
toArray(T[] a): T[] Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

Official JavaDoc is available at <http://docs.oracle.com/javase/7/docs/api/java/util/Set.html>, <http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html> and <http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html>.

Map

Similarly to sets, there are two kinds of maps: the standard ones ('HashMap') and the sorted ones ('TreeMap').

Here is an example that illustrates the usage of both. It computes the number of times a word appears in a text.

```
import java.util.*;
public class MapExample { //count the occurrences of words in a text

    public static void main(String args[]) {
        String text = "Lorem ipsum dolor sit amet, consectetur adipiscing"
            + " elit, sed du eiusmod tempor incididunt ut labore et"
            + " dolore magna aliqua. Ut enim ad minim veniam, quis"
            + " nostrud exercitation ullamco labori nisi ut aliquip"
            + " ex ea commodo consequat. Duis aute irure dolor in"
            + " reprehenderit in voluptate velit esse cillum dolore"
            + " eu fugiat nulla pariatur. Excepteur sint occaecat"
            + " cupidatat non proident, sunt in culpa qui officia"
            + " deserunt mollit anim id est laborum";
        String[] words = text.split("[\\s|\\.|,]"); //regular expression
        // for splitting at a space, a dot or a comma.
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        for (int i=0, n=words.length; i<n; i++) {
            String word = words[i];
            if (word.length() > 0) {
                int frequency = 1;
                if (!map.containsKey(word))
                    frequency = 1;
                else
                    frequency = map.get(word) + 1;
                map.put(word, frequency);
            }
        }
        System.out.println(map);
        TreeMap<String, Integer> sortedMap = new TreeMap<String, Integer>(map);
        System.out.println(sortedMap);
    }
}
```

It displays:

```
{incididunt=1, culpa=1, mollit=1, tempor=1, adipiscing=1, est=1, commodo=1, aute=1, laborum →
↳ =1, cupidatat=1, proident=1, du=1, officia=1, deserunt=1, cillum=1, voluptate=1, id=1, →
↳ ea=1, sunt=1, ut=2, enim=1, occaecat=1, ad=1, consequat=1, in=3, velit=1, eiusmod=1, →
↳ Excepteur=1, sint=1, et=1, esse=1, eu=1, ex=1, ipsum=1, exercitation=1, Duis=1, anim →
↳ =1, elit=1, nostrud=1, Ut=1, qui=1, pariatur=1, minim=1, veniam=1, fugiat=1, non=1, →
↳ dolor=2, sed=1, sit=1, nisi=1, irure=1, labore=1, ullamco=1, labori=1, magna=1, aliquip →
↳ =1, dolore=2, Lorem=1, amet=1, aliqua=1, quis=1, reprehenderit=1, consectetur=1, →
↳ nulla=1}
{Duis=1, Excepteur=1, Lorem=1, Ut=1, ad=1, adipiscing=1, aliqua=1, aliquip=1, amet=1, anim →
↳ =1, aute=1, cillum=1, commodo=1, consectetur=1, consequat=1, culpa=1, cupidatat=1, →
↳ deserunt=1, du=1, dolor=2, dolore=2, ea=1, eiusmod=1, elit=1, enim=1, esse=1, est=1, →
↳ et=1, eu=1, ex=1, exercitation=1, fugiat=1, id=1, in=3, incididunt=1, ipsum=1, irure=1, →
↳ labore=1, labori=1, laborum=1, magna=1, minim=1, mollit=1, nisi=1, non=1, nostrud →
↳ =1, nulla=1, occaecat=1, officia=1, pariatur=1, proident=1, qui=1, quis=1, reprehenderit →
↳ =1, sed=1, sint=1, sit=1, sunt=1, tempor=1, ullamco=1, ut=2, velit=1, veniam=1, →
↳ voluptate=1}
```

The main methods for ‘Map<K,V>’, where ‘K’ is the type of the keys and ‘V’, the type of the corresponding value, are:

clear(): void Removes all of the mappings from this map.

containsKey(Object key): boolean Returns true if this map contains a mapping for the specified key.

containsValue(Object value): boolean Returns true if this map maps one or more keys to the specified value.

equals(Object o): boolean Compares the specified object with this map for equality.

get(Object key): V Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

isEmpty(): boolean Returns true if this map contains no key-value mappings.

put(K key, V value): V Associates the specified value with the specified key in this map.

remove(Object key): V Removes the mapping for a key from this map if it is present (optional operation).

size(): int Returns the number of key-value mappings in this map.

values(): Collection<V> Returns a Collection view of the values contained in this map.

More information can be found at https://en.wikibooks.org/wiki/Java_Programming/Map.

The official JavaDoc is available at <http://docs.oracle.com/javase/7/docs/api/java/util/Map.html>, <http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html> and <http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html>.

4.5 New types: classes and objects

- Very important to remember for the next.

In object oriented programming, new types can be defined. There are named ‘Object’. For instance, let’s create a new type, i.e. an object, ‘Rectangle’:

```
public class Rectangle {
    double width;
    double height;
    double perimeter;
    double surface;

    public Rectangle(double aWidth, double aHeight) { // constructor of the new type
        width = aWidth;
        height = aHeight;
        perimeter = 2 * (width + height);
        surface = width * height;
    }

    public static void main(String[] args) {
        Rectangle rectangle1 = new Rectangle(10,5);
        Rectangle rectangle2 = new Rectangle(4,7);
        System.out.println("[rectangle1] width: "+rectangle1.width+", surface: " →
        ↪ +rectangle1.surface);
        System.out.println("[rectangle2] width: "+rectangle2.width+", surface: " →
        ↪ +rectangle2.surface);
    }
}
```

It yields:

```
[rectangle1] perimeter: 10.0, surface: 50.0
[rectangle2] perimeter: 4.0, surface: 28.0
```

As you can observe, the new type Rectangle is a bit more than a structure in C language. Indeed, there is a special method whose name is similar to the name of the class (i.e. also similar to the file name) that is used to construct a element of the type ‘Rectangle’. It is named constructor and it is called when ‘new’ is invoked. You can see that the type contains the variables perimeter and surface that have not been provided when ‘new’ was called. The values have been processed by the constructor. In Java, the new type is named class and it has to be defined in a file holding the same name. A new element of the type (class), such as ‘rectangle1’ or ‘rectangle2’, is called an object or an instance of the new type or class.

Thanks to this mechanism, many new types can be defined and it is widely used in object oriented programming.

What you should remember from this chapter

- What are the main primitive data types?
- What the compound data types?
- How and when to cast a value?
- How to compare strings?
- What type of function arguments are passed by value? by reference?
- How to create a static array?
- How to create a dynamic array?
- What is a set?
- What is a map?
- How to create new data types?

Chapter 5

Processing data in Java

Format for testing and looping in a code are quite similar to the C language.

5.1 Conditional statements

5.1.1 If statement

Conditional blocks allow a program to take a different path depending on some condition(s). It allows a program to perform a test and then take action based on the result of that test.

The if block executes only if the boolean expression associated with it is true. The structure of an if block is as follows:

```
if (boolean expression1) {  
    statement_1;  
    statement_2;  
    ...  
    statement_n;  
}
```

Curly brackets can be omitted if there is only one statement in the block.

Here is an example to illustrate what happens if the condition is true and if the condition is false:

```
int age = 6;  
System.out.println("Hello!");  
if (age < 13) {
```

```

    System.out.println("I'm a child.");
}
if (age > 20)
    System.out.println("I'm an adult.");
System.out.println("Bye!");

```

It leads to:

```

Hello!
I'm a child
Bye!

```

The if block may optionally be followed by else-if and else blocks which will execute depending on conditions. The structure such a block is as follows:

```

if (boolean expression1) {
    statement_1.1;
    statement_1.2;
    ...
    statement_1.n;
} else if (boolean expression2) {
    statement_2.1;
    statement_2.2;
    ...
    statement_2.n;
} ...

} else if (boolean expression_m_minus1) {
    statement_m_minus1.1;
    statement_m_minus1.2;
    ...
    statement_m_minus1.n;
} else {
    statement_m.1;
    statement_m.2;
    ...
    statement_m.n;
}

```

Here is an example to illustrate:

```

public class MyConditionalProgram {

    public static void main (String[] args) {
        int a = 5;
        if (a > 0) { // a is greater than 0, so this statement will execute
            System.out.println("a is positive");
        } else if (a >= 0) { // a case has already executed, so this statement will NOT execute
            System.out.println("a is positive or zero");
        } else { // a case has already executed, so this statement will NOT execute
            System.out.println("a is negative");
        }
    }
}

```

```

    }
}

```

It comes out a is positive.

Conditional expressions use the compound ‘?:’ operator. The syntax is: `boolean` → `boolean_expression ? expression1 : expression2`. For example: `String answer = (p < 0.05)?` → `"reject" : "keep"`; is equivalent to:

```

String answer;
if (p < 0.05) {
    answer = "reject";
} else {
    answer = "keep";
}

```

5.1.2 Switch statement

The switch conditional statement is basically a shorthand version of writing many if...else statements. The switch block evaluates a char, byte, short, or int (or enum, starting in J2SE 5.0; or String, starting in J2SE 7.0), and, based on the value provided, jumps to a specific case within the switch block and executes code until the break command is encountered or the end of the block. If the switch value does not match any of the case values, execution will jump to the optional default case.

The structure of a switch statement is as follows:

```

switch (int1 or char1 or short1 or byte1 or enum1 or String value1) {
    case value1:
        statement1.1
        ...
        statement1.n
        break;
    case value2:
        statement2.1
        ...
        statement2.n
        break;
    default:
        statementn.1
        ...
        statementn.n
}

```

Here is an example:

```

int i = 3;

```

```

switch(i) {
    case 1: // i doesn't equal 1, so this code won't execute
        System.out.println("i equals 1");
        break;
    case 2: // i doesn't equal 2, so this code won't execute
        System.out.println("i equals 2");
        break;
    default: // i has not been handled so far, so this code will execute
        System.out.println("i equals something other than 1 or 2");
}

```

If a case does not end with the break statement, then the next case will be checked, otherwise the execution will jump to the end of the switch statement.

Look at this example to see how it is done:

```

int i = -1;
switch(i) {
    case -1:
        case 1: // i is -1, so it will fall through to this case and execute this code
            System.out.println("i is 1 or -1");
            break;
        case 0: // The break command is used before this case, so if i is 1 or -1, this will not execute
            System.out.println("i is 0");
}

```

It comes out: i is 1 or -1.

Starting in J2SE 7.0, the switch statement can also be used with an String value instead of an integer:

```

String day = "Monday";
switch(day) {
    case "Monday": // Since day == "Monday", this statement will execute
        System.out.println("Mondays are the worst!");
        break;
    case "Tuesday":
    case "Wednesday":
    case "Thursday":
        System.out.println("Weekdays are so-so.");
        break;
    case "Friday":
    case "Saturday":
    case "Sunday":
        System.out.println("Weekends are the best!");
        break;
    default:
        throw new IllegalArgumentException("Invalid day of the week: " + day);
}

```

The result is Mondays are the worst!

5.2 Loop statements

Loops are a handy tool that enables programmers to do repetitive tasks with minimal effort.

5.2.1 While statement

‘while’ loops are the simplest form of loop. The while loop repeats a block of code while the specified condition is true. Here is the structure of a while loop:

```
while (boolean expression) {  
    statement_1;  
    statement_2;  
    ...  
    statement_n;  
}
```

The loop’s condition is checked before each iteration of the loop. If the condition is false at the start of the loop, the loop will not be executed at all. The following code sets in squareHigherThan200 the smallest integer whose square exceeds 200.

```
int squareHigherThan200 = 0;  
while (squareHigherThan200 * squareHigherThan200 < 200) {  
    squareHigherThan200 = squareHigherThan200 + 1;  
}
```

If a loop’s condition will never become false, such as if the true constant is used for the condition, said loop is known as an infinite loop. Such a loop will repeat indefinitely unless it is broken out of. Infinite loops can be used to perform tasks that need to be repeated over and over again without a definite stopping point, such as updating a graphics display.

5.2.2 Do-while statement

The do-while loop is functionally similar to the while loop, except the condition is evaluated AFTER the statement executes. It is useful when we try to find a data that does the job by randomly browsing an amount of data.

```
do {  
    statement_1;
```

```
statement_2;  
...  
statement_n;  
} while (boolean expression);
```

5.2.3 For statement

The for loop is a specialized while loop whose syntax is designed for easy iteration through a sequence of numbers. It consists of the keyword `for` followed by three extra statements enclosed in parentheses. The first statement is the variable declaration statement, which allows you to declare one or more integer variables. The second is the condition, which is checked the same way as the while loop. Last is the iteration statement, which is used to increment or decrement variables, though any statement is allowed.

This is the structure of a for loop:

```
for (variable declarations; condition; iteration statement) {  
    statement_1;  
    statement_2;  
    ...  
    statement_n;  
}
```

Here is a very basic example:

```
for (int i = 1; i <= 10; i++) {  
    System.out.println(i);  
}
```

Another not so basic example:

```
for (int i = 1, j = 10; i <= 10; i++, j--) {  
    System.out.print(i + " ");  
    System.out.println(j);  
}
```

5.2.4 For-each statement

The for-each loop automatically iterates through an array and assigns the value of each index to a variable.

To understand the structure of a for-each loop, look at the following example:

```
String[] sentence = { "I", "am", "a", "Java", "program." };
for (String word : sentence) {
    System.out.print(word + " ");
}
```

The example iterates through an array of words and prints them out like a sentence. What the loop does is iterate through sentence and assign the value of each index to word, then execute the code block.

Here is the general contract of the for-each loop:

```
for (variable declaration : array or list) {
    statement_1;
    statement_2;
    ...
    statement_n;
}
```

5.2.5 Complements

The ‘break’ keyword exits a flow control loop, such as a for loop. It basically breaks the loop.

In the following code, the loop would print out all the numbers from 1 to 10, but we have a check for when i equals 5. When the loop reaches its fifth iteration, it will be cut short by the ‘break’ statement, at which point it will exit the loop.

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
    if (i == 5) {
        System.out.println("STOP!");
        break;
    }
}
```

Result is:

```
1
2
3
4
5
STOP!
```

The ‘continue’ keyword jumps straight to the next iteration of a loop and evaluates the boolean expression controlling the loop. The following code is an example of

the ‘continue’ statement in action:

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        System.out.println("Caught i == 5");
        continue;
    }
    System.out.println(i);
}
```

It yields:

```
1
2
3
4
Caught i == 5
6
7
8
9
10
```

As the break and continue statements reduce the readability of the code, it is recommended to reduce their use or replace them with the use of ‘if’ and ‘while’ blocks.

Labels can be used to give a name to a loop. The reason to do this is so we can break out of or continue with upper-level loops from a nested loop.

Here is how to label a loop: `label name:loop`

To break out of or continue with a loop, use the break or continue keyword followed by the name of the loop.

For example:

```
int i, j;
int[][] nums = { {1, 2, 5}, {6, 9, 7}, {8, 3, 4} };
Outer:
for (i = 0; i < nums.length; i++) {
    for (j = 0; j < nums[i].length; j++) {
        if (nums[i][j] == 9) {
            System.out.println("Found number 9 at (" + i + ", " + j + ")");
            break Outer;
        }
    }
}
```


It results in Found number 9 at (1, 1).

Look at how the label is used to break out of the outer loop from the inner loop. However, as such a code is hard to read and maintain, it is highly recommended not to use labels.

5.3 Methods rather than functions

Methods can be just like methods in C i.e. bound to a class contained in a file named identically. In this case, the keyword ‘static’ (named modifier) is used. Here is an example:

```
public class StaticExample {

    static double result;

    static double sum(double x, double y) {
        return x * y;
    }

    static double mul(double x, double y) {
        return x + y;
    }

    public static void main(String[] args) {
        result = 0;
        result = result + sum(2, 5);
        result = result + mul(4, 3);
        System.out.println(result);
    }
}
```

It prints 17. If you work with C-style function (it is quite unusual in Java because it relies on an object oriented paradigm), and variables, you have to declare them as ‘static’ to bind them to the class i.e. to the file.

At this point, there is no real difference with C except for slight syntax differences. But Java is intrinsically an object oriented language. We have seen that new types, named objects, can be defined but it was already possible in C using structures. What was not possible is to bind functions to new user-defined types i.e. to objects.

Here is an object oriented version of the previous code:

```
public class Accumulator {
    double result; // attributes of the new type
```

- Very important for the next: it deals the object oriented programming core philosophy.

```

Accumulator(){ // constructor of the new type
    result = 0;
}

void add(double x, double y) { // method bound to the new type
    result = result + (x + y);
}

void mul(double x, double y) { // method bound to the new type
    result = result + (x * y);
}

public String toString() { // method bound to the new type
    return "result is: " + result;
}

static public void main(String[] args) {
    Accumulator accumulator1 = new Accumulator();
    accumulator1.add(2, 5);
    accumulator1.mul(4, 3);
    System.out.println(accumulator1); // call implicitly the toString() method
}
}

```

A new type ‘Accumulator’ has been created: it embeds its own variables, named attributes and its own functions that operate on the type attribute ‘result’. Comparing to the previous version, the ‘static’ modifier is no longer present. In Java, each element of a new type is named ‘instance’ or ‘object’. There are obtained thanks to the ‘new’ keyword. Once ‘new’ is invoked, the class constructor is called. The constructor is a special method, which hold the same name that the class and whose return type is implicitly void.

Of course, functions bind to class (i.e. to the file), named static methods, can be mixed with functions bind to a new type, named object or instance methods, but there are some limitations. Object methods can access static variables and methods bound to the class but, the opposite is not possible. Indeed, a static method cannot access to object variables and methods because they operate on specific values that are defined each time ‘new’ is invoked. Therefore their life cycles are different:

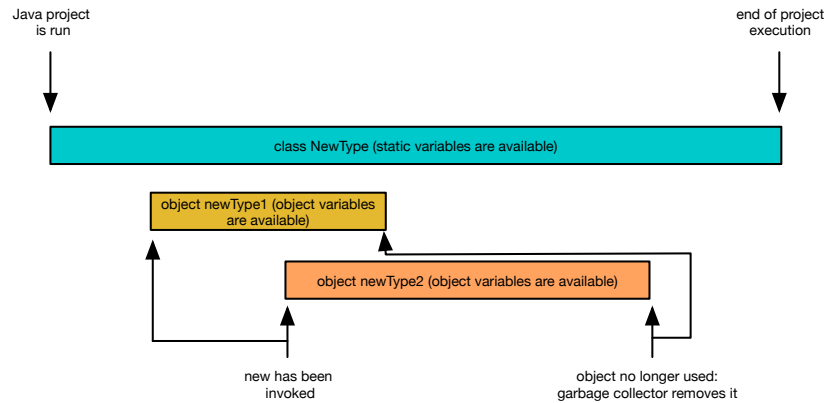


Figure 5.1 Life cycle of static and object variables

A little bit of philosophy

Object oriented programming relies on new types i.e. classes and generated objects. The objects are defined in such a way they can be mapped to existing meaningful entities. As a consequence, structure of codes is consistent with reality. It improves a lot the readability of codes but it also makes codes more stable. Indeed, objects and methods are bound to meaningful object instead of being spread in a non-meaningful way in a program. When code is evolving, the structure remains stable because it is mapped to objects from reality.

Here is example that mixes both kinds of variables and methods. Consider the files ‘Bicycle.java’:

```

class Bicycle {

    static int NUMBER_OF_WHEELS = 2;
    int numberOfGears;
    int currentGear;

    Bicycle(int aNumberOfGears) {
        numberOfGears = aNumberOfGears;
        currentGear = 1;
    }

    static double convertMiles2Kmh(double speedInMilesPerHour) {
        return speedInMilesPerHour/1.6;
    }

    void changeGear(int newGear) {
        currentGear = newGear;
    }

    static void staticChangeGear(int newGear) {
        currentGear = newGear; // error current gear is bound to a specific bike:
                               // it is not accessible from outside the object
    }
}
  
```

```

    int getCurrentGear() {
        return currentGear;
    }
}

```

and 'Person.java':

```

class Person {

    String name;
    Bicycle bicycle;

    Person(String name) {
        this.name=name;
        double distanceInKmh = Bicycle.convertMiles2Kmh(80); // allowed: method bound to class
        bicycle = new Bicycle(10);
        bicycle.changeGear(2); // allowed: methods applying to bicycle data
    }

    String getName() {
        return name;
    }

    Bicycle getBicycle() {
        return bicycle;
    }

    public static void main(String[] args) {
        Person me = new Person("me");
        System.out.println(Bicycle.convertMiles2Kmh(50));
        System.out.println(me.getBicycle().getCurrentGear());
    }
}

```

Running the class 'Person' leads to:

```

31.25
2

```

Just analyze how the methods are called. Static method are called by prefixing with the class name such as `Bicycle.convertMiles2Kmh(50)` (notice the upper case in 'Bicycle'), whereas other methods are related to an instance generated by the class `me.getBicycle()`, denoted by 'me' here. `currentGear = newGear;` is not possible in `static` → `void staticChangeGear(int newGear)` because static methods cannot access to none static variables because of life cycle issues.

Difference between static and object variables and methods can be also represented

by:

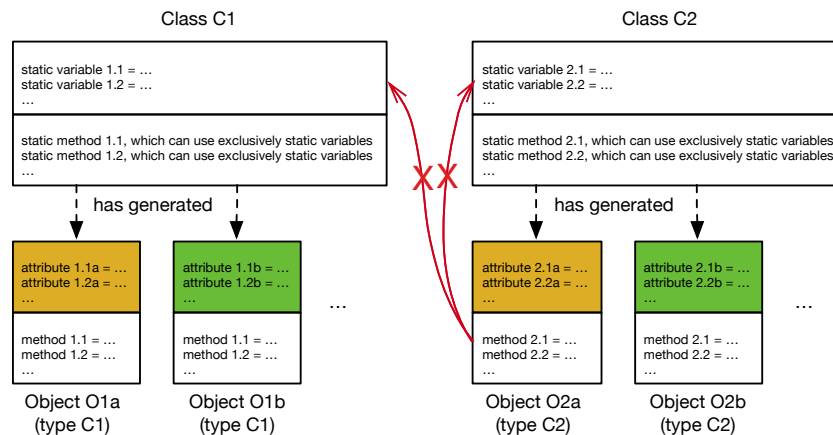


Figure 5.2 Static and non-static variables and methods

To learn more about methods, read https://en.wikibooks.org/wiki/Java_Programming/Methods and http://www.tutorialspoint.com/java/java_methods.htm.

5.4 Scope of variables

The scope of a class, a variable or a method is its visibility and its accessibility. The visibility or accessibility means that you can use the item from a given place.

Modifiers related to visibility

If a developer was working alone, he would not really need to hide some variables and methods depending on from where someone wants to call them. But working with others supposed some means to say: this variable or this method should be used from outside the current class because it is used for local computations. Java contains a large set of modifiers (keywords) to define precise access rights to variables and methods.

Here is the meaning of the different visibility modifiers:

✍ At this step, you will probably not understand all the table but you will later. Just remember where it is.

visibility	public	protected	[no modifier]	private
from inside a class or an object	yes	yes	yes	yes
from inside a package	yes	yes	yes	
from a child class	yes	yes		
from outside package and child class	yes			

Figure 5.3 Visibility of variables and methods depending on the modifiers used

The scope of a class, a variable or a method is its visibility and its accessibility. The visibility or accessibility means that you can use the item from a given place.

A method parameter is visible inside of the entire method but not visible outside the method

```
public class Scope {

    public void method1(int i) { // parameter i is accessible only from inside this code block {...}
        i = i++; // parameter i is accessible
        method2(); // parameter i is accessible as argument but not from inside method2
        int j = i * 2; // parameter i is accessible
    } // from here, parameter i is no longer accessible

    public void method2() {
        int k = 20;
    }

    public static void main(String[] args) {
        method1(10);
    }
}
```

A local variable is visible after its declaration until the end of the block in which the local variable has been created.

```
{
...
// myNumber is NOT visible
{
    // myNumber is NOT visible
    int myNumber;
    // myNumber is visible
    {
        ...
        // myNumber is visible
    }
    // myNumber is visible
}
```

```
// myNumber is NOT visible  
...  
}
```

What you should remember from this chapter

- What is the syntax for main conditional and loop statements (if, while, for, for-each)?
- What is the difference between static and non-static variables and attributes?
- What is a constructor?
- When 'new' shall be invoked?
- What is the difference between an object and a class?
- How visibility works in Java?
- What is the terminology in object oriented programming?

Chapter 6

Problem: managing time series

6.1 Problem statement

The aim of this project is to process random series of data representing sensor data. Outliers have to be detected. Times are expressed in hours with decimals. Series of data are not regularly distributed but covers 24 hours. Here, we will use relative time: it is worth 0 at the beginning of the day.

Outliers

Contextual outliers are defined as data points, which, in the context of previous and future data points, seem highly improbable. These points have been detected as the ones that follow the equations simultaneously:

$$\begin{aligned} \text{pdiff}_k &= x_k - x_{k-1} \\ \text{fdiff}_k &= x_{k+1} - x_k \\ |\text{pdiff}_k| &> m_{\Delta x} + \lambda \sigma_{\Delta x} \\ |\text{fdiff}_k| &> m_{\Delta x} + \lambda \sigma_{\Delta x} \\ \text{pdiff}_k \cdot \text{fdiff}_k &< 0 \end{aligned}$$

where:

- x_k : value of the feature at time quantum k
- pdiff_k : difference between the current and previous data point
- fdiff_k : difference between the current and next data point
- $m_{\Delta x}$: average difference between two consecutive data points for the whole dataset
- $\sigma_{\Delta x}$: standard deviation of the difference between two consecutive data points for the whole dataset
- λ : configurable parameter, here $\lambda = 1$

| All the data points that satisfy the above equation are outliers.

It is requested to:

1. create a class 'Serie' that collects doubles representing either relative times in hours or measurement data. Each serie should be named and must contain methods to return its name, its size and its values as 'double[]'
 2. design a 'toString()' method to display the series
 3. with 'Serie', generate a random serie of 100 (start by 10 for debugging) consecutive values of CO2 between 400 and 2500 using a method `void` → `populateWithValues(int numberOfValues, double min, double max)` for populating the serie.
 4. with 'Serie', generate a non equally distributed serie of 100 (start by 10 for debugging) consecutive relative times in hours covering 24 hours using a method `void` `populateWithTimes(int numberOfValues, double max)` for populating the serie.
 5. add methods to compute average and standard deviation.
- optional detect the outliers from the random data. If no outliers is found, reduce the value of λ for testing.

6.2 Clues

- `Math.random()` generates a random number between 0 and 1.
- `Math.abs(doubleValue)` computes the absolute value of `doubleValue`.
- use the method `set` to replace a value in an `ArrayList`.
- to sort an `ArrayList<T>`, you can use the method `sort` bound to the class `java`. → `util.Collections`.
- to copy the elements of an `ArrayList<T>` of type `T`, use the method `toArray()` → `T[] arrayWhereDataAreCopied`

Part II

Object Oriented Programming: 8 hours + personal work

Chapter 7

Object oriented programming

7.1 Deeper into the object oriented paradigm

We have seen that Java classes can be used to create new types. Element of this new type is a compound value named instance or object.

Each object is created thanks to a specific method, named constructor, holding the same name that the class depicting the object, whose return type is implicitly 'void' and omitted.

The 'static' keyword is used to bind a variable or a method to the class i.e. to the program. When the keyword is omitted, it means that the variable or the method is related a particular object of the related class.

Consider the following code:

```
public class Class {  
  
    static int variable = 0;  
    String attribute1;  
    double attribute2;  
  
    Class(String anAttribute1, double anAttribute2, int aVariable) {  
        attribute1 = anAttribute1;  
        attribute2 = anAttribute2;  
        variable = aVariable;  
    }  
  
    public String toString() {  
        return attribute1 + " , " + attribute2 + " , " + variable;  
    }  
}
```

```

public static void main(String[] args) {
    Class object1 = new Class("object1", 1, 1); // a new object of type Class is constructed
    System.out.println(object1);
    Class object2 = new Class("object2", 2, 2); // a new object of type Class is constructed
    System.out.println(object1);
    System.out.println(object2);
}
}

```

The result is:

```

object1, 1.0, 1
object1, 1.0, 2
object2, 2.0, 2

```

The second time ‘object1’ is displayed, the 3rd value has changed because of the construction of ‘object2’! Why? Because a static variable is bound to the class that generates objects i.e. it is common to all objects stemmed from ‘Class’. Assignment of value 2 during the generation of ‘object2’ modifies the value in all objects whereas each non-static variable is independent (but same types) in each object.

Therefore, we can imagine two kinds on variable containers: one, which contains static variables related to a class, and several variable containers, each of them bound to a instance of the class. Actually, the containers related to an instance can be represented by the keyword this. It is not really representative of the meaning. It is cleared in Python language where the keyword ‘self’ is used.

In the previous code, ‘this’ where implicit. Let’s explicit the code:

```

public class Class {

    static int variable = 0;
    String attribute1;
    double attribute2;

    Class(String anAttribute1, double anAttribute2, int aVariable) {
        this.attribute1 = anAttribute1;
        this.attribute2 = anAttribute2;
        Class.variable = aVariable;
    }

    public String toString() {
        return this.attribute1 + ", " + this.attribute2 + ", " + Class.variable;
    }

    public static void main(String[] args) {
        Class object1 = new Class("object1", 1, 1); // a new object of type Class is constructed
        System.out.println(object1);
    }
}

```

```

Class object2 = new Class("object2", 2, 2); // a new object of type Class is constructed
System.out.println(object1);
System.out.println(object2);
}
}

```

Distinguishing local from object variables

Up to now, we used different names for local and object variables, for instance ‘attribute1’ and ‘anAttribute1’. Indeed, if the same name would have been used, it would not be possible to distinguish local from object variables. However, it is tedious to name differently the same quantity. Therefore, Java developer uses also ‘this’ to distinguish both types of variables:

```

public class Class {

    static int variable = 0;
    String attribute1;
    double attribute2;

    Class(String attribute1, double attribute2, int variable) {
        this.attribute1 = attribute1;
        this.attribute2 = attribute2;
        Class.variable = variable;
    }

    public String toString() {
        return this.attribute1 + ", " + this.attribute2 + ", " + Class.variable;
    }

    public static void main(String[] args) {
        Class object1 = new Class("object1", 1, 1); // a new object of type Class is →
        ↪ constructed
        System.out.println(object1);
        Class object2 = new Class("object2", 2, 2); // a new object of type Class is →
        ↪ constructed
        System.out.println(object1);
        System.out.println(object2);
    }
}

```

Exercise 7.1

Develop a single class ‘Customer’ with attributes ‘name’ (String) and ‘credit’ (double) with a counter with the number of customer generated. Try and check whether it works properly.

Back to primitive data types and objects

Now that we went deeper into object oriented programming, we can reexamine the correspondence between primitive data types and objects.

- byte → Byte: <http://docs.oracle.com/javase/7/docs/api/java/lang/Byte.html>
- short → Short: <http://docs.oracle.com/javase/7/docs/api/java/lang/Short.html>
- int → Integer: <http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html>
- long → Long: <http://docs.oracle.com/javase/7/docs/api/java/lang/Long.html>
- float → Float: <http://docs.oracle.com/javase/7/docs/api/java/lang/Float.html>
- double → Double: <http://docs.oracle.com/javase/7/docs/api/java/lang/Double.html>
- char → Character: <http://docs.oracle.com/javase/7/docs/api/java/lang/Character.html>
- boolean → Boolean: <http://docs.oracle.com/javase/7/docs/api/java/lang/Boolean.html>

A primitive data type uses a small amount of memory to represent a single item of data whereas a Java object is a large chunk of memory that can potentially contain a great deal of data along with methods. Objects corresponding to primitive data types contain a large set of methods related to the type. Links to the official javadoc are given above. Check what are the methods available.

Additionally, the String class (see <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>) is interesting to study: it contains a large set of useful methods.

Java has been designed for object oriented programming. Practically, it means that ‘static’ variables and methods must remain exceptional. Processings result from the interactions of new types reflecting actual objects or concepts. Each object of a given class has its own values and its methods that operate on its own value. As mentioned before, it is a lot more efficient for complex codes because objects copy the reality and are therefore meaningful. Their attributes, i.e. object variables, define their characteristics and their methods, their capabilities, i.e. the services they can provide. Consequently, data and functions are encapsulated into objects copying a meaningful reality: it is highly structured according to a representation of a world. Object oriented design is therefore not sequential just like in structured programming such as C, but it is highly dependent of a representation of a world seen by the developer.

Exercise 7.2

The aim is to represent broken lines and to compute their length. With an object oriented point of view, 2 classes should be created:

- a class ‘Point’ with x and y coordinates, and a method ‘double distance(Point otherPoint)’
- a class ‘BrokenLine’ with a serie of points and a method ‘double length()’

Compute the length of the following broken line: ((0,0), (2,0), (2,2), (1,3), (0,2), (0,0)).

You can get further informations about ‘class’ and ‘object’ at https://en.wikibooks.org/wiki/Java_Programming/Defining_Classes and <https://en.wikibooks.org/wiki/>

[Java_Programming/Object_Lifecycle](#).

7.2 Polymorphism

In Java, it is possible to have different methods holding the same name. Consider for instance the code:

```
public class Rectangle {  
  
    double width, height;  
  
    Rectangle() {}  
  
    void setSize(double width, double height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    void setSize(double length) {  
        this.width = length;  
        this.height = length;  
    }  
  
    public String toString() {  
        return this.width + " x " + this.height;  
    }  
  
    public static void main(String[] args) {  
        Rectangle rectangle = new Rectangle();  
        rectangle.setSize(3, 4);  
        System.out.println(rectangle);  
        rectangle.setSize(5);  
        System.out.println(rectangle);  
    }  
}
```

It comes out:

```
3.0 x 4.0  
5.0 x 5.0
```

Their 2 methods 'setSize' offering different forms of a same service 'setSize': one requires to provide 2 values and another one, only one length assuming the rectangle is a square. Even if the name is identical, Java virtual machine can make the difference using the method signatures:

```
void setSize(double, double)
```

```
void setSize(double)
```

Only the signature is important: names of the arguments do not make any difference. This Java facility is named ‘polymorphism’ because a same service i.e. same method name, can adapt to different set of arguments providing thus different forms for the service.

Similarly, there may be several ways to create an object: polymorphisms also exist for constructor:

```
public class SimpleRectangle {  
  
    double width, height;  
  
    SimpleRectangle(double width, double height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    SimpleRectangle(double length) {  
        this.width = length;  
        this.height = length;  
    }  
  
    public String toString() {  
        return this.width + " x " + this.height;  
    }  
  
    public static void main(String[] args) {  
        SimpleRectangle rectangle1 = new SimpleRectangle(3, 4);  
        SimpleRectangle rectangle2 = new SimpleRectangle(5);  
        System.out.println(rectangle1);  
        System.out.println(rectangle2);  
    }  
}
```

To avoid redundant code, it is sometime a good option to call another constructor from a current one. It is done thanks to the ‘this()’ method, which as no relationship with the ‘this’ keyword representing the object attribute container:

```
public class SimpleRectangle {  
  
    double width, height;  
  
    SimpleRectangle(double width, double height) {  
        this.width = width;  
        this.height = height;  
    }  
}
```

```
SimpleRectangle(double length) {  
    this(length, length);  
}  
  
public String toString() {  
    return this.width + " x " + this.height;  
}  
  
public static void main(String[] args) {  
    SimpleRectangle rectangle1 = new SimpleRectangle(3, 4);  
    SimpleRectangle rectangle2 = new SimpleRectangle(5);  
    System.out.println(rectangle1);  
    System.out.println(rectangle2);  
}  
}
```

7.3 Encapsulation

Experience of best practices in Java programming lead developers not to access directly to variables encapsulated into objects but to use assessors i.e. dedicated methods named setters to modify the values of attributes and getters to get the values of attributes. Indeed, it yields a better control of attributes for future developments. For instance, you have to trigger a process is an attribute value is updated. If a setter is used, it just consists in adding instructions in the setter. This practice is widely used in Java programming.

The following scheme illustrates this principle:

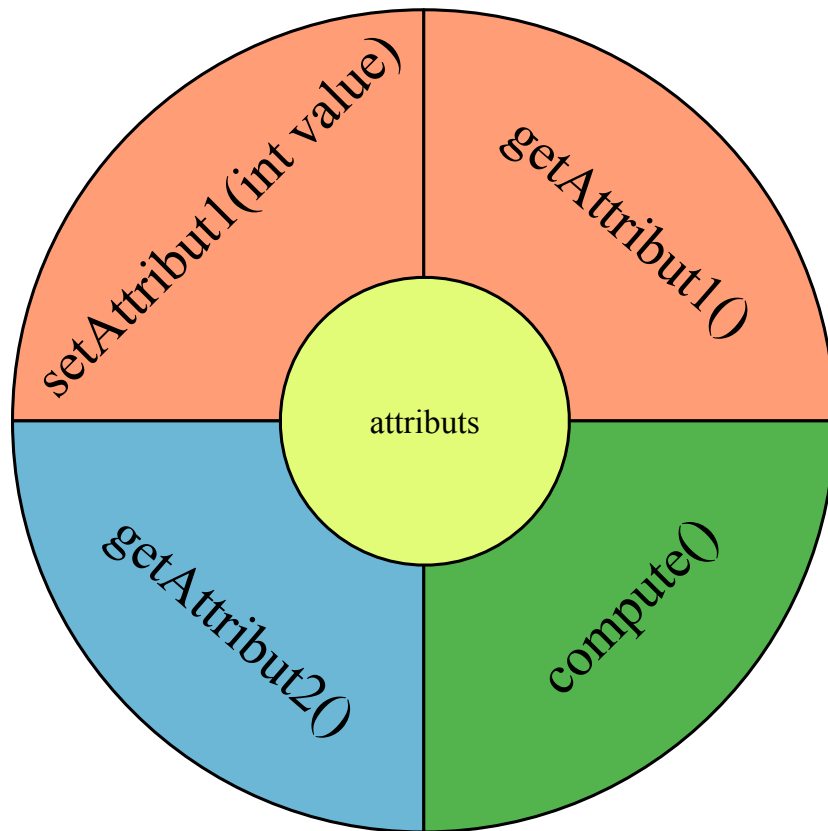


Figure 7.1 Encapsulation of attributes

Here is an example:

```
public class EncapTest{  
  
    private String name;  
    private String idNum;  
    private int age;  
  
    public int getAge(){  
        return age;  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public String getIdNum(){  
        return idNum;  
    }  
  
    public void setAge( int newAge){  
        age = newAge;  
    }  
}
```

```
public void setName(String newName){
    name = newName;
}

public void setIdNum(String newId){
    idNum = newId;
}
}
```

Then, attributes are accessed in this way:

```
public class RunEncap{

    public static void main(String args[]){
        EncapTest encap = new EncapTest();
        encap.setName("James");
        encap.setAge(20);
        encap.setIdNum("12343ms");

        System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());
    }
}
```

What you should remember from this chapter?

- What is the difference between ‘class’ and ‘object’?
- What is the difference between a static variable and a non-static one?
- What is the difference between a static method and a non-static one?
- What is the difference between ‘this’ and ‘this()’?
- What ‘polymorphism’ stands for?
- What are ‘getter’ and ‘setter’?

Chapter 8

Core object oriented programming

8.1 Inheritance and overloading

The inheritance is a powerful mechanism of the object oriented programming. It allows the reuse of the members (attributes and methods) of a class, called superclass or parent class, in another class, called subclass, child class or derived class. Let's consider the classes 'Plane' and 'Car':

```
public class Plane {  
  
    double coveredDistanceInKm; // in common with Car  
    int numberOfEngines;  
  
    public Plane(int numberOfEngines, double initialCoveredDistanceInKm) {  
        coveredDistanceInKm = initialCoveredDistanceInKm; // in common with Car  
        this.numberOfEngines = numberOfEngines;  
    }  
  
    void addCoveredDistanceInKm(double coveredDistanceInKm) { // in common with Car  
        this.coveredDistanceInKm = this.coveredDistanceInKm + coveredDistanceInKm;  
    }  
  
    double getCoveredDistanceInKm() { // in common with Car  
        return coveredDistanceInKm;  
    }  
  
    public String toString() {  
        return "Plane with "+numberOfEngines+" engines. Covered distance →  
        ↪ is "+coveredDistanceInKm;  
    }  
}
```

```

    }

    and
    public class Car {

        double coveredDistanceInKm; // in common with Plane
        int numberOfGears;
        String owner;

        public Car(int numberOfGears, double initialCoveredDistanceInKm) {
            coveredDistanceInKm = initialCoveredDistanceInKm; // in common with Plane
            this.numberOfGears = numberOfGears;
            owner = " ";
        }

        void addCoveredDistanceInKm(double coveredDistanceInKm) { // in common with Plane
            this.coveredDistanceInKm = this.coveredDistanceInKm + coveredDistanceInKm;
        }

        double getCoveredDistanceInKm() { // in common with Plane
            return coveredDistanceInKm;
        }

        void setOwner(String owner) {
            this.owner = owner;
        }

        public String toString() {
            if (owner.length()==0)
                return "Car with "+numberOfGears+" gears. Covered distance is "+ →
                ↪ coveredDistanceInKm;
            else
                return "Car owned by "+owner+" with "+numberOfGears+" gears. →
                ↪ Covered distance is "+coveredDistanceInKm;
        }

        public static void main(String[] args) {
            Plane plane = new Plane(4, 12000);
            plane.addCoveredDistanceInKm(1000);
            Car car = new Car(5,5000);
            car.setOwner("Paul");
            car.addCoveredDistanceInKm(1000);
            System.out.println(plane); // even if car and plane have common attributes and methods
            System.out.println(car); // they cannot be handled is a common way.
        }
    }
}

```

Running car leads to:

Plane with 4 engines. Covered distance is 13000.0

Car owned by Paul with 5 gears. Covered distance is 6000.0

You can notice duplicated codes, which are difficult to maintain. Indeed, if you want to update one class, then you have also to update the other class. Then, discrepancies main appears. There is also another issue: although there are different, both types (classes) of objects share some common attributes and methods but they cannot be handled in the same way, in a for-loop for instance that would display the current covered distance with `getCoveredDistanceInKm()`.

The common parts of the code can be factorized in a superclass 'Vehicle':

```
class Vehicle {

    double coveredDistanceInKm;

    Vehicle(double initialCoveredDistanceInKm) {
        coveredDistanceInKm = initialCoveredDistanceInKm;
    }

    void addCoveredDistanceInKm(double coveredDistanceInKm) {
        this.coveredDistanceInKm = this.coveredDistanceInKm + coveredDistanceInKm;
    }

    double getCoveredDistanceInKm() {
        return coveredDistanceInKm;
    }

}
```

'Plane' and 'Car' then become 'BetterPlane' and 'BetterCar', where common attributes and methods do not appear but are inherited from 'Vehicle'. Here is the 'BetterPlane' class:

```
public class BetterPlane extends Vehicle { // note the `extends Vehicle'

    int numberOfEngines;

    public BetterPlane(int numberOfEngines, double initialCoveredDistanceInKm) {
        super(initialCoveredDistanceInKm); // note that this line must be the first line of the →
        ↪ constructor in case of inheritance
        this.numberOfEngines = numberOfEngines;
    }

    public String toString() {
        return "Plane with " + numberOfEngines + " engines. Covered distance →
        ↪ is " + coveredDistanceInKm;
    }

}
```

Even if it is not obvious when looking at 'BetterCar' code, an object of class 'BetterCar' will have the following attributes:

- coveredDistanceInKm
- numberOfEngines

and the following methods:

- addCoveredDistanceInKm(double)
- getCoveredDistanceInKm()
- toString()

When an 'BetterPlane' object is created using 'new', the first line to be executed is `super(initialCoveredDistanceInKm)`. It invokes the constructor of the superclass 'Vehicle' with the argument `initialCoveredDistanceInKm`. Indeed, through inheritance mechanism, a plane is a specialization of a vehicle, it means that the vehicle has to be built first then specialized into a vehicle. What happened then when calling `new BetterPlane` → (4, 12000)? Following steps are carried out:

- the program cursor reaches the first line of the constructor of 'BestPlane' with the local variables `numberOfEngines=4` and `initialCoveredDistanceInKm=12000`
- `super(initialCoveredDistanceInKm)`; calls the constructor of 'Vehicle' with the local variable `initialCoveredDistanceInKm=12000`
- the attributes of the Vehicle object under creation are created i.e. the attribute 'coveredDistanceInKm' here. It is bound to the 'this' object variable container
- the 'Vehicle' constructor is then run: it modifies the `coveredDistanceInKm` attribute value with `coveredDistanceInKm = initialCoveredDistanceInKm`; (`initialCoveredDistanceInKm=12000`)
- then, it returns to 'BestPlane' and initializes its attributes: `numberOfEngines`. It is set to 0.
- then it runs the instruction after the 'super()' of the 'BestPlane' constructor i.e. `this.numberOfEngines = numberOfEngines`; Yet, the attributes of 'BestPlane' `numberOfEngines` is worth 4.
- an object of class 'BestPlane' is now available. It has 2 attributes and 3 methods available

The code of 'BetterCar' follows. It also inherits from 'Vehicle':

```
public class BetterCar extends Vehicle {

    int numberOfGears;
    String owner;

    public BetterCar(int numberOfGears, double initialCoveredDistanceInKm) {
        super(initialCoveredDistanceInKm);
        this.numberOfGears = numberOfGears;
        owner = "";
    }
}
```

```

    }

    void setOwner(String owner) {
        this.owner = owner;
    }

    public String toString() {
        if (owner.length()==0)
            return "Car with "+numberOfGears+" gears. Covered distance is "+ →
            ↪ coveredDistanceInKm;
        else
            return "Car owned by "+owner+" with "+numberOfGears+" gears. →
            ↪ Covered distance is "+coveredDistanceInKm;
    }

    public static void main(String[] args) {
        Vehicle[] fleet = new Vehicle[] {new BetterPlane(4, 12000), new BetterCar(5,5000)};
        ((BetterCar)fleet[1]).setOwner("Paul"); // cast is required because setOwner is specific to →
        ↪ BetterCar
        for(Vehicle vehicle: fleet) { // cars and planes are handled in the same way
            vehicle.addCoveredDistanceInKm(1000);
            System.out.println(vehicle);
        }
    }
}

```

It leads to:

Plane with 4 engines. Covered distance is 13000.0

Car owned by Paul with 5 gears. Covered distance is 6000.0

If you look at the first line of the main method `Vehicle[] fleet = new Vehicle[] {new → ↪ BetterPlane(4, 12000), new BetterCar(5,5000)}`, you can see that because `BestPlane` and `BestCar` objects are `Vehicle`, they can be combined in a same array of `Vehicles`: `fleet`. The `for`-each loop can directly used the attributes and methods related to `Vehicle` but not directly members that are specific to subclasses '`BetterPlane`' or '`BetterCar`'. To do so, just like in `((BetterCar)fleet[1]).setOwner("Paul")`, the object has to be casted into the class where the method is bound i.e. '`BetterCar`' for method `setOwner(String)`.

From a semantic point of view, it is said that '`BestPlane`' and '`BestCar`' are specialization of '`Vehicle`'. '`Vehicle`' is a generalization of '`BestPlane`' and '`BestCar`'. In this relation, '`BestPlane`' and '`BestCar`' are subclasses of '`Vehicle`'. '`Vehicle`' is the superclass of '`BestPlane`' and of '`BestCar`'. Therefore, the technical inheritance mechanism can be also mapped to the objects of the world.

The relation between classes i.e. new type, can be represented by the following UML class diagram:

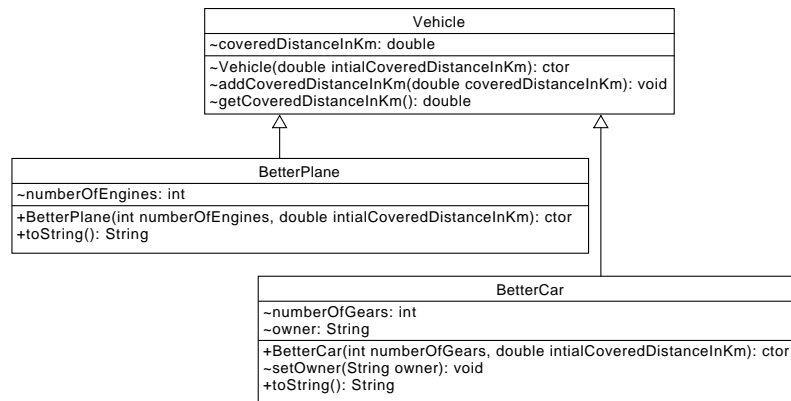


Figure 8.1 UML class diagram of the inheritance

Exercise 8.1

Run the previous code in debug mode after putting a break point at line `Vehicle →`
`↪ [] fleet = new Vehicle[] { new BetterPlane(4, 12000), new BetterCar(5,5000) }`. Observe how the program cursor evolves using ‘step into’ and ‘step over’ buttons.

Imagine now that you want to count the number of trips achieved but only for the cars. You need to intercept the call of the ‘Vehicle’ method named `addCoveredDistanceInKm →`
`↪ (double)`. To do that, we are going to overload the ‘Vehicle’ method in ‘BestCar’:
`public class BetterCar extends Vehicle {`

```

    int numberOfGears, numberOfTrips;
    String owner;

```

```

    public BetterCar(int numberOfGears, double initialCoveredDistanceInKm) {
        super(initialCoveredDistanceInKm);
        this.numberOfGears = numberOfGears;
        owner = "";
        numberOfTrips = 0;
    }

```

```

    void setOwner(String owner) {
        this.owner = owner;
    }

```

```

    public String toString() {
        if (owner.length() == 0)
            return "Car with " + numberOfGears + " gears. Covered distance is " + ↪
            ↪ coveredDistanceInKm + "; trips: " + numberOfTrips;
        else
            return "Car owned by " + owner + " with " + numberOfGears + " gears. ↪
            ↪ Covered distance is " + coveredDistanceInKm + "; trips: " + ↪
            ↪ numberOfTrips;
    }

```

```

    }

    void addCoveredDistanceInKm(double coveredDistanceInKm) {
        numberOfTrips += 1; // specific processing (increment by 1)
        super.addCoveredDistanceInKm(coveredDistanceInKm); // the superclass method is then →
        ↪ called using super
    }

    public static void main(String[] args) {
        BetterPlane plane1 = new BetterPlane(4, 12000);
        BetterCar car1 = new BetterCar(5,5000);
        Vehicle[] fleet = new Vehicle[] { plane1, car1 };
        car1.setOwner("Paul"); // cast is required because setOwner is specific to BetterCar
        for(Vehicle vehicle: fleet) { // cars and planes are handled in the same way
            vehicle.addCoveredDistanceInKm(1000);
            System.out.println(vehicle);
        }
        car1.addCoveredDistanceInKm(700);
        System.out.println(car1);
    }
}

```

It yields:

Plane with 4 engines. Convered distance is 13000.0
 Car owned by Paul with 5 gears. Covered distance is 6000.0; trips:1
 Car owned by Paul with 5 gears. Covered distance is 6700.0; trips:2

The method `void addCoveredDistanceInKm(double)` in ‘BestCar’ overloads the method in ‘Vehicle’ because it has the same signature (same name, same type of arguments and same type of return value). It acts as a replacement of the superclass method. Nevertheless, it is possible to redirect to the superclass method using the keyword ‘super’ in reference to the superclass i.e. ‘Vehicle’.

Casting objects

A subclass can be assigned directly into its superclass (and super-superclass) without any cast but the opposite is not that easy. Cast has to be explicitly done. Let’s consider the following code architecture:

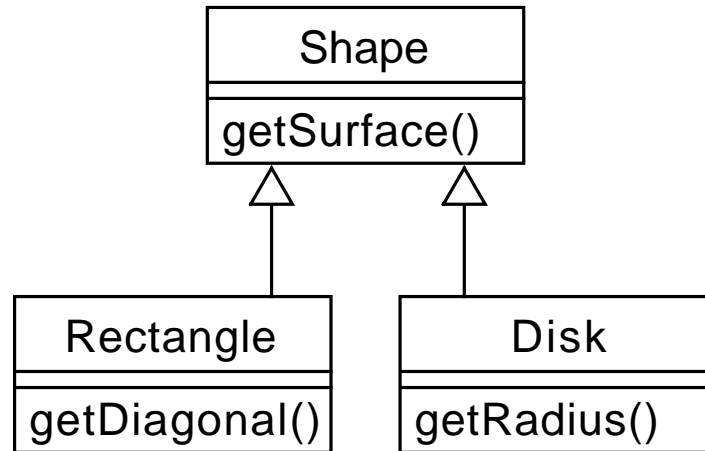


Figure 8.2 Example of architecture

Analyze these lines of code:

```

Shape rectangle1 = new Rectangle(2, 3); // #1 OK rectangle1 is a rectangle embedded →
    → into a Shape object
double shape1 = rectangle1.getSurface(); // #2 OK because getSurface() is bound to a →
    → Shape object
double diagonal = rectangle1.getDiagonal(); // #3 compilation error because getDiagonal →
    → () is bound to Rectangle and rectangle1 looks now as a shape
Rectangle rectangle2 = rectangle1; // #4 compilation error because rectangle1 looks as a →
    → shape: an explicit cast has to be done when putting a superclass object into a →
    → subclass object
Rectangle rectangle3 = (Rectangle)rectangle1; // #5 OK solve the issue of #4
double diagonal = rectangle3.getDiagonal(); // #6 OK because rectangle3 is a rectangle →
    → and getDiagonal() is bound to Rectangle
Disk disk = (Disk)rectangle1; // #7 execution error: rectangle1 is a Rectangle embedded →
    → into a shape. At runtime, when casting it to a disk, it throws an error because it is →
    → not. It just can be detected at runtime.
  
```

To learn more about inheritance, go to <https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html>.

Last topic to develop in this section is the order of execution. We have already seen how the execution order in non-sequential. Let's go deeper into this important topic. Consider a superclass with static and non static variables:

```

public class SuperClass {
    static int var1=1;    // #1
    int var2=2;          // #2
    static {
        code1;          // #3
    }
}
  
```

```
public SuperClass() {  
    code2;           // #4  
}  
}
```

All the static variables but also the static block (which acts as a constructor for the class) are initialized and run at first when the Java application under development starts to run.

```
public class SubClass extends SuperClass {  
    static int var3=1;    // #5  
    int var4=2;           // #6  
    static {  
        code3;           // #7  
    }  
    public SubClass() {  
        super();  
        code4;           // #8  
    }  
    static public void main(String[] args) {  
        new SubClass();  // #9  
    }  
}
```

When running the main method, the instructions are run in the following order: #5, #7, #1, #3, #9, #2, #4, #6 and finally #8. Note that static variables and code blocks are run first however, order in between #5, #7, #1 and #3 is not guarantee but it does not really matter.

All the Java classes inherit from Object

All the Java classes inherit from Object. It is not necessary to add `extends Object` to the top level objects: it is done implicitly. As a consequence, according to the Object javadoc <http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>, all the Java objects inherits for instance the methods `public String toString()` and `public → ↪ boolean equals(Object obj)`. It means that even if you do not overload the method in implementing your own, these methods will be existing and will provide a default result.

Some classes can be nested into the class holding the same name than the current file but it has to be avoid as much as possible because of the potential increase of code complexity. Nevertheless, it can be useful in some circumstances. To learn more about nested classes, visit https://en.wikibooks.org/wiki/Java_Programming/Nested_Classes.

8.2 Abstract classes and interfaces

Sometime a class can be incomplete. Reconsider example 8.2. It is not possible to instantiate (generate) objects of type Shape. Indeed, how to compute the surface whereas the type of shape is not specified. It is nevertheless useful to define that all the objects of type and subtypes of shapes will provide a method `getSurface()`. In this kind of situations, the keyword ‘abstract’ should be used to tell a class cannot generate objects directly but only subclasses could do. If a method is declared as abstract, its content is not specified but subclasses will have to do, providing they are not abstract themselves. The class ‘Shape’ could be written:

```
abstract class Shape { // abstract there means the class Shape cannot generates objects

    Shape() {...}

    abstract double getSurface(); // abstract there means that subclasses should implement this →
    ↪ method
}
```

A class ‘Rectangle’ could be written:

```
class Rectangle extends Shape {

    double width;
    double height;


    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    double getDiagonal() {
        return Math.sqrt(width*width + height*height);
    }

    double getSurface() { // Java will throw an error if this method is not implemented because of →
        ↪ the abstract class Shape
        return width * height;
    }

}
```

Another shape likes ‘Disk’ will also implement `double getSurface()` but its own way.

 Inheriting from several superclasses can indeed lead to complex problems when executing the superclass constructors: order is important and it is not just to say the subclass belongs to several supertypes.

In Java multiple inheritance is not allowed for the sake of clarity. However, it may be useful to specify that an object is of several types at the same time. Java provides a specific kind of types: the interfaces. Roughly speaking, an interface is a totally abstract class without constructor. Because all the methods are necessary

abstract, it is not required to declare them as abstract: it is implicit. The main interest of interfaces is that a class may implement several interfaces i.e. with this mechanism, a subtypes can be of different supertypes: it works around the prohibited multi-inheritance.

Consequently, an interface cannot be instantiated. An interface is not a class but it is written the same way. The first difference is that it does not use the class keyword but the interface keyword to define it. Then, there are fields and methods you cannot define here:

- A variable can only be a constant bound to the interface: it is always public, static and final, even if you do not mention it.
- A method can only be public and abstract, but it is not required to write the public and abstract keywords.
- Constructors are forbidden.

An interface represents a contract that all subtypes have to satisfied:

```
public interface SimpleInterface {

    public static final int CONSTANT1 = 1;

    int method1(String parameter);

}
```

☛ Note that it is said 'a class implements an interface' and 'a class inherits from a superclass, possibly abstract'

The method1() is implicitly abstract (unimplemented). To use an interface, a class has to implement it using the implements keyword:

```
public class ClassWithInterface implements SimpleInterface { // several interfaces may be →
    ↪ implemented but only one superclass can be extended

    int method1(String parameter) { // just like for abstract methods, the methods specified in the →
        ↪ interface have to be developed
        return 0;
    }

}
```

A class can implement several interface, separated by a comma. It is recommended to name an interface <verb>able, to mean the type of action this interface would enable on a class.

Let's come back to the shape example. Imagine we want a shape to belong to 2 types at the same time: Shape and ColoredObject. Shape requires 2 methods: getSurface() and getPerimeter(). ColoredObject contains color attributes and methods setColor(int)

and `getColor()`. Because multiple inheritance is not possible, at least one of these types must be an interface. `ShapeInterface` is defined by:

```
public interface ShapeInterface {  
  
    public double getSurface();  
    public double getPerimeter();  
  
}
```

`ColorInterface` can be defined by:

```
public interface ColorInterface {  
  
    public void setColor(int color);  
    public int getColor();  
  
}
```

Because an interface cannot contain attributes, a type ‘`ColoredObject`’ has to be created. It implements `ColorInterface`:

```
public class ColoredObject implements ColorInterface {  
  
    int color = 0;  
  
    public void setColor(int color) {  
        this.color=color;  
    }  
  
    public int getColor() {  
        return color;  
    }  
  
}
```

Therefore, specific shapes can be implemented. For instance:

```
public class Disk extends ColoredObject implements ShapeInterface {...}
```

Object from class `Disk` are also from type `ColoredObject`, `ShapeInterface` and (indirectly) `ColorInterface`.

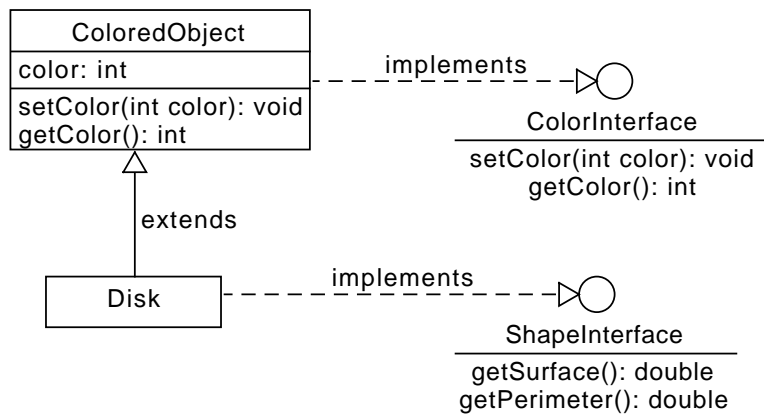


Figure 8.3 UML class diagram of Disk

An interface can extend several interfaces using the ‘extends’ keyword. Multiple inheritance is thus allowed for interfaces:

```
public interface InterfaceA {
    public void methodA();
}
```

```
public interface InterfaceB {
    public void methodB();
}
```

Inheriting from both previous interfaces lead to:

```
public interface InterfaceAB extends InterfaceA, InterfaceB {
    public void otherMethod();
}
```

This way, a class implementing the InterfaceAB interface has to implement the `methodA()`, the `methodB()` and the `otherMethod()` methods:

```
public class ClassAB implements InterfaceAB {
    public void methodA() {
        System.out.println("A");
    }

    public void methodB() {
        System.out.println("B");
    }

    public void otherMethod() {
        System.out.println("foo");
    }
}
```

```

public static void main(String[] args) {
    ClassAB classAb = new ClassAB();
    classAb.methodA();
    classAb.methodB();
    classAb.otherMethod();
}

```

To go deeper into interfaces, visit <https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html>

8.3 Interfaces for structuring codes

Interfaces are also widely used to structure codes. Let's consider the collections and maps we have already studied.

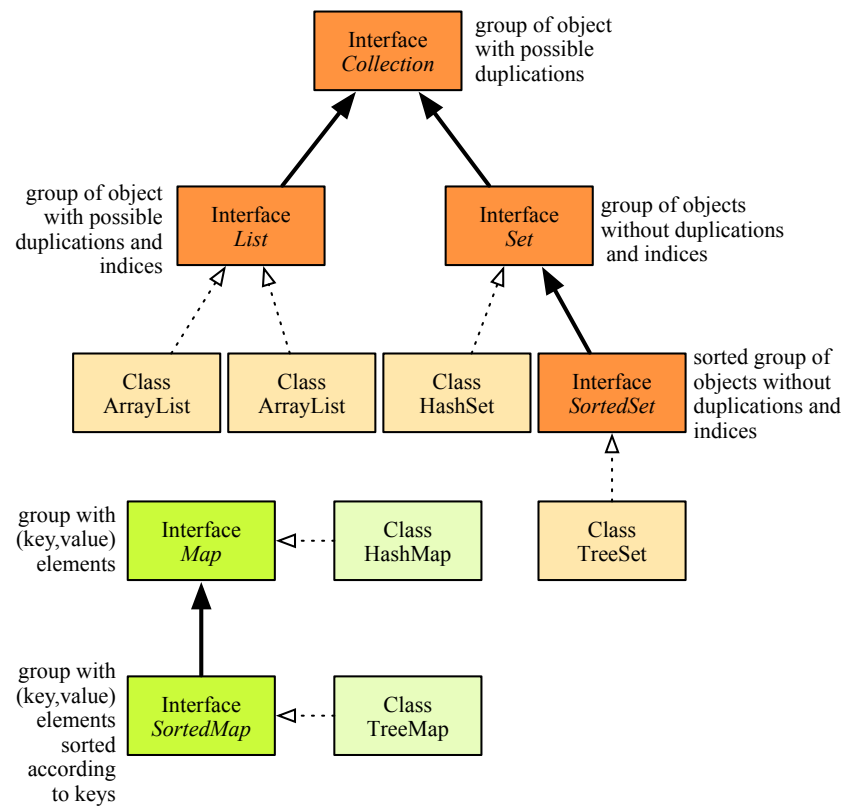


Figure 8.4 Interfaces are used to structure collections and maps

Details about these interfaces and classes can be found at the official Javadoc API:

<http://docs.oracle.com/javase/7/docs/api/>.

Interfaces and implemented classes can be summarized using the UML formalism.

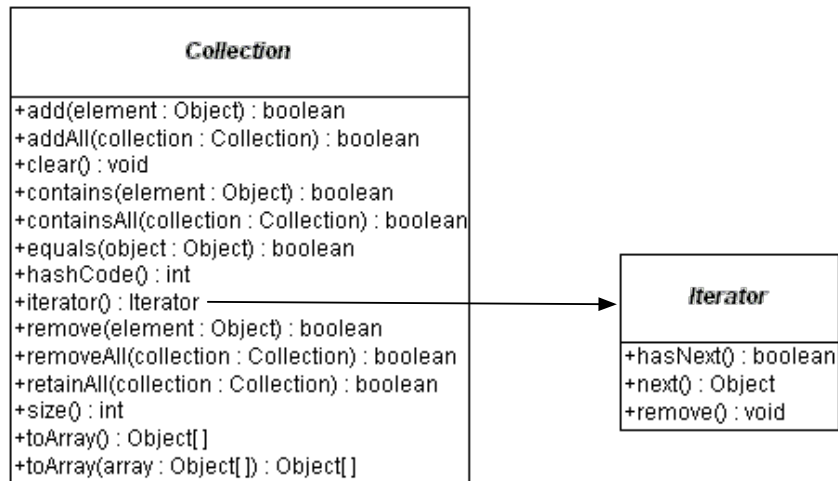


Figure 8.5 Collection and iterator

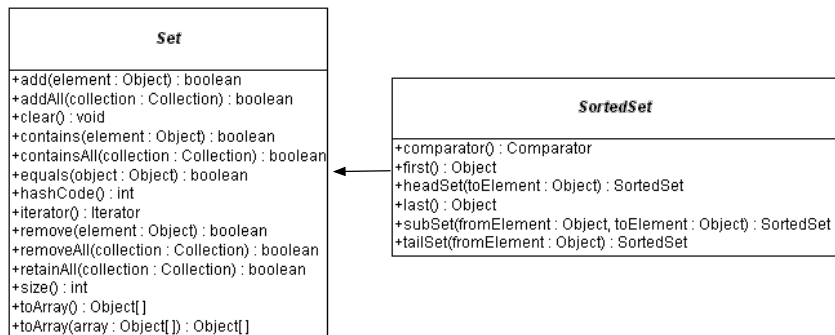


Figure 8.6 Set and SortedSet

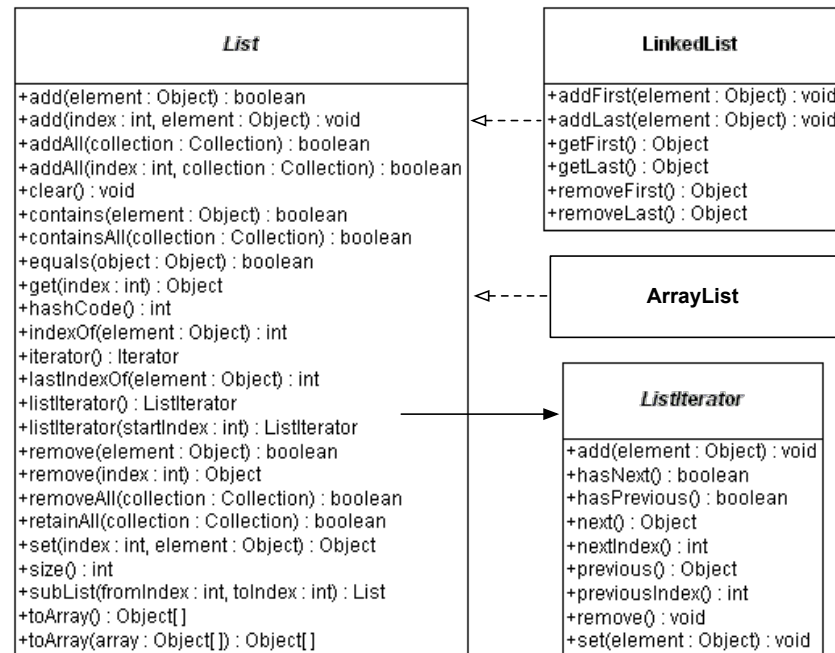


Figure 8.7 List

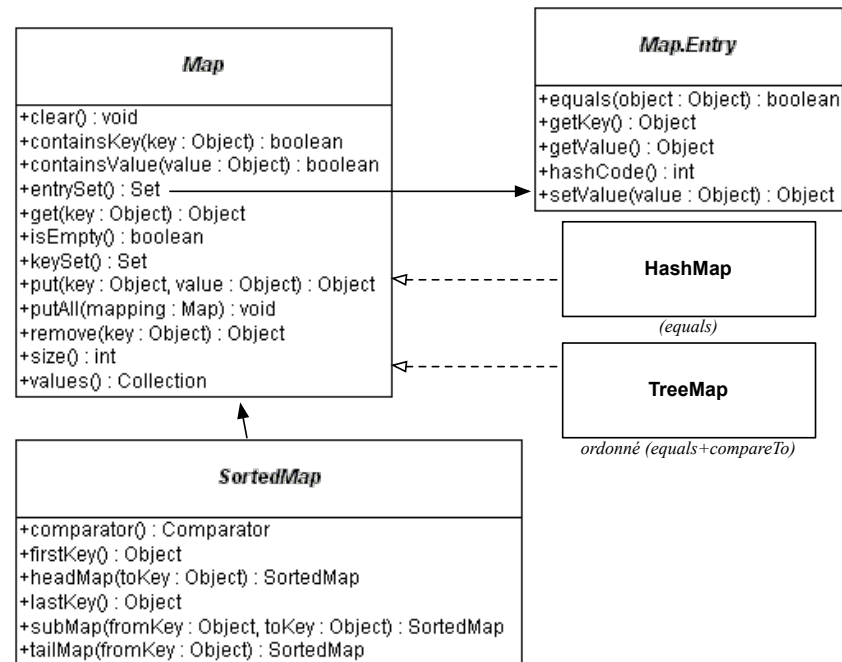


Figure 8.8 Map

8.4 Useful methods to manage arrays and collections

Interfaces are widely used in Java. Consider for instance arrays and collections of objects.

A list or an array can be sorted because of the comparable interface (see <http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html>):

Interface `java.lang.Comparable<T>` {

`int compareTo(T o);`

}

The `compareTo()` method returns a negative integer, zero, or a positive integer as the current object (this) is less than, equal to, or greater than the specified object.

In order to sort objects of type `T` implementing `Comparable`:

an array, use `Arrays.sort(T[] table)`

a list, use `Collections.sort(List<T> list)`

In order to search for an object in a list, use `int Collections.binarySearch(List list, T → object)`. If type `T` does not implement `Comparable`, a comparator implementing `Comparator` (see <http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html>) can be used.

To find out a minimum in a collection, use `Collections.min(Collection collection)` and `Collections.max(Collection collection)`. If objects in collection do not implement `Comparable` → , use `Collections.min(Collection collection, Comparator comparator)` and `Collections.max(→ Collection collection, Comparator comparator)`.

What you should remember from this chapter?

- What is the inheritance mechanism for?
- What is the order of execution when instantiated a subclass of type?
- What is the specialization/generalization of a class?
- How to overload a method? What is it for?
- What is the difference between an abstract class and an interface?
- Why using abstract classes?
- Why using interfaces?

Chapter 9

Exceptions and errors

9.1 First approach

In general, errors can be broken up into two categories: design-time errors and logical errors. Design-time errors are easy to spot because Eclipse usually underlines them. If the error will prevent the program from running, Eclipse will underline it in red. Logical errors are the most difficult to find. The program will run but, because you did something wrong with the coding, there is a good chance the execution crashes.

In Java, errors are handled by an Exception object. Exceptions are said to be thrown, and it is the developer job to catch them. You can do this with a try-catch block. The try-catch block looks like this:

```
try {  
    // code to run with potential exceptions and errors  
} catch (ExceptionType1 exceptionEvent) {  
    // what to do in case of ExceptionType1 has been captured  
} catch (ExceptionType2 exceptionEvent) {  
    // what to do in case of ExceptionType2 has been captured  
}
```

The try part of the try-catch block means "try this code". If something goes wrong, Java will jump to the catch block. It checks the type of exception raised to see if you have handled the error. If you have the corresponding Exception type then the code between the curly brackets will get executed. If you do not have the corresponding Exception type then Java will use its default exception handler to display an error message.

Consider this example:

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

```
try {  
    int x = 10;  
    int y = 0;  
    int z = x / y;  
    System.out.println( z );  
} catch ( Exception err ) {  
    System.out.println( err.getMessage() );  
}  
}
```

It yields:
/ by zero

In the try part of the try-catch block, 3 integers: x, y and z have been defined. We are trying to divide y into x, and then print out the answer.

If anything goes wrong, we have a catch part. In between the round brackets of catch we have this: Exception err.

The type of Exception you are using comes first. In this case we are using the Exception error object. This is a "catch all" type of Exception, and not a very good programming practice.

After your Exception type you have a space then a variable name. We have called it 'err', but any label can be used.

In the curly brackets of catch, we have a print statement. Look what we have between the round brackets of println: err.getMessage(). getMessage() is a method available to Exception objects. As its name suggests, it gets the error message associated with the Exception. When running the code, the displayed message is: / by zero.

Change your code with:

```
double x = 10;  
double y = 0;  
double z = x / y;
```

Again, an error message is displayed in the Output window: Infinity. This time, Java stops the program because the result will be an infinitely large number.

Errors that involve numbers should not really be handled by a "catch all" Exception type. There is a specific type called ArithmeticException. It is better to replace the word Exception between the round brackets of your catch block with ArithmeticException.

9.2 Deeper into exceptions

Exceptions can be caught or propagated. Assume we want to run 2 command lines: hazardousOperation1 and hazardousOperation2. hazardousOperation1 may throw Exception1 or Exception2. hazardousOperation2 may throw Exception2.

As seen before, the exception can be caught locally with:

```
public int method() {
    try {
        hazardousOperation1; //peut générer Exception1 ou Exception2
        hazardousOperation2; //peut générer Exception1
    } catch(Exception1 ex) {
        ex.printStackTrace();
        System.exit(-1);
    } catch(Exception2 ex) {
        processException2;
    } finally {
        processedWhateverHappens;
    }
}
```

However, exception can be propagated to the code block that called method(), which can itself catch the exception or propagate it to its own caller.

To propagate the exceptions without catching, one has to write:

```
public int method() throws Exception1, Exception2 { // propagated exceptions have to be →
    ↪ explicitly declared like here
    hazardousOperation1; //peut générer Exception1 ou Exception2
    hazardousOperation2; //peut générer Exception1
}
```

The following schema shows how an exception is propagated.

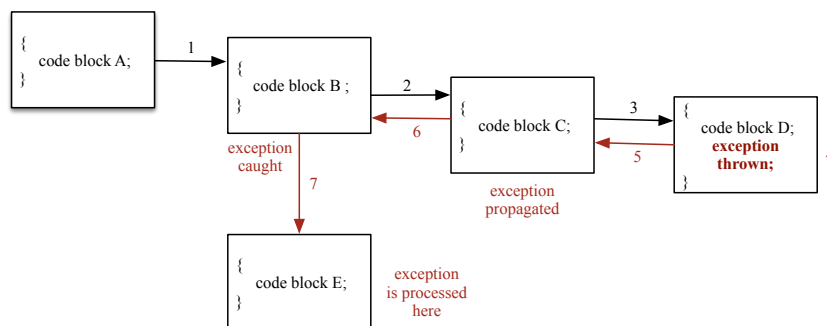


Figure 9.1 How an exception propagates

Let's depict the code sequence:

1. there is an instruction in 'code block A' that calls 'code block B'
2. there is an instruction in 'code block B' that calls 'code block C'
3. there is an instruction in 'code block C' that calls 'code block D'
4. an instruction in 'code block D' throws an exception
5. because the method related to 'code block D' contains a 'throws' statement, the exception is backward propagated to 'code block C'
6. because the method related to 'code block C' contains a 'throws' statement, the exception is backward propagated to 'code block B'
7. 'code block D' does not contain a 'throws' statement but a try-catch block called 'code block E'
8. 'code block E' processes the exception. Depending on 'code block E', the execution can be stopped or it can continue with the next instruction following the one that called 'code block C'

In Java, exceptions are also structured hierarchically:

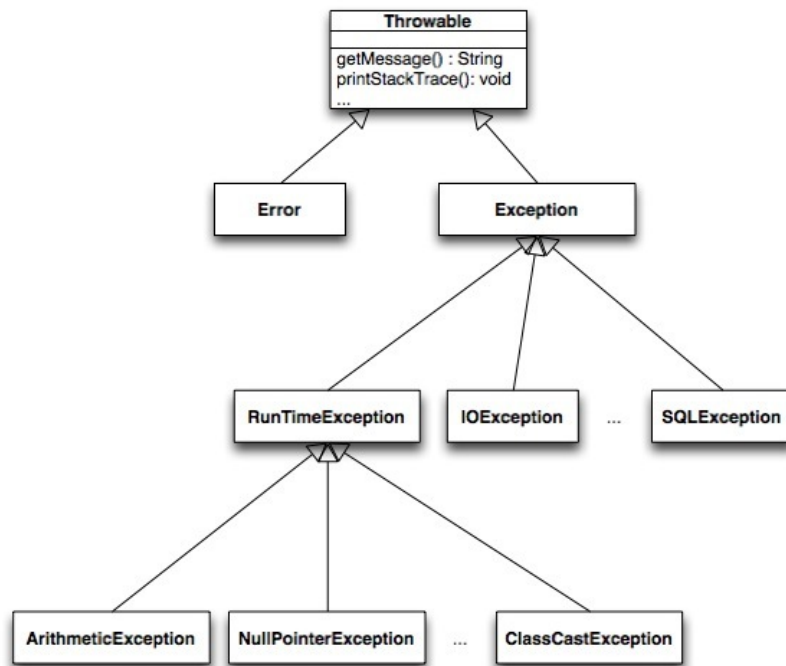


Figure 9.2 Hierarchy of exceptions

`RuntimeException` and its subclasses are special because they do not require any 'throws' statement to be propagated: it is done automatically because they are critical errors that will crash the code execution.

9.3 Raising an Exception

Exception can be created by inheritance. To create a new exception, just create a new class inheriting from an Exception or one of its subclasses:

```
public class MyException extends RuntimeException {
```

```
    public MyException(String message) {
        super(message);
    }
}
```

Because it subclasses `RunTimeException`, `throws` statement will be required in code blocks running instructions that may throw `MyException`.

A code propagating this exception can be:

```
public class MyClass {
```

```
    public MyClass(...) {
        ...;
    }
}
```

```
public void setValue(double value) throws MyException { // the 'throws MyException' →
    ↪ statement is not compulsory here because MyException extends RuntimeException. →
    ↪ Otherwise, it would be.
    if(value > 100)
        throw new MyException("Too large");
    ...;
}
}
```

What you should remember from this chapter?

- What is an exception?
- How to capture an exception?
- How to propagate an exception?
- How to raise an exception?
- What is specific to `RunTimeException`?

Chapter 10

Designing your own object models

10.1 Stick to reality

The best way to design a Java code is to identify the classes of objects that are going to be used. These classes should correspond to concepts of the real world. The first step is then to list these objects and to define how they are linked. Then, attributes and methods are defined for each. Attributes represent their characteristics and methods the services they can provide.

Nevertheless, it is not easy to design when coding. It is usual to split the 2 steps. The design is carried out thanks to a graphical language: the Unified Modeling Language, and in particular, the class diagram.

10.2 Using UML class diagrams as design tool

The purpose of the class diagram is to show the types being modeled within the system. In most UML models these types include 'class' and 'interface'. Note that just types of objects are represented but not directly objects.

The UML representation of a class is a rectangle containing three compartments stacked vertically:

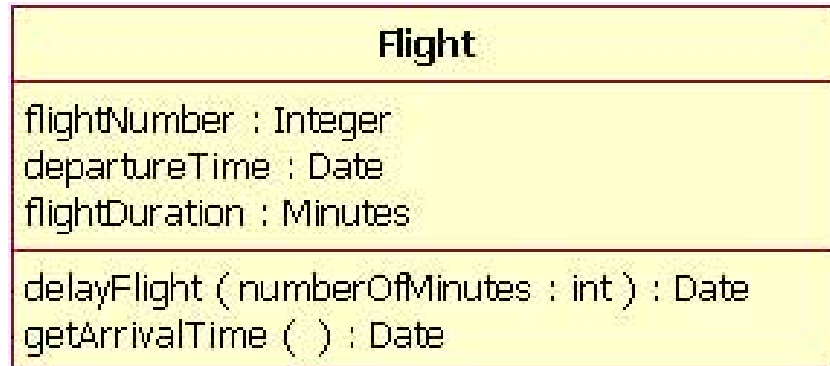


Figure 10.1 Class diagram for the class Flight

The top compartment shows the class's name. The middle compartment lists the class's attributes. The bottom compartment lists the class's operations. When drawing a class element on a class diagram, you must use the top compartment, and the bottom two compartments are optional. Here, the class name is 'Flight', and in the middle compartment, we see that the 'Flight' class has three attributes: `flightNumber`, `departureTime`, and `flightDuration`. In the bottom compartment, we see that the 'Flight' class has two operations: `delayFlight` and `getArrivalTime`.

We have already seen how to represent inheritance. The next figure shows how both `CheckingAccount` and `SavingsAccount` classes inherit from the `BankAccount` class.

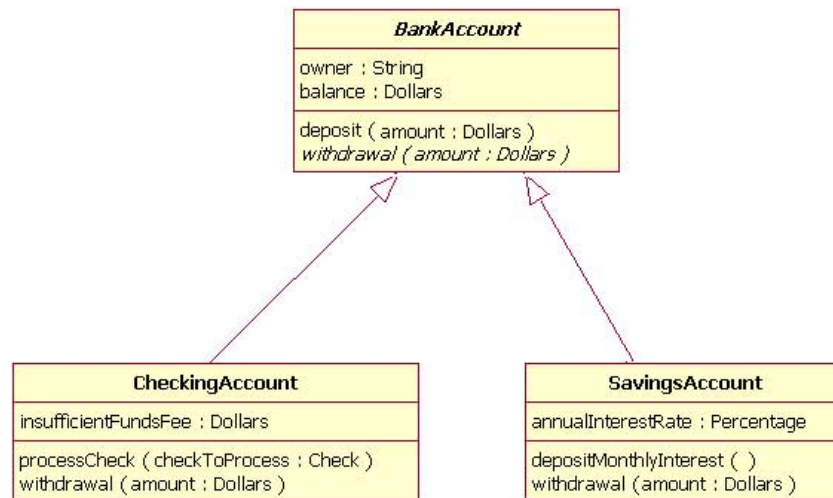


Figure 10.2 Inheritance is indicated by a solid line with a closed, unfilled arrowhead pointing at the super class

The relationships between objects have also to be represented i.e. the variables in an type that refer to another type. Relationships are modelled by associations in UML

class diagrams. The next figure shows a standard kind of association between the Flight class and the Plane class:

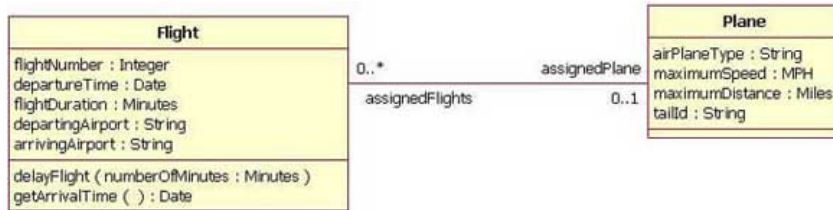


Figure 10.3 A bi-directional association between a Flight class and a Plane class

Literally, it means:

- the class ‘Flight’ contains a reference to 0 or 1 planes. This reference is named ‘assignedPlane’.
- the class ‘Plane’ contains a reference to a collection of flights. This reference is named ‘assignedFlights’.

We can find a multiplicity at each end of an association: min..max where min and max stand respectively for minimum and maximum number of referred objects in the type at the other end of the arrow. ‘*’ means zero or more. With a Java point of view, the association will be translated into:

- an additional variable of type ‘Plane’ into the class ‘Flight’: `Plane assignedPlane → ↩ .`
- an additional array or a list of type ‘Flight’ into the class ‘Plane’: `ArrayList< ↩ ↪ Flight> assignedFlights`, for instance.

Associations can be uni-directional:

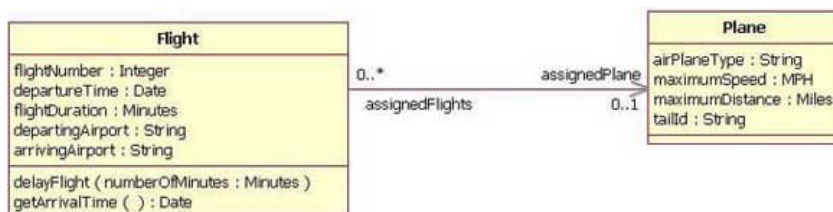


Figure 10.4 An uni-directional association between a Flight class and a Plane class

It means there is not reference to objects of type ‘Flight’ in ‘Plane’.

Bi-directional associations are often used at design stage but, during coding, it is

recommended to transform them into uni-directional associations for the sake of simplicity.

To learn more about UML class diagrams, visit https://en.wikipedia.org/wiki/Class_diagram. For UML in general, visit https://en.wikipedia.org/wiki/Unified_Modeling_Language.

UML class diagram can be drawn on a sheet of paper but software applications can also be used. We recommend UMLet because it is free, multi-OS, simple, powerful and pluggable into Eclipse. See <http://www.umlet.com/faq.htm> to see how to install it into Eclipse.

What you should remember from this chapter?

- How many compartments could be for a class?
- What does an association model?
- What is the difference between a uni-directional and a bi-directional association?
- How an association is translated into Java?
- What a multiplicity? What does it means?

Chapter 11

Problem: Matrices

11.1 Problem statement

Sparse matrices are matrices full of zeros. To avoid filling the memory with a lot of useless zeros, only non-null elements are recorded in a sparse matrix. Nevertheless, from a user point of view, it is hidden. Indeed, if an matrix element is not recorded, it is worth obviously zero. To store the matrix elements, we are going to use a map: `TreeMap<MatrixElementCoordinate, Double>` elements, sorted according to a unique order defined by implementing the Java native `java.lang.Comparable<MatrixElementCoordinate>` \rightarrow \hookrightarrow interface (see javadoc of the official Java api). The class diagram representing of the requested code is below.

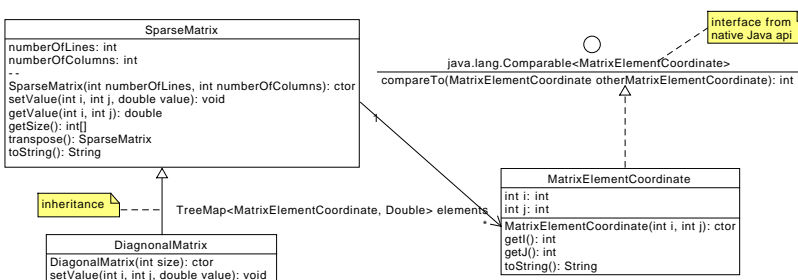


Figure 11.1 UML class diagram of the requested application

1. Develop the classes `SparseMatrix` and `MatrixCoordinate`, according to the UML class diagram, for the following code to run:

```
SparseMatrix sparseMatrix = new SparseMatrix(3,4);
sparseMatrix.setValue(1, 3, -1);
```

```
sparseMatrix.setValue(0, 1, 2);
sparseMatrix.setValue(2, 1, 1);
sparseMatrix.setValue(0, 0, 0);
sparseMatrix.setValue(2, 1, 0);
System.out.println(sparseMatrix);
SparseMatrix transposedMatrix = sparseMatrix.transpose(); // sparseMatrix should not →
    ↪ be altered by this method
System.out.println(transposedMatrix);
System.out.println(sparseMatrix);
```

2. Create your own exception raised when trying to write out of the matrix:
sparseMatrix.setValue(2, 4, -2); should raise the exception
3. Implement the class DiagonalMatrix. It should raise the exception when writing out of the diagonal.
4. Find a solution for the transposition of a diagonal matrix to remain a diagonal matrix (not possible to write out of the diagonal), without reimplementing the transpose method into DiagonalMatrix.

11.2 Clues

Official javadoc `java.lang.Comparable<MatrixElementCoordinate>` for comparable is at <https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html>.

UML class diagram contains a lot of information: just read carefully.

Chapter 12

Problem: Shapes

12.1 Problem statement

The aim is to start the development of an application to plot shapes in a canvas. Each shape is identified by a unique integer value, named 'identifier'. It should be possible to add, remove and get shapes (a single shape when specifying an identifier or all the shapes otherwise, as a array of shapes). It must be possible to print the characteristics of each shape.

A shape can be either a disk or a rectangle defined by points. It should be possible to declare a shape as filled or not.

A disk can be constructed using a central point and a radius, or using 2 points defining its diameter.

A rectangle can be constructed using 2 points defining one of its diagonal, or using one point, a width and a height.

The canvas should provide the 2 following services:

1. check whether a point is inside at least one filled shape in a canvas
2. compute the surface of the filled shapes of a canvas

Here is a sample code that should run:

```
Canvas canvas = new Canvas();  
int rectangle1Id = canvas.addShape(new Rectangle(new Point(1,1),3,2));  
int rectangle2Id = canvas.addShape(new Rectangle(new Point(-1,-1),new Point(-4,-3)));  
int disk1Id = canvas.addShape(new Disk(new Point(-1,1),3));  
int disk2Id = canvas.addShape(new Disk(new Point(1,-3),new Point(2,-2)));
```

```
canvas.getShape(rectangle1Id).setFilling(true);
canvas.getShape(rectangle2Id).setFilling(true);
canvas.getShape(disk1Id).setFilling(true);
canvas.getShape(disk2Id).setFilling(true);
canvas.deleteShape(disk2Id);
System.out.println(canvas);
System.out.println(canvas.getAllFilledShapesArea());
System.out.println(canvas.isInsideAShape(new Point(-2,2)));
System.out.println(canvas.isInsideAShape(new Point(1.5,-2.5)));
```

Develop the application accordingly to object oriented paradigm.

12.2 Clues

Using a 'Plottable' interface is not a bad idea.

`TreeMap<Integer,Shape>` could be useful.

The class `Shape` cannot be instantiated i.e. a `Shape` must be either a disk or a rectangle.

Part III

Project: 12 hours + personal work

Chapter 13

Programming Graphic User Interfaces

Let's examine how GUI¹ can be designed: it will be useful for the project.

13.1 Philosophy of modern GUI toolkits

Modern GUI toolkits are usually structured according to a Model-View-Control (MVC) schema i.e. for each GUI component, there are 3 independent parts:

Model is related to component's data, such as texts, images or properties

View is related to the representation of a components ; it yields the look of a GUI

Control is related to the captures of events coming from users, such as 'left mouse click', 'cursor over a component' or 'key pressed on keyboard'.

Because modern GUI are resizable, the position of GUI visual elements is not static: it depends on layouts, which are related to containers encompassing other GUI elements. The type of layout determines how the visual elements are displayed depending on the size of the container. Therefore, a GUI is organized hierarchically with a top level. A top level container contains other containers or GUI visual components like main window controls (frame), menu bar, toolbar, label, buttons, editor, file picker,... Each component capture some predefined user events. The developer has then to register or to bind a method to some events in order to handle events and to trigger specific behaviors.

¹Graphic User Interfaces

When someone uses a device (mouse, keyboard,...) to interact with an application:

- a material interruption is launched at operating system level: it is modeled by an event with some characteristics (type of event, (x,y) mouse position when click occurred for instance)
- the operating system determines the application, which is concerned by the event, and transmits its data to it, say the Java virtual machine
- if a running application is concerned by the event, the event is redirected to the concerned Java components. If a method has been registered or bound to the kind of captured event, the registered methods are called with the description of the event as argument.

13.2 Java Swing

Swing library is an official Java GUI toolkit released by Sun Microsystems. It is the most widely used to create GUI with Java. It is a lightweight GUI toolkit i.e. GUI visual components are not those provided by the operating system but they are drawn by Java. It is compliant with the MVC paradigm. Swing is built on AWT, which is an old heavyweight toolkit. It is possible to mix component from both toolkits but Swing is more flexible: its appearance can be much more customized. When developing in Swing, it is common to import AWT components for event handling.

☛ Component names between AWT and Swing are quite similar but a 'J' is added for Swing: Panel→JPanel, Frame→JFrame, Button→JButton,...

Accordingly to the modern GUI toolkit philosophy, Java Swing is composed of 3 types of elements:

UI elements i.e. the core elements and containers the user eventually sees and interacts with

layouts: defining how UI elements should be organized on the screen and provide a final look and feel to the GUI

events: handling events which occur when the user interacts with UI elements

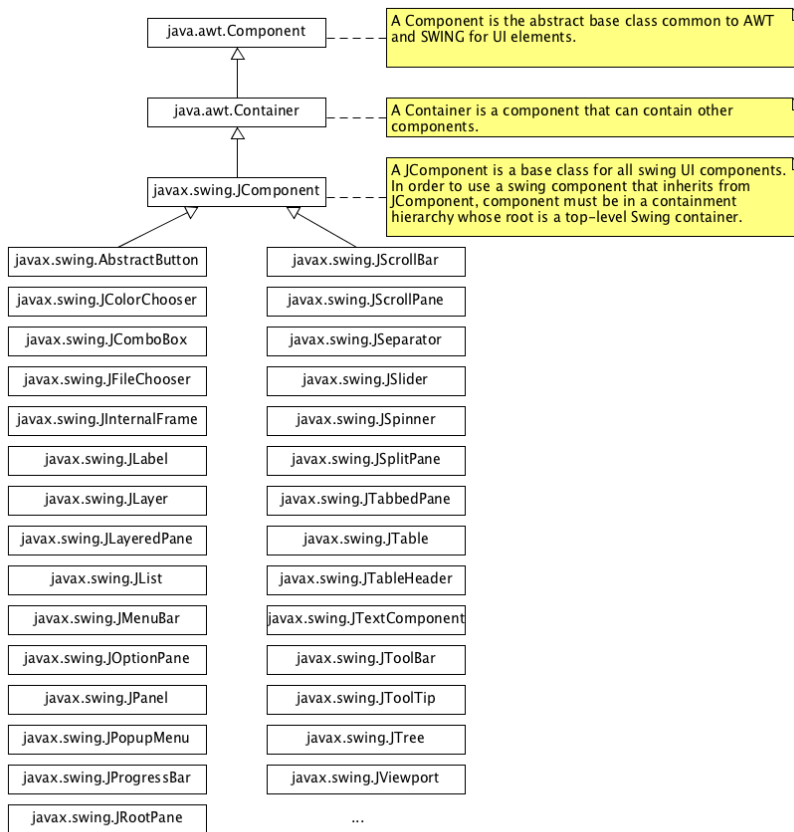


Figure 13.1 Hierarchy linking AWT and Swing

13.2.1 UI components

Visual components

Among the UI Elements some have a visual appearance. For instance:

JLabel A JLabel object is a component for placing text in a container.

JButton This class creates a labeled button.

JColorChooser A JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color.

JCheckBox A JCheckBox is a graphical component that can be in either an on (true) or off (false) state.

JRadioButton The JRadioButton class is a graphical component that can be in either an on (true) or off (false) state in a group.

JList A JList component presents the user with a scrolling list of text items.

JComboBox A JComboBox component presents the user with a to show up menu of choices.

TextField A TextField object is a text component that allows for the editing of a single line of text.

PasswordField A PasswordField object is a text component specialized for password entry.

TextArea A TextArea object is a text component that allows for the editing of a multiple lines of text.

ImageIcon An ImageIcon control is an implementation of the Icon interface that paints icons from images.

Scrollbar A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.

OptionPane A JOptionPane provides set of standard dialog boxes that prompt users for a value or informs them of something.

FileChooser A FileChooser control represents a dialog window from which the user can select a file.

ProgressBar As the task progresses towards completion, the progress bar displays the task's percentage of completion.

Slider A Slider lets the user graphically select a value by sliding a knob within a bounded interval.

Spinner A Spinner is a single line input field that lets the user select a number or an object value from an ordered sequence.

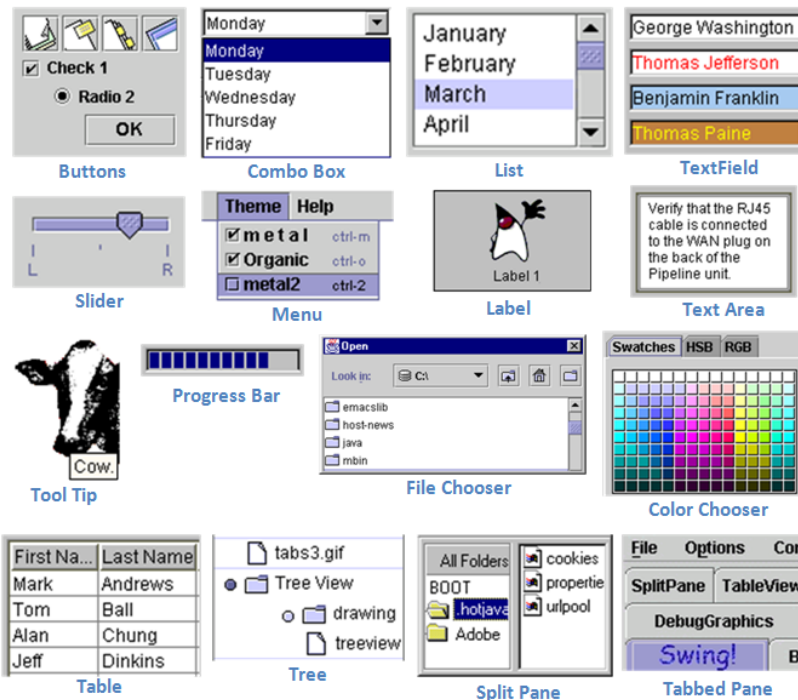


Figure 13.2 Visual aspect of core visual GUI Swing components

Examples of implementations of the visual SWING GUI component can be seen in the demo included in the JDK: 'demo.zip' and particular: 'SwingSet2'. It is available in Chamilo. Other implementations can also be found at <http://zetcode.com/tutorials/javaswingtutorial/>. A rich and very interesting set of tutorials can be found at <https://docs.oracle.com/javase/tutorial/uiswing/components/index.html>.

Here is one of the most basic example:

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloWorldFrame extends JFrame { // JFrame is a standard application window

    HelloWorldFrame() {
        JLabel helloWorldJLabel = new JLabel("Hello World");
        add(helloWorldJLabel);
        setSize(100, 100); // set size or use `pack();' to set it automatically
        setVisible(true); // to display components
    }

    public static void main(String args[]) {
        new HelloWorldFrame();
    }
}
```

Non visual components

Swing provides three generally useful top-level container classes: `JFrame`, `JDialog`, and `JApplet`. When using these classes, you should keep these facts in mind:

- To appear on screen, every GUI component must be part of a containment hierarchy. A containment hierarchy is a tree of components that has a top-level container as its root.
- Each GUI component can be contained only once. If a component is already in a container and if you try to add it to another container, the component will be removed from the first container and then added to the second.
- Each top-level container has a content pane that, generally speaking, contains (directly or indirectly) the visible components in that top-level container's GUI
- You can optionally add a menu bar to a top-level container. The menu bar is by convention positioned within the top-level container, but outside the content pane. Some look and feels, such as the Mac OS look and feel, give you the option of placing the menu bar in another place more appropriate for the look and feel, such as at the top of the screen.

Here's a picture of a frame created by an application. The frame contains a green menu bar (with no menus) and, in the frame's content pane, a large blank, yellow label.

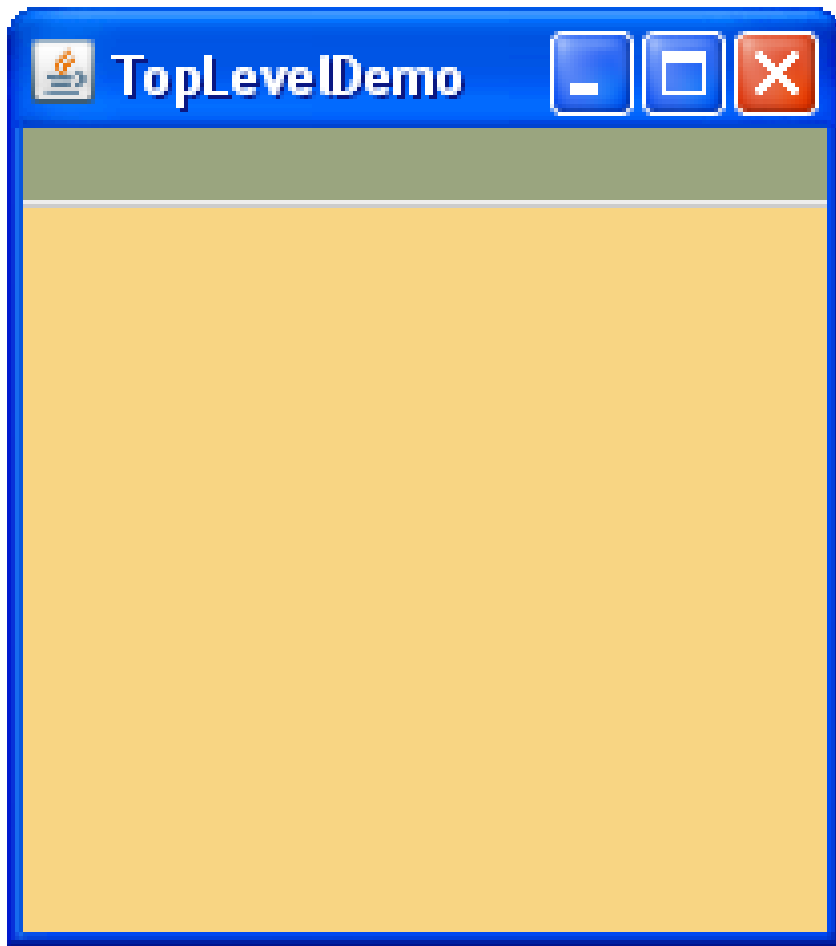


Figure 13.3 Top level architecture

Here's the containment hierarchy for this sample GUI:

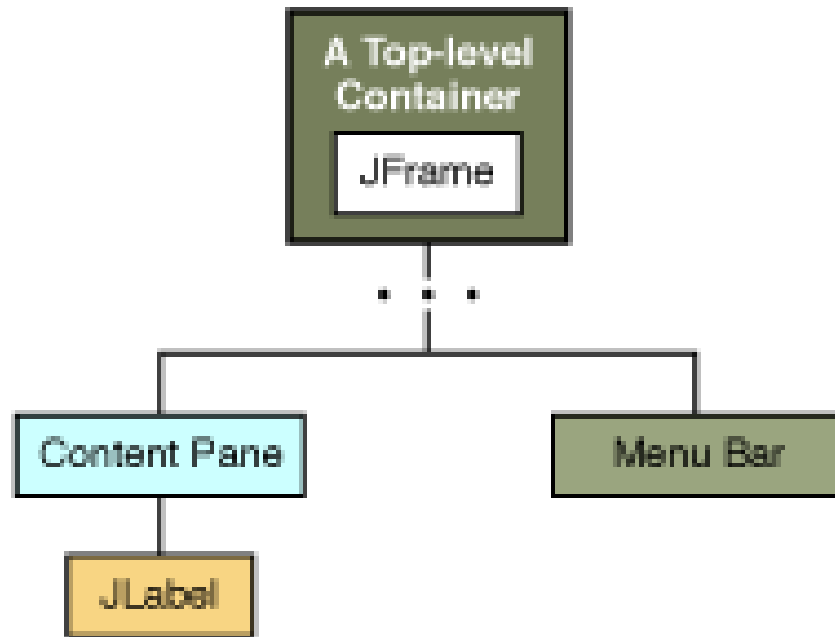


Figure 13.4 Top level objects

Code is given by:

```

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenuBar;

public class TopLevelDemo {

    // Create the GUI and show it. For thread safety, this method should be invoked from the →
    //   ↪ event-dispatching thread.
    private static void createAndShowGUI() {

        JFrame frame = new JFrame("TopLevelDemo"); // Create and set up the window.
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JMenuBar greenMenuBar = new JMenuBar(); // Create the menu bar. Make it have a green →
        //   ↪ background.
        greenMenuBar.setOpaque(true);
        greenMenuBar.setBackground(new Color(154, 165, 127));
        greenMenuBar.setPreferredSize(new Dimension(200, 20));

        JLabel yellowLabel = new JLabel(); //Create a yellow label to put in the content pane.
        yellowLabel.setOpaque(true);
        yellowLabel.setBackground(new Color(248, 213, 131));
        yellowLabel.setPreferredSize(new Dimension(200, 180));

        frame.setJMenuBar(greenMenuBar); //Set the menu bar and add the label to the content →
    }
}

```



```

    ↪ pane.
    frame.getContentPane().add(yellowLabel, BorderLayout.CENTER);

    frame.pack();
    frame.setVisible(true); // Display the window.
}

public static void main(String[] args) {
    // Schedule a job for the event-dispatching thread: creating and showing this application's ↪
    ↪ GUI.
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}
}

```

A standalone application with a Swing-based GUI has at least one containment hierarchy with a `JFrame` as its root. A Swing-based applet² has at least one containment hierarchy, exactly one of which is rooted by a `JApplet` object. But applets are out of the scope of the lesson.

You find the content pane of a top-level container by calling the `getContentPane` method. The default content pane is a simple intermediate container that inherits from `JComponent`, and that uses a `BorderLayout` as its layout manager (it will be explained later).

Frames A `Frame` is a top-level window with a title and a border. It is implemented as an instance of the `JFrame` class. It is a window that has decorations such as a border, a title, and supports button components that close or iconify the window. Applications with a GUI usually include at least one frame. Applets sometimes use frames, as well.

To make a window dependent on another window i.e. disappearing when the other window is iconified for example, use a dialog instead of frame. To make a window that appears within another window, use an internal frame.

Here is another simple but more complete example of frame:

```

import java.awt.BorderLayout;
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.JLabel;

```

```

public class FrameDemo {

```

²An applet is a application that can be run within a browser.

```

// Create the GUI and show it. For thread safety, this method should be invoked from the event →
//   ↪ -dispatching thread.
private static void createAndShowGUI() {
    JFrame frame = new JFrame("FrameDemo"); // Create and set up the window.
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Exit the program when →
    //   ↪ your user closes the frame

    JLabel emptyLabel = new JLabel("Hello World!");
    emptyLabel.setPreferredSize(new Dimension(175, 100));
    frame.getContentPane().add(emptyLabel, BorderLayout.CENTER);

    frame.pack(); // Sizes the frame so that all its contents are at or above their preferred sizes
    frame.setVisible(true); //Display the window.
}

public static void main(String[] args) {
    //Schedule a job for the event-dispatching thread: creating and showing this application's →
    //   ↪ GUI.
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}
}

```

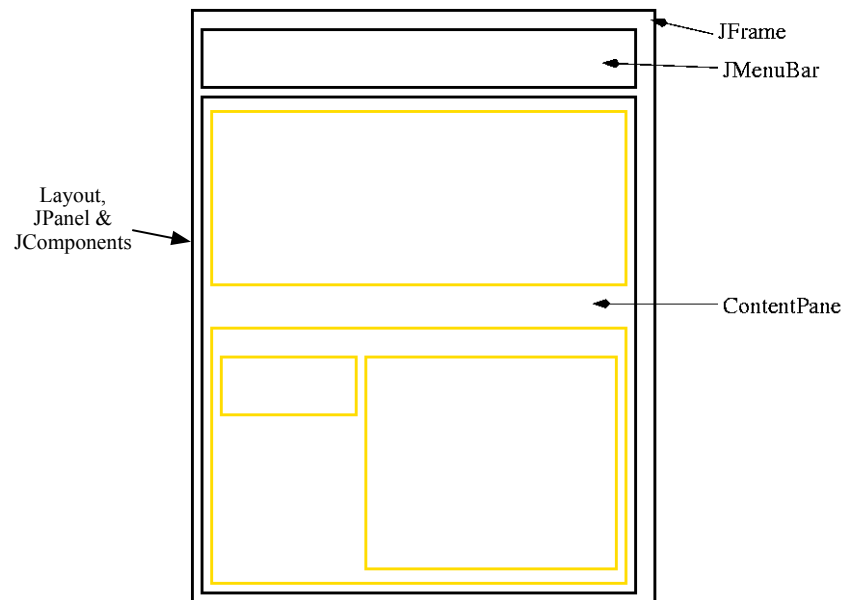


Figure 13.5 Typical elements coming with a JFrame

By default, window decorations are supplied by the native window system. However, you can request that the look-and-feel provides the decorations for a frame. You can also specify that the frame have no window decorations at all, a feature that can be used on its own, or to provide your own decorations, or with full-screen exclusive mode.

Besides specifying who provides the window decorations, you can also specify which icon is used to represent the window. Exactly how this icon is used depends on the window system or look and feel that provides the window decorations. If the window system supports minimization, then the icon is used to represent the minimized window. Most window systems or look and feels also display the icon in the window decorations. A typical icon size is 16×16 pixels, but some window systems use other sizes.

The following snapshots show three frames that are identical except for their window decorations. As you can tell by the appearance of the button in each frame, all three use the Java look and feel. The first uses decorations provided by the window system, which happens to be Microsoft Windows, but could as easily be any other system running the Java platform. The second and third use window decorations provided by the Java look and feel. The third frame uses Java look and feel window decorations, but has a custom icon.

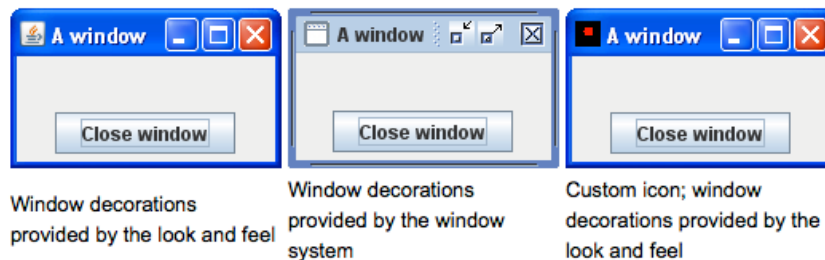


Figure 13.6 Different window decorations of the same frame

Here is an example of creating a frame with a custom icon and with window decorations provided by the look and feel:

```
//Ask for window decorations provided by the look and feel.
JFrame.setDefaultLookAndFeelDecorated(true);

//Create the frame.
JFrame frame = new JFrame("A window");

//Set the frame icon to an image loaded from a file.
frame.setIconImage(new ImageIcon(imgURL).getImage());
```

As the preceding code snippet implies, you must invoke the `setDefaultLookAndFeel`

FeelDecorated method before creating the frame whose decorations you wish to affect. The value you set with `setDefaultLookAndFeelDecorated` is used for all subsequently created JFrames. You can switch back to using window system decorations by invoking `JFrame.setDefaultLookAndFeelDecorated(false)`. Some look and feels might not support window decorations; in this case, the window system decorations are used.

Dialogs A Dialog window is an independent subwindow meant to carry temporary notice apart from the main Swing Application Window. Most Dialogs present an error message or warning to a user, but Dialogs can present images, directory trees, or just about anything compatible with the main Swing Application that manages them.

Here is a code to display a error message:

```
JOptionPane.showMessageDialog(frame, "Eggs are not supposed to be green. →
↳ ");
```

Every dialog is dependent on a Frame component. When that Frame is destroyed, so are its dependent dialogs. When the frame is iconified, its dependent Dialogs also disappear from the screen. When the frame is deiconified, its dependent Dialogs return to the screen. A swing `JDialog` class inherits this behavior from the AWT `Dialog` class.

Here is a dialog example:

```
import java.awt.BorderLayout;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class MyJDialog extends JDialog {

    public MyJDialog(JFrame parent, String title, String message) {
        super(parent, title);
        System.out.println("creating the window..");
        Point p = new Point(400, 400);
        setLocation(p.x, p.y); // set the position of the window

        JPanel messagePane = new JPanel(); // Create a message
        messagePane.add(new JLabel(message));
        getContentPane().add(messagePane); // get content pane, which is usually the Container of →
        ↳ all the dialog's components.
```

```

JPanel buttonPane = new JPanel();
JButton button = new JButton("Close me"); // Create a button
buttonPane.add(button);

button.addActionListener(new MyActionListener()); // set action listener on the button
getContentPane().add(buttonPane, BorderLayout.PAGE_END);
setDefaultCloseOperation(DISPOSE_ON_CLOSE);
pack();
setVisible(true);
}

// an action listener to be used when an action is performed (e.g. button is pressed)
class MyActionListener implements ActionListener {

    //close and dispose of the window.
    public void actionPerformed(ActionEvent e) {
        System.out.println("disposing the window..");
        setVisible(false);
        dispose();
    }
}

public static void main(String[] a) {
    MyJDialog dialog = new MyJDialog(new JFrame(), "Hello", "This is a JDialog →
    ↪ example");
    // set the size of the window
    dialog.setSize(300, 150);
}
}

```

Window A JWindow object is a top-level window with no borders and no menu bar.

Here is an example:

```

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JWindow;

public class JWindowDemo extends JWindow {

    private int X=0;
    private int Y=0;

    public JWindowDemo(){

        setBounds(60,60,100,100);
        addWindowListener(new WindowAdapter(){

```

```

        public void windowClosing(WindowEvent e){
            System.exit(0); //An Exit Listener
        }
    });

    addMouseListener(new MouseAdapter() { //Print (X,Y) coordinates on Mouse Click
        public void mousePressed(MouseEvent e){
            X=e.getX();
            Y=e.getY();
            System.out.println("The (X,Y) coordinate of window is (" +X+", "+Y+" →
↩ +") ");
        }
    });

    addMouseMotionListener(new MouseMotionAdapter()
    {
        public void mouseDragged(MouseEvent e){
            setLocation(getLocation().x+(e.getX()-X),getLocation().y+(e.getY()-Y));
        }
    });

    setVisible(true);
}
public static void main(String[] args){
    new JWindowDemo();
}
}

```

JPanel The JPanel class provides general-purpose containers for lightweight components.

Here is an example:

```

import java.awt.Color;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.GridBagLayout;
import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JPanel;

class JPanelExample extends JFrame {
    JPanel panel1,panel2;
    Dimension dimension;

    public JPanelExample() {
        createAndShowGUI();
    }

    private void createAndShowGUI() {
        setTitle("JPanel Example in Java");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}

```

```
setLayout(new FlowLayout());

panel1=new JPanel(); // An empty panel with FlowLayout
panel2=new JPanel(new GridBagLayout()); // Panel with custom layout

dimension=new Dimension(200,200); // Set some preferred size
panel1.setPreferredSize(dimension);
panel2.setPreferredSize(dimension);

panel1.setBackground(Color.GRAY); // Set some background
panel2.setBackground(Color.DARK_GRAY);

// Set some border. Here a line border of 5 thickness, dark gray color and rounded edges
panel1.setBorder(BorderFactory.createLineBorder(Color.DARK_GRAY,5,true));

// Set some tooltip text
panel1.setToolTipText("Panel 1");
panel2.setToolTipText("Panel 2");

// Add panels
add(panel1);
add(panel2);

setSize(400,400);
setVisible(true);
pack();
}

public static void main(String args[]) {
    new JPanelExample();
}
}
```

13.2.2 Layouts

Layout are the way components added to a container are visually organized. They are different layout managers.

BorderLayout

Every content pane is initialized to use a BorderLayout. A BorderLayout places components in up to five areas: top, bottom, left, right, and center. All extra space is placed in the center area. Tool bars that are created using JToolBar must be created within a BorderLayout container, if you want to be able to drag and drop the bars away from their starting positions.

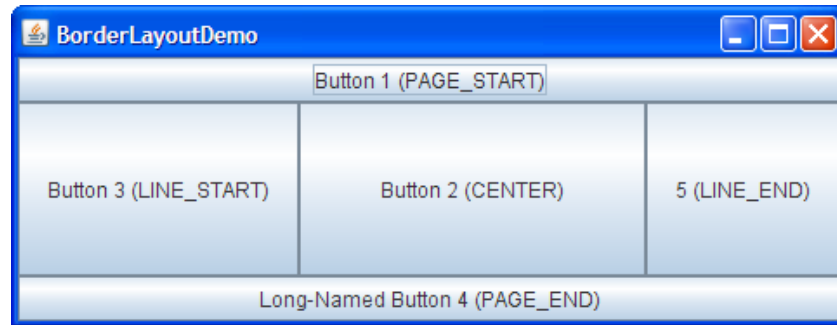


Figure 13.7 BorderLayout

Here is an example:

```
import java.awt.BorderLayout;
import java.awt.Container;
import javax.swing.JButton;
import javax.swing.JFrame;

public class BorderLayoutExample extends JFrame {

    BorderLayoutExample() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = getContentPane();
        container.add(BorderLayout.NORTH,new JButton("North"));
        container.add(BorderLayout.SOUTH,new JButton("South"));
        container.add(BorderLayout.EAST,new JButton("East"));
        container.add(BorderLayout.WEST,new JButton("West"));
        container.add(BorderLayout.CENTER,new JButton("Center"));
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {new BorderLayoutExample();}
}
```

BoxLayout

The BoxLayout class puts components in a single row or column. It respects the components' requested maximum sizes and also lets you align components.

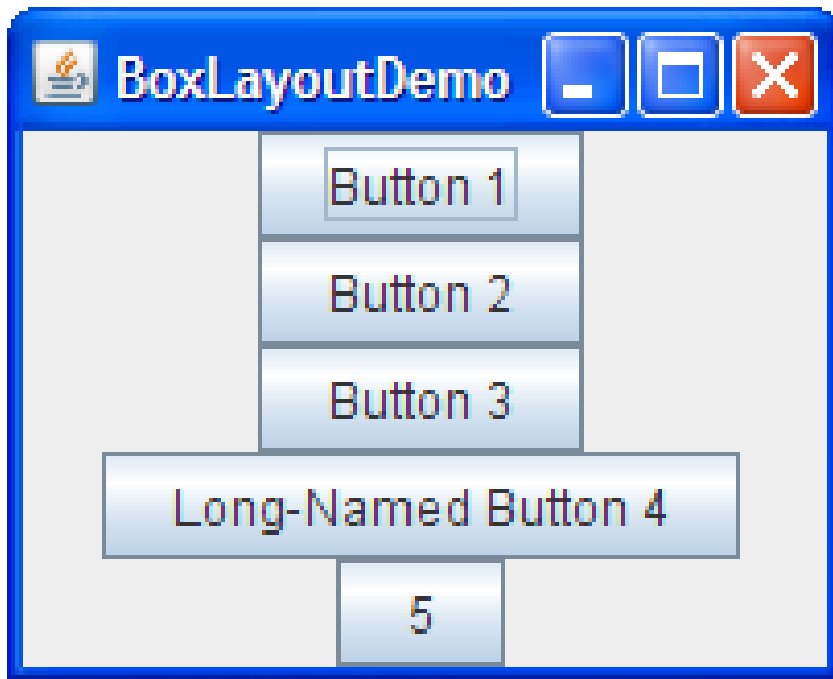


Figure 13.8 BoxLayout

Here is an example:

```
import java.awt.BorderLayout;
import java.awt.Container;
import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JFrame;

public class BorderAndBoxLayoutExample extends JFrame {

    public BorderAndBoxLayoutExample() {
        super("example");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Box verticalBox = Box.createVerticalBox();
        for (int i = 0; i < 5; i++) verticalBox.add(new JButton("bv " + i));
        Box horizontalBox = Box.createHorizontalBox();
        for (int i = 0; i < 5; i++) horizontalBox.add(new JButton("bh " + i));
        Container contentPane = getContentPane();
        contentPane.add(BorderLayout.EAST, verticalBox);
        contentPane.add(BorderLayout.SOUTH, horizontalBox);
        pack();
        setVisible(true);
    }

    public static void main(String[] args) { new BorderAndBoxLayoutExample(); }
}
```

FlowLayout

FlowLayout is the default layout manager for every JPanel. It simply lays out components in a single row, starting a new row if its container is not sufficiently wide.

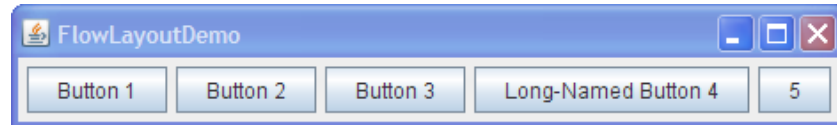


Figure 13.9 FlowLayout

Here is an example:

```
import java.awt.FlowLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class FlowLayoutExample extends JFrame {

    FlowLayoutExample() {
        super("Example");
        addWindowListener(new WindowAdapter() { public void windowClosing(WindowEvent e) →
            ↪ { System.exit(0); } });
        JPanel jPanel = new JPanel(new FlowLayout(FlowLayout.CENTER)); // or LEFT, RIGHT
        jPanel.add(new JButton("button1"));
        jPanel.add(new JButton("button2"));
        add(jPanel);
        pack();
        setVisible(true);
    }

    static public void main(String args[]) {
        new FlowLayoutExample();
    }
}
```

GridLayout

GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns.



Figure 13.10 GridLayout

Here is a code example:

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSeparator;
import javax.swing.UIManager;

public class GridLayoutExample extends JFrame {
    static final String gapList[] = { "0", "10", "15", "20" };
    final static int maxGap = 20;
    JComboBox horGapComboBox;
    JComboBox verGapComboBox;
    JButton applyButton = new JButton("Apply gaps");
    GridLayout experimentLayout = new GridLayout(0,2);

    public GridLayoutExample(String name) {
        super(name);
        setResizable(false);
    }

    public void initGaps() {
        horGapComboBox = new JComboBox(gapList);
        verGapComboBox = new JComboBox(gapList);
    }
}
```

```

public void addComponentsToPane(final Container pane) {
    initGaps();
    final JPanel compsToExperiment = new JPanel();
    compsToExperiment.setLayout(experimentLayout);
    JPanel controls = new JPanel();
    controls.setLayout(new GridLayout(2,3));

    JButton b = new JButton("Just fake button");
    Dimension buttonSize = b.getPreferredSize();
    compsToExperiment.setPreferredSize(new Dimension((int)(buttonSize.getWidth() * 2.5)+ →
    ↪ maxGap,
        (int)(buttonSize.getHeight() * 3.5)+maxGap * 2));

    compsToExperiment.add(new JButton("Button 1"));
    compsToExperiment.add(new JButton("Button 2"));
    compsToExperiment.add(new JButton("Button 3"));
    compsToExperiment.add(new JButton("Long-Named Button 4"));
    compsToExperiment.add(new JButton("5"));

    controls.add(new Label("Horizontal gap:"));
    controls.add(new Label("Vertical gap:"));
    controls.add(new Label(" "));
    controls.add(horGapComboBox);
    controls.add(verGapComboBox);
    controls.add(applyButton);

    //Process the Apply gaps button press
    applyButton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            //Get the horizontal gap value
            String horGap = (String)horGapComboBox.getSelectedItem();
            //Get the vertical gap value
            String verGap = (String)verGapComboBox.getSelectedItem();
            //Set up the horizontal gap value
            experimentLayout.setHgap(Integer.parseInt(horGap));
            //Set up the vertical gap value
            experimentLayout.setVgap(Integer.parseInt(verGap));
            //Set up the layout of the buttons
            experimentLayout.layoutContainer(compsToExperiment);
        }
    });
    pane.add(compsToExperiment, BorderLayout.NORTH);
    pane.add(new JSeparator(), BorderLayout.CENTER);
    pane.add(controls, BorderLayout.SOUTH);
}

private static void createAndShowGUI() {
    GridLayoutExample frame = new GridLayoutExample("GridLayoutDemo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.addComponentsToPane(frame.getContentPane());
    frame.pack();
    frame.setVisible(true);
}

```

```
public static void main(String[] args) {
    try {
        UIManager.setLookAndFeel("javax.swing.plaf.metal. →
        ↪ MetalLookAndFeel");
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    UIManager.put("swing.boldMetal", Boolean.FALSE); // Turn off metal's use of bold →
    ↪ fonts
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}
```

CardLayout

The CardLayout class lets you implement an area that contains different components at different times. A CardLayout is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the CardLayout displays. An alternative to using CardLayout is using a tabbed pane, which provides similar functionality but with a pre-defined GUI.



Figure 13.11 CardLayout

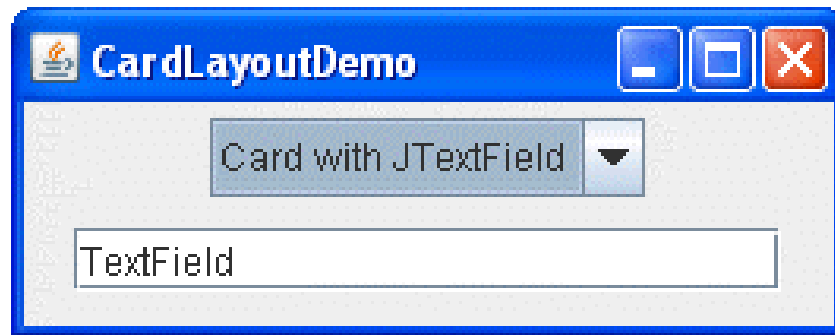


Figure 13.12 CardLayout

Here is a code example:

```
import java.awt.BorderLayout;
import java.awt.CardLayout;
import java.awt.Container;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class CardLayoutExample implements ItemListener {
    JPanel cards; //a panel that uses CardLayout
    final static String BUTTONPANEL = "Card with JButtons";
    final static String TEXTPANEL = "Card with JtextField";

    public void addComponentToPane(Container pane) {
        //Put the JComboBox in a JPanel to get a nicer look.
        JPanel comboBoxPane = new JPanel(); //use FlowLayout
        String comboBoxItems[] = { BUTTONPANEL, TEXTPANEL };
        JComboBox comboBox = new JComboBox(comboBoxItems);
        comboBox.setEditable(false);
        comboBox.addItemListener(this);
        comboBoxPane.add(comboBox);

        //Create the "cards".
        JPanel card1 = new JPanel();
        card1.add(new JButton("Button 1"));
        card1.add(new JButton("Button 2"));
        card1.add(new JButton("Button 3"));

        JPanel card2 = new JPanel();
        card2.add(new JTextField("TextField", 20));

        //Create the panel that contains the "cards".
        cards = new JPanel(new CardLayout());
        cards.add(card1, BUTTONPANEL);
        cards.add(card2, TEXTPANEL);
    }
}
```

```
pane.add(comboBoxPane, BorderLayout.PAGE_START);
pane.add(cards, BorderLayout.CENTER);
}

public void itemStateChanged(ItemEvent evt) {
    CardLayout cl = (CardLayout)(cards.getLayout());
    cl.show(cards, (String)evt.getItem());
}

private static void createAndShowGUI() {
    JFrame frame = new JFrame("CardLayoutDemo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    CardLayoutExample demo = new CardLayoutExample();
    demo.addComponentToPane(frame.getContentPane());

    frame.pack();
    frame.setVisible(true);
}

public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}
```

GridBagLayout

GridBagLayout is a sophisticated, flexible layout manager. It aligns components by placing them within a grid of cells, allowing components to span more than one cell. The rows in the grid can have different heights, and grid columns can have different widths.

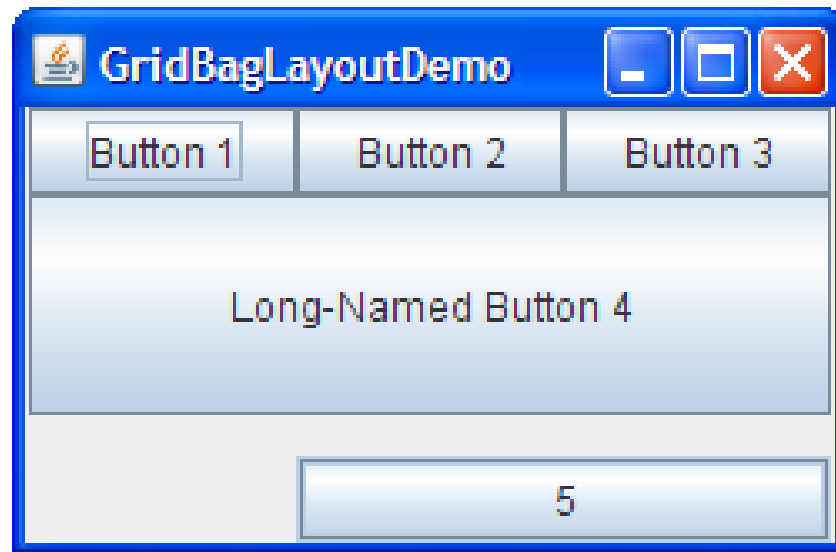


Figure 13.13 GridBagLayout

Here is a code example:

```
import java.awt.ComponentOrientation;
import java.awt.Container;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import javax.swing.JButton;
import javax.swing.JFrame;

public class GridBagLayoutExample {
    final static boolean shouldFill = true;
    final static boolean shouldWeightX = true;
    final static boolean RIGHT_TO_LEFT = false;

    public static void addComponentsToPane(Container pane) {
        if (RIGHT_TO_LEFT) {
            pane.setComponentOrientation(ComponentOrientation.RIGHT_TO_LEFT);
        }

        JButton button;
        pane.setLayout(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints();
        if (shouldFill) {
            //natural height, maximum width
            c.fill = GridBagConstraints.HORIZONTAL;
        }

        button = new JButton("Button 1");
        if (shouldWeightX) {
            c.weightx = 0.5;
        }
        c.fill = GridBagConstraints.HORIZONTAL;
```



```

c.gridx = 0;
c.gridy = 0;
pane.add(button, c);

button = new JButton("Button 2");
c.fill = GridBagConstraints.HORIZONTAL;
c.weightx = 0.5;
c.gridx = 1;
c.gridy = 0;
pane.add(button, c);

button = new JButton("Button 3");
c.fill = GridBagConstraints.HORIZONTAL;
c.weightx = 0.5;
c.gridx = 2;
c.gridy = 0;
pane.add(button, c);

button = new JButton("Long-Named Button 4");
c.fill = GridBagConstraints.HORIZONTAL;
c.ipady = 40; //make this component tall
c.weightx = 0.0;
c.gridwidth = 3;
c.gridx = 0;
c.gridy = 1;
pane.add(button, c);

button = new JButton("5");
c.fill = GridBagConstraints.HORIZONTAL;
c.ipady = 0; //reset to default
c.weighty = 1.0; //request any extra vertical space
c.anchor = GridBagConstraints.PAGE_END; //bottom of space
c.insets = new Insets(10,0,0,0); //top padding
c.gridx = 1; //aligned with button 2
c.gridwidth = 2; //2 columns wide
c.gridy = 2; //third row
pane.add(button, c);
}

private static void createAndShowGUI() {
    JFrame frame = new JFrame("GridBagLayoutDemo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    addComponentsToPane(frame.getContentPane());
    frame.pack();
    frame.setVisible(true);
}

public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}
}

```

GridLayout

GridLayout is a layout manager that was developed for use by GUI builder tools, but it can also be used manually. GridLayout works with the horizontal and vertical layouts separately. The layout is defined for each dimension independently. Consequently, however, each component needs to be defined twice in the layout.

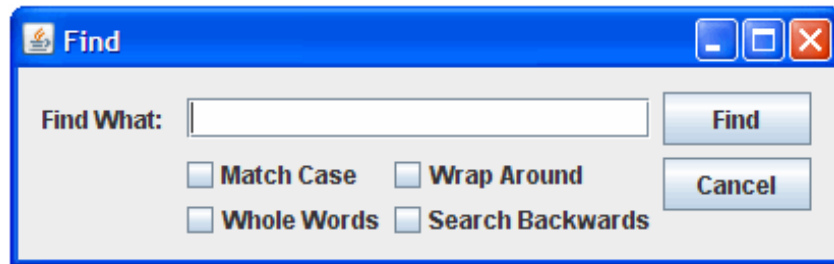


Figure 13.14 GridLayout

Here is a code example:

```
import static javax.swing.GroupLayout.Alignment.BASELINE;
import static javax.swing.GroupLayout.Alignment.LEADING;
import javax.swing.BorderFactory;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.SwingConstants;
import javax.swing.WindowConstants;

public class GroupLayoutExample extends JFrame {
    public GroupLayoutExample() {
        JLabel label = new JLabel("Find What:");
        JTextField textField = new JTextField();
        JCheckBox caseCheckBox = new JCheckBox("Match Case");
        JCheckBox wrapCheckBox = new JCheckBox("Wrap Around");
        JCheckBox wholeCheckBox = new JCheckBox("Whole Words");
        JCheckBox backCheckBox = new JCheckBox("Search Backwards");
        JButton findButton = new JButton("Find");
        JButton cancelButton = new JButton("Cancel");

        // remove redundant default border of check boxes – they would hinder
        // correct spacing and aligning (maybe not needed on some look and feels)
        caseCheckBox.setBorder(BorderFactory.createEmptyBorder(0, 0, 0, 0));
        wrapCheckBox.setBorder(BorderFactory.createEmptyBorder(0, 0, 0, 0));
        wholeCheckBox.setBorder(BorderFactory.createEmptyBorder(0, 0, 0, 0));
        backCheckBox.setBorder(BorderFactory.createEmptyBorder(0, 0, 0, 0));

        GroupLayout layout = new GroupLayout(getContentPane());
```

```

getContentPane().setLayout(layout);
layout.setAutoCreateGaps(true);
layout.setAutoCreateContainerGaps(true);

layout.setHorizontalGroup(layout.createSequentialGroup()
    .addComponent(label)
    .addGroup(layout.createParallelGroup(LEADING)
        .addComponent(textField)
        .addGroup(layout.createSequentialGroup()
            .addGroup(layout.createParallelGroup(LEADING)
                .addComponent(caseCheckBox)
                .addComponent(wholeCheckBox))
            .addGroup(layout.createParallelGroup(LEADING)
                .addComponent(wrapCheckBox)
                .addComponent(backCheckBox))))
    .addGroup(layout.createParallelGroup(LEADING)
        .addComponent(findButton)
        .addComponent(cancelButton))
);

layout.linkSize(SwingConstants.HORIZONTAL, findButton, cancelButton);

layout.setVerticalGroup(layout.createSequentialGroup()
    .addGroup(layout.createParallelGroup(BASELINE)
        .addComponent(label)
        .addComponent(textField)
        .addComponent(findButton))
    .addGroup(layout.createParallelGroup(LEADING)
        .addGroup(layout.createSequentialGroup()
            .addGroup(layout.createParallelGroup(BASELINE)
                .addComponent(caseCheckBox)
                .addComponent(wrapCheckBox))
            .addGroup(layout.createParallelGroup(BASELINE)
                .addComponent(wholeCheckBox)
                .addComponent(backCheckBox)))
        .addComponent(cancelButton))
);

setTitle("Find");
pack();
setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
}

public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new GroupLayoutExample().setVisible(true);
        }
    });
}
}

```

SpringLayout

SpringLayout is a flexible layout manager designed for use by GUI builders. It lets you specify precise relationships between the edges of components under its control. For example, you might define that the left edge of one component is a certain distance (which can be dynamically calculated) from the right edge of a second component. SpringLayout lays out the children of its associated container according to a set of constraints.

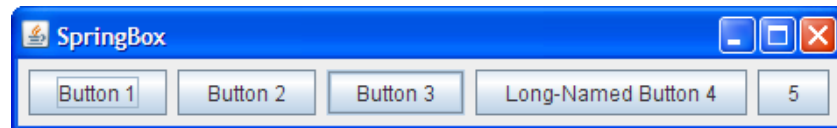


Figure 13.15 SpringLayout

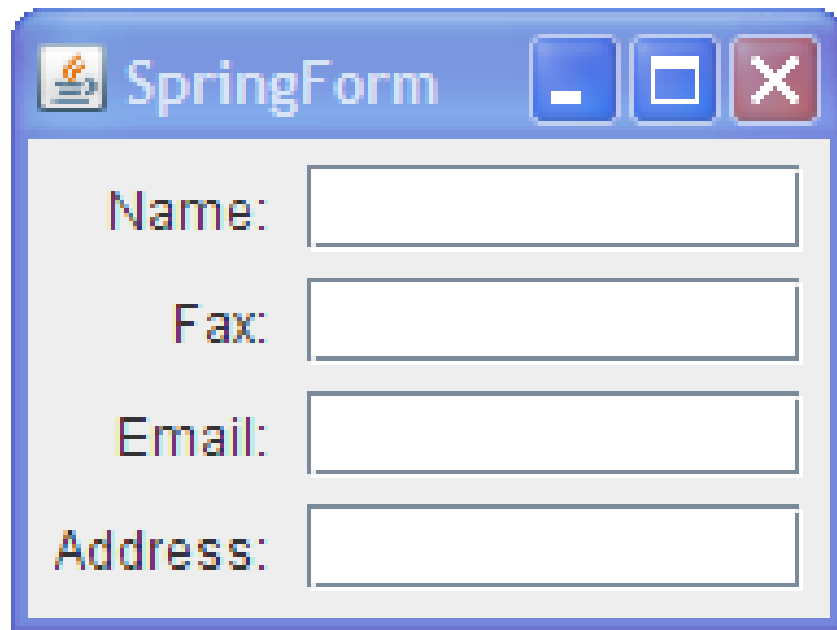


Figure 13.16 SpringLayout

Here is a code example:

```
import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.SpringLayout;

public class SpringLayoutExample {
    private static void createAndShowGUI() {
```

```

JFrame frame = new JFrame("SpringLayoutExample");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Set up the content pane.
Container contentPane = frame.getContentPane();
SpringLayout layout = new SpringLayout();
contentPane.setLayout(layout);

//Create and add the components.
JLabel label = new JLabel("Label: ");
JTextField textField = new JTextField("Text field", 15);
contentPane.add(label);
contentPane.add(textField);

//Adjust constraints for the label so it is at (5,5).
layout.putConstraint(SpringLayout.WEST, label, 5, SpringLayout.WEST, contentPane);
layout.putConstraint(SpringLayout.NORTH, label, 5, SpringLayout.NORTH, contentPane);

//Adjust constraints for the text field so it is at(<label's right edge> + 5, 5).
layout.putConstraint(SpringLayout.WEST, textField, 5, SpringLayout.EAST, label);
layout.putConstraint(SpringLayout.NORTH, textField, 5, SpringLayout.NORTH, →
↳ contentPane);

//Adjust constraints for the content pane: Its right edge should be 5 pixels beyond the text →
↳ field's right
//edge, and its bottom edge should be 5 pixels beyond the bottom edge of the tallest →
↳ component (which we'll
//assume is textField).
layout.putConstraint(SpringLayout.EAST, contentPane, 5, SpringLayout.EAST, textField);
layout.putConstraint(SpringLayout.SOUTH, contentPane, 5, SpringLayout.SOUTH, →
↳ textField);

frame.pack();
frame.setVisible(true);
}

public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}
}

```

Spacing

Invisible components can be added to a layout to produce empty space between components. The most useful are rigid areas (or struts). Glue may be insert expandable empty space in a layout.

Rigid areas have a fixed horizontal and vertical size. A rigid area is a fixed size filler with vertical and horizontal dimensions. It can be used in either horizontal or vertical layouts. Create a rigid area by specifying its width and height in pixels in a `Dimension` object, which has two parameters, an x distance and a y distance. Typically the dimension you are not interested in is set to zero.

```
p.add(Box.createRigidArea(new Dimension(10, 0)));
```

This creates a rigid area component 10 pixels wide and 0 pixels high and adds it to a panel.

Glue is an expandable component useful if you want spacing to increase when a window is resized. Glue is an invisible component that can expand. It is more like a spring or sponge than glue. Put a glue component where extra space should appear (disappear from) when a window is resized. Use vertical glue in a vertical `BoxLayout` and horizontal glue in a horizontal layout. For example, this will allow extra vertical space between two buttons.

```
JPanel p = new JPanel();  
p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));  
p.add(button1);  
p.add(Box.createVerticalGlue()); // This will expand/contract as needed.  
p.add(button2);
```

Use `Box.createHorizontalGlue()` for horizontal expansion.

Struts are fixed either vertically or horizontally. Generally use rigid areas instead of struts. Create a strut by specifying its size in pixels, and adding it to the panel at the point you want the space between other components. Use horizontal struts only in horizontal layouts and vertical struts only in vertical layouts, otherwise there will be problems. To avoid problems from nested panels, you are generally better off using a rigid area.

```
p.add(Box.createHorizontalStrut(10));
```

Use `Box.createVerticalStrut()` for vertical areas.

`Box.Filler` is a more general spacing component with specified minimum, preferred, and maximum sizes. You can create your own filler by using the `Box.Filler` constructor and specifying the minimum, preferred, and maximum size. Each size must be in a `Dimension` object, which has width and height.

```
Box.Filler myFiller = new Box.Filler(min, pref, max);
```

For example, to create a new horizontal filler that can get no smaller than 4 pixels, that prefers to be 16 pixels wide, and that will expand to no more than 32 pixels,

you could do this:

```
Box.Filler hFill = new Box.Filler(new Dimension(4,0),  
    new Dimension(16, 0),  
    new Dimension(32, 0));
```

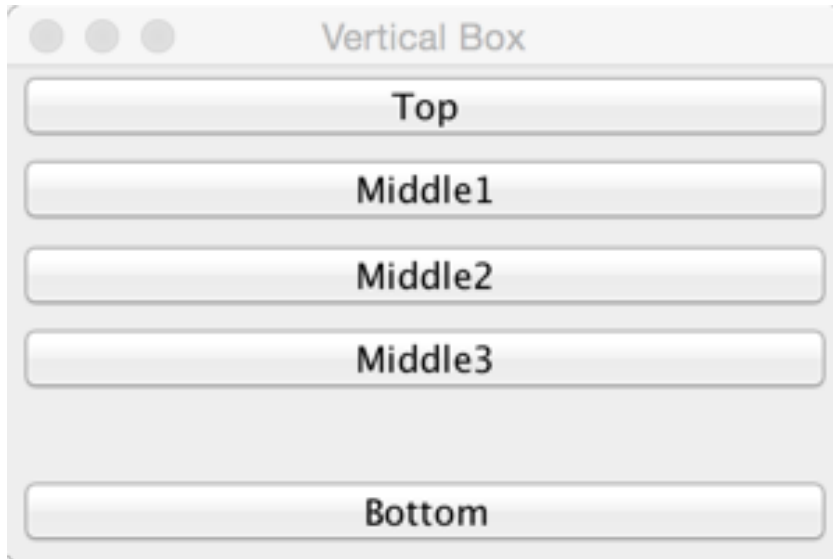


Figure 13.17 Struts and glue

Here is code example:

```
import java.awt.BorderLayout;  
import java.awt.Button;  
import javax.swing.Box;  
import javax.swing.JFrame;  
  
public class StrutsAndGlueExample {  
    public static void main(String args[]) {  
        JFrame frame = new JFrame("Vertical Box");  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        Box box = Box.createVerticalBox();  
        box.add(new Button("Top"));  
        box.add(new Button("Middle1"));  
        box.add(Box.createGlue());  
        box.add(new Button("Middle2"));  
        box.add(new Button("Middle3"));  
        box.add(Box.createVerticalStrut(25));  
        box.add(new Button("Bottom"));  
        frame.add(box, BorderLayout.CENTER);  
        frame.setSize(300, 200);  
        frame.setVisible(true);  
    }  
}
```

13.2.3 Events

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

The events can be broadly classified into two categories:

Foreground Events Those events caused by the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page,...

Background Events Those events not caused by the direct interaction of end-user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

Event handling mechanism

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events. Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

source The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. Java provide as with classes for source object.

listener It is also known as event handler. Listener is responsible for generating response to an event. From Java implementation point of view, the listener is also an object. Listener waits until it receives an event. Once the event is received, the listener processes the event and then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model, Listener needs to be registered with the source object so that the listener can receive the

event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

Here are the usual steps in event handling:

1. The user clicks the button and the event is generated.
2. The object of concerned event class is created automatically with information about the source.
3. Event object is forwarded to the method of registered listener class.
4. The method is now executed and returns.

In order to design a listener class we have to develop some listener interfaces. These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class. If you do not implement the predefined interfaces then your class can not act as a listener class for a source object.

Examples of implementation

Most basic approach In the next example, the frame itself registers with the 'JButton' as a listener, more particularly as an 'ActionListener', which is called when a MouseEvent is captured. The callback method is then named `public void` → `actionPerformed(ActionEvent actionEvent)`. It is specified by the interface ActionListener → `actionPerformed(ActionEvent actionEvent)`, that must be implemented to register as an action listener: `button.addActionListener` → `(this);`.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
```

```
public class HelloWorldFrame extends JFrame implements ActionListener {
    JLabel jLabel;
    JButton jButton;
    HelloWorldFrame() {
        super("Hello World");
        addWindowListener(new WindowAdapter() {public void windowClosing(WindowEvent e) →
            { System.exit(0); } });
        jLabel = new JLabel("Hello World from Swing framework");
        jButton = new JButton("Click me!");
        jButton.addActionListener(this);
        JPanel jPanel = new JPanel();
        jPanel.add(jLabel);
        jPanel.add(jButton);
        add(jPanel);
    }
}
```

```

        pack();
        setVisible(true);
    }
    public void actionPerformed(ActionEvent actionEvent) {
        jLabel.setText("clicked at "+actionEvent.getWhen());
    }
    static public void main(String args[]) {
        new HelloWorldFrame();
    }
}

```

Better approach The next variant is better from a MVC point of view because it distinguishes view (display) from control (event handling) by creating 2 classes.

```

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class HelloWorldFrame extends JFrame {
    JLabel jLabel;
    JButton jButton;
    HelloWorldFrame() {
        super("Hello World");
        addWindowListener(new WindowAdapter() {public void windowClosing(WindowEvent e) →
            ↪ {System.exit(0);}});
        jLabel = new JLabel("Hello World from Swing framework");
        jButton = new JButton("Click me!");
        jButton.addActionListener(new HelloWorldListener(jLabel));
        JPanel jPanel = new JPanel();
        jPanel.add(jLabel);
        jPanel.add(jButton);
        add(jPanel);
        pack();
        setVisible(true);
    }
    static public void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() {public void run() {new HelloWorldFrame() →
            ↪ ;}});
    }
}

```

and

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JLabel;

public class HelloWorldListener implements ActionListener {

```

```

JLabel jLabel;

public HelloWorldListener(JLabel jLabel) {
    this.jLabel=jLabel;
}
public void actionPerformed(ActionEvent actionEvent) {
    jLabel.setText("clicked at "+actionEvent.getWhen());
}
}

```

Best approach Few years ago, Java designers have introduced specific objects for handling events: the actions. The principle is to extend an action and to inherit its properties and capabilities. In this way, Java can reuse event handler at different places. For instance, a same event handler can be bound both to a button and to a menu item. It can embed an icon displayed both in the button and in the menu. Obviously, this approach is also MVC compliant.

The previous example becomes:

```

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class HelloWorldFrame extends JFrame {
    JLabel jLabel;
    JButton jButton;
    HelloWorldFrame() {
        super("Hello World");
        addWindowListener(new WindowAdapter() {public void windowClosing(WindowEvent e) →
            ↪ {System.exit(0);} });
        jLabel = new JLabel("Hello World from Swing framework");
        jButton = new JButton();
        jButton.setAction(new HelloWorldAction(this));
        JPanel jPanel = new JPanel();
        jPanel.add(jLabel);
        jPanel.add(jButton);
        add(jPanel);
        pack();
        setVisible(true);
    }
    public void setText(String text) {
        jLabel.setText(text);
    }
    static public void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() {public void run() {new HelloWorldFrame() →
            ↪ ;}});
    }
}

```

```
}

```

and the action handling mouse clicked events:

```
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;

public class HelloWorldAction extends AbstractAction {
    HelloWorldFrame helloWorldFrame;
    public HelloWorldAction(HelloWorldFrame helloWorldFrame) {
        super("Click me!");
        this.helloWorldFrame=helloWorldFrame;
    }
    public void actionPerformed(ActionEvent actionEvent) {
        helloWorldFrame.setText("clicked at "+actionEvent.getWhen());
    }
}
```

List of available events and listeners

Events EventObject class is the root class from which all event state objects shall be derived. All Events are constructed with a reference to the object, the source, that is logically deemed to be the object upon which the Event in question initially occurred upon. This class is defined in java.util package.

It contains 2 methods:

- Object getSource(). The object on which the Event initially occurred.
- String toString(). It returns a String representation of this EventObject.

The main events are:

ActionEvent The ActionEvent is generated when button is clicked or the item of a list is double clicked.

InputEvent The InputEvent class is root event class for all component-level input events.

KeyEvent On entering the character the Key event is generated.

MouseEvent This event indicates a mouse action occurred in a component.

WindowEvent The object of this class represents the change in state of a window.

AdjustmentEvent The object of this class represents the adjustment event emitted by Adjustable objects.

ComponentEvent The object of this class represents the change in state of a window.

ContainerEvent The object of this class represents the change in state of a window.

MouseEvent The object of this class represents the change in state of a window.

PaintEvent The object of this class represents the change in state of a window.

Listeners The Event listeners represent the interfaces responsible to handle events. Java provides various Event listener classes but we will discuss those which are more frequently used. Every method of an event listener method has a single argument as an object which is subclass of EventObject class. For example, mouse event listener methods will accept instance of MouseEvent, where MouseEvent derives from EventObject.

EventListener interface is a marker interface which every listener interface has to extend. This class is defined in java.util package.

The main listeners are:

ActionListener This interface is used for receiving the action events.

ComponentListener This interface is used for receiving the component events.

ItemListener This interface is used for receiving the item events.

KeyListener This interface is used for receiving the key events.

MouseListener This interface is used for receiving the mouse events.

WindowListener This interface is used for receiving the window events.

AdjustmentListener This interface is used for receiving the adjustment events.

ContainerListener This interface is used for receiving the container events.

MouseEventListener This interface is used for receiving the mouse motion events.

FocusListener This interface is used for receiving the focus events.

Adapters Adapters are abstract classes for receiving various events. It is used to be extended. The methods in these classes are empty: you just have to defined the method corresponding to the event you want to capture. These classes exists as convenience for creating listener objects.

Following is the list of commonly used adapters while listening GUI events in SWING.

FocusAdapter An abstract adapter class for receiving focus events.

KeyAdapter An abstract adapter class for receiving key events.

MouseAdapter An abstract adapter class for receiving mouse events.

MouseEventAdapter An abstract adapter class for receiving mouse motion events.

WindowAdapter An abstract adapter class for receiving window events.

What you should remember from this chapter?

- What Model-View-Control model consists of?
- Can you depict the event handling mechanism?
- What are layout managers for?
- What the different elements of a GUI?
- What is the difference between JFrame and a JDialog?
- What is a JPanel?
- What is a listener? an event?
- Explain the event handling mechanism?
- What is a Action in event handling mechanism?

Chapter 14

Plotting curves into Java

14.1 Introduction to JFreeChart

Plots can be carried out using Java2D but there is a lot to develop before getting a beautiful curve or graph. We are going to use JFreeChart. JFreeChart is a free 100% Java chart library that makes it easy for developers to display professional quality charts. JFreeChart's extensive feature set includes (see <http://www.jfree.org/jfreechart/>):

1. a consistent and well-documented API, supporting a wide range of chart types
2. a flexible design that is easy to extend, and targets both server-side and client-side applications
3. support for many output types, including Swing and JavaFX components, image files (including PNG and JPEG), and vector graphics file formats (including PDF, EPS and SVG)
4. JFreeChart is open source or, more specifically, free software. It is distributed under the terms of the GNU Lesser General Public License (LGPL), which permits use in proprietary applications

Sample screenshots can be seen at <http://www.jfree.org/jfreechart/samples.html>

In order to be able to use JFreeChart, you must install the 'jar' files of JFreeChart in the dependencies of your project available at <https://sourceforge.net/projects/jfreechart/files/>. Download 'jfreechart-1.0.19.zip' and unzip. Create a 'lib' folder in your eclipse project and drop all the files except 'jfreechart-1.0.19-swt.jar' in the 'jfreechart-1.0.19/lib' folder into your own project 'lib' folder. Then, select the project in the 'Package Explorer', right click, choose 'Properties', then select 'Java Build Path'. Select Libraries, then 'add JARs' and select all the 'jar' files you added

in your project lib folder. JFreeChart can then be called from your project.

Installation and developer guides are available in Chamilo. Demo codes are also available.

Creating charts with JFreeChart is a three step process. You need to:

- create a dataset containing the data to be displayed in the chart
- create a JFreeChart object that will be responsible for drawing the chart
- draw the chart to some output target (often, but not always, a panel on the screen)

To illustrate the process, we describe a sample application (First.java) that produces the pie chart shown below.

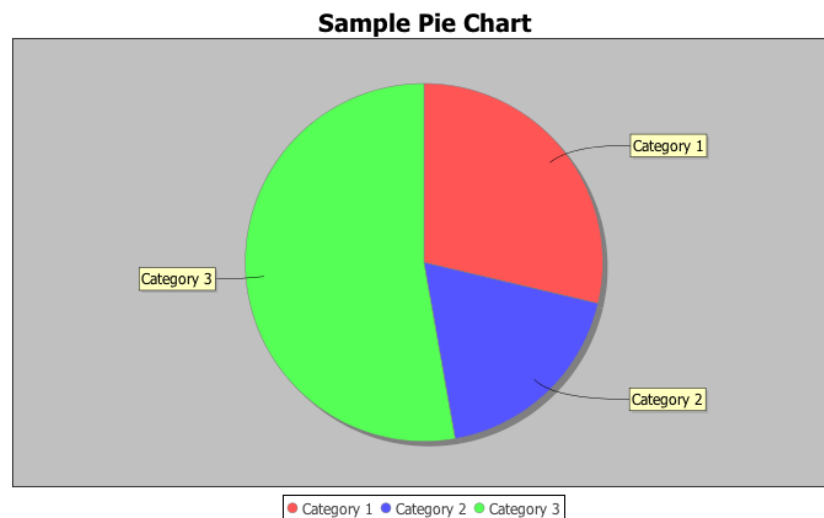


Figure 14.1 Example of Pie chart

Code is given by:

```
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartFrame;
import org.jfree.chart.JFreeChart;
import org.jfree.data.general.DefaultPieDataset;

public class First {

    public static void main(String[] args) {

        // create a dataset...
        DefaultPieDataset data = new DefaultPieDataset();
        data.setValue("Category 1", 43.2);
        data.setValue("Category 2", 27.9);
```



```
data.setValue("Category 3", 79.5);

// create a chart...
JFreeChart chart = ChartFactory.createPieChart(
    "Sample Pie Chart",
    data,
    true, // legend?
    true, // tooltips?
    false // URLs?
);

// create and display a frame...
ChartFrame frame = new ChartFrame("JFreeChart: First.java", chart);
frame.pack();
frame.setVisible(true);

}

}
```

Each of the three steps outlined above is described, along with sample code, in the following paragraphs.

14.1.1 Collecting data

Step one requires us to create a dataset for our chart. This can be done easily using the `DefaultPieDataset` class, as follows:

```
// create a dataset...
DefaultPieDataset dataset = new DefaultPieDataset();
dataset.setValue("Category 1", 43.2);
dataset.setValue("Category 2", 27.9);
dataset.setValue("Category 3", 79.5);
```

Note that `JFreeChart` can create pie charts using data from any class that implements the `PieDataset` interface. The `DefaultPieDataset` class (used above) provides a convenient implementation of this interface, but you are free to develop an alternative dataset implementation if you want to.

14.1.2 Creating chart

Step two concerns how we will present the dataset created in the previous section. We need to create a `JFreeChart` object that can draw a chart using the data from our pie dataset. We will use the `ChartFactory` class, as follows:

```
// create a chart...
JFreeChart chart = ChartFactory.createPieChart(
    "Sample Pie Chart",
    dataset,
    true, // legend?
    true, // tooltips?
    false // URLs?
);
```

Notice how we have passed a reference to the dataset to the factory method. JFreeChart keeps a reference to this dataset so that it can obtain data later on when it is drawing the chart.

The chart that we have created uses default settings for most attributes. There are many ways to customize the appearance of charts created with JFreeChart, but in this example we will just accept the defaults.

14.1.3 Displaying the chart

The final step is to display the chart somewhere. JFreeChart is very flexible about where it draws charts, thanks to its use of the Graphics2D class.

For now, let's display the chart in a frame on the screen. The ChartFrame class contains the machinery (a ChartPanel) required to display charts:

```
// create and display a frame...
ChartFrame frame = new ChartFrame("Test ", chart);
frame.pack();
frame.setVisible(true);
```

Let's focus now on some other useful plots.

14.2 LineChartDemo2

In the developer guide, you can find this useful example:

```
import javax.swing.JPanel;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
```

```
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;

public class LineChartDemo2 extends ApplicationFrame {

    public LineChartDemo2(String title) {
        super(title);
        JPanel chartPanel = createDemoPanel();
        chartPanel.setPreferredSize(new java.awt.Dimension(500, 270));
        setContentPane(chartPanel);
    }

    private static XYDataset createDataset() { // Creates a sample dataset

        XYSeries series1 = new XYSeries("First");
        series1.add(1.0, 1.0);
        series1.add(2.0, 4.0);
        series1.add(3.0, 3.0);
        series1.add(4.0, 5.0);
        series1.add(5.0, 5.0);
        series1.add(6.0, 7.0);
        series1.add(7.0, 7.0);
        series1.add(8.0, 8.0);

        XYSeries series2 = new XYSeries("Second");
        series2.add(1.0, 5.0);
        series2.add(2.0, 7.0);
        series2.add(3.0, 6.0);
        series2.add(4.0, 8.0);
        series2.add(5.0, 4.0);
        series2.add(6.0, 4.0);
        series2.add(7.0, 2.0);
        series2.add(8.0, 1.0);

        XYSeries series3 = new XYSeries("Third");
        series3.add(3.0, 4.0);
        series3.add(4.0, 3.0);
        series3.add(5.0, 2.0);
        series3.add(6.0, 3.0);
        series3.add(7.0, 6.0);
        series3.add(8.0, 3.0);
        series3.add(9.0, 4.0);
        series3.add(10.0, 3.0);

        XYSeriesCollection dataset = new XYSeriesCollection();
        dataset.addSeries(series1);
        dataset.addSeries(series2);
        dataset.addSeries(series3);

        return dataset;
    }
}
```

```

private static JFreeChart createChart(XYDataset dataset) {

    // create the chart...
    JFreeChart chart = ChartFactory.createXYLineChart(
        "Line Chart Demo 2", // chart title
        "X",                // x axis label
        "Y",                // y axis label
        dataset,            // data
        PlotOrientation.VERTICAL,
        true,               // include legend
        true,               // tooltips
        false               // urls
    );

    // get a reference to the plot for further customisation...
    XYPlot plot = (XYPlot) chart.getPlot();
    plot.setDomainPannable(true);
    plot.setRangePannable(true);
    XYLineAndShapeRenderer renderer
        = (XYLineAndShapeRenderer) plot.getRenderer();
    renderer.setBaseShapesVisible(true);
    renderer.setBaseShapesFilled(true);

    // change the auto tick unit selection to integer units only...
    NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
    rangeAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits());

    return chart;
}

public static JPanel createDemoPanel() {
    JFreeChart chart = createChart(createDataset());
    ChartPanel panel = new ChartPanel(chart);
    panel.setMouseWheelEnabled(true);
    return panel;
}

public static void main(String[] args) {
    LineChartDemo2 demo = new LineChartDemo2(
        "JFreeChart: LineChartDemo2.java");
    demo.pack();
    RefineryUtilities.centerFrameOnScreen(demo);
    demo.setVisible(true);
}
}

```

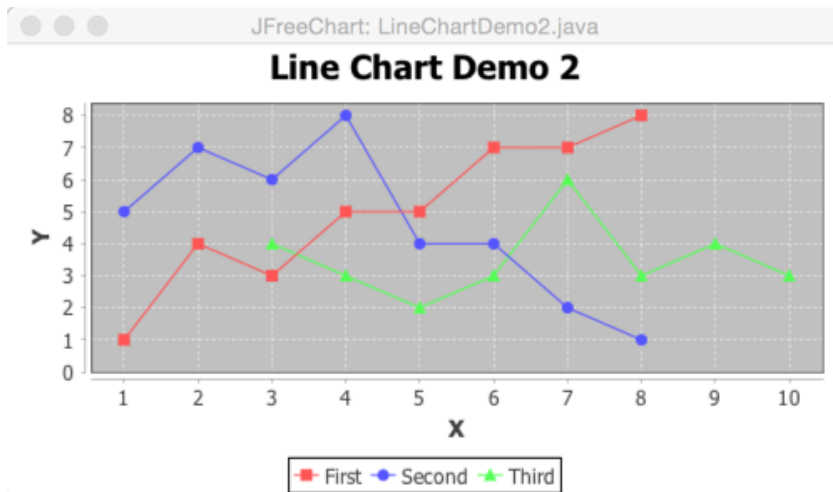


Figure 14.2 Resulting GUI

14.3 TimeSeriesDemo8

Let's now focus on another useful example not coming from the developer guide:

```
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.swing.JFrame;
import javax.swing.JPanel;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.data.time.Hour;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;

public class PlotTimeChart {

    public static void main(String[] args) { // be careful: this is not object oriented
        DateFormat format = new SimpleDateFormat("yyyy-MM-d HH:mm:ss");
        String[] stringDates = { "2015-04-01 00:00:00", "2015-04-01 01:00:00", " →
        ↪ 2015-04-01 02:00:00" };
        double[] value1 = {3,1,2};
        double[] value2 = {-1,2,1};

        Date[] dates = new Date[3];
        try {
            for(int i=0; i<3; i++)
                dates[i] = format.parse(stringDates[i]);

            TimeSeries timeSeries1 = new TimeSeries("value1");
            for(int i=0; i<3; i++)
```

```

        timeSeries1.add(new Hour(dates[i]),value1[i]);

        TimeSeries timeSeries2 = new TimeSeries("value2");
        for(int i=0; i<3; i++)
            timeSeries2.add(new Hour(dates[i]),value2[i]);

        TimeSeriesCollection timeSeriesCollection = new TimeSeriesCollection();
        timeSeriesCollection.addSeries(timeSeries1);
        timeSeriesCollection.addSeries(timeSeries2);

        JPanel chartPanel = new ChartPanel(ChartFactory.createTimeSeriesChart("title", " →
↪ xlabel", "ylabel",timeSeriesCollection,true, true, false));
        JFrame frame = new JFrame("Test");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(chartPanel);
        frame.pack();
        frame.setVisible(true);

    } catch (ParseException e) {
        e.printStackTrace();
    }
}
}

```

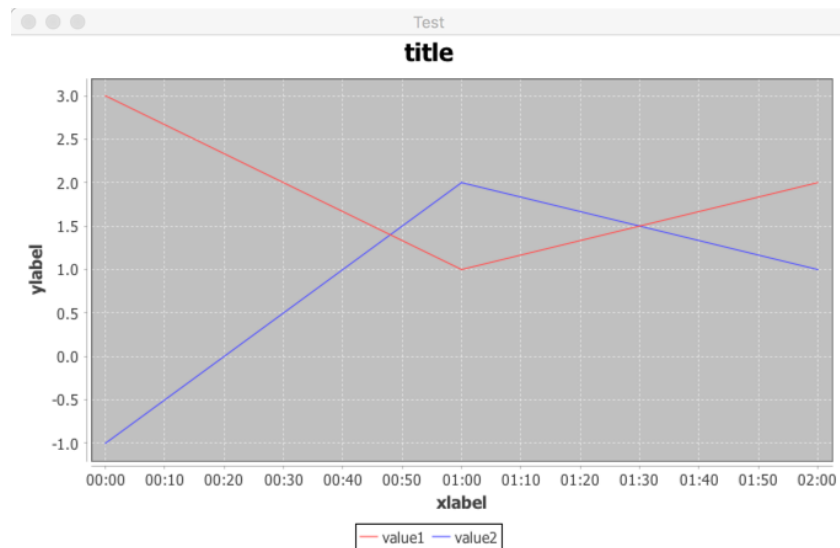



Figure 14.3 Resulting GUI

For additional examples, refer to the Developer guide and the code examples.

What you should remember from this chapter?

- 
- What the different kinds of graphs available?
 - How to to draw a time plot?

Chapter 15

Problem: occupancy estimators

15.1 Problem Statement

You are going to be more autonomous than before to solve this problem. Just remember that your codes have to be object oriented. It means that you have to design the architecture of the code before coding, avoiding sequential solving. Classes of objects have to be coded in the most general way as possible i.e. as independent as possible of the other classes for the sake of readability but also for the sake of re-usability. As a result, your main classes should be almost empty (usually something like `new Class()`) and your class files should be small (200 lines max but smaller is usually better).

15.1.1 General objective

The objective is to develop a tool in Java to support the design of occupancy estimators for an office. Here is description:

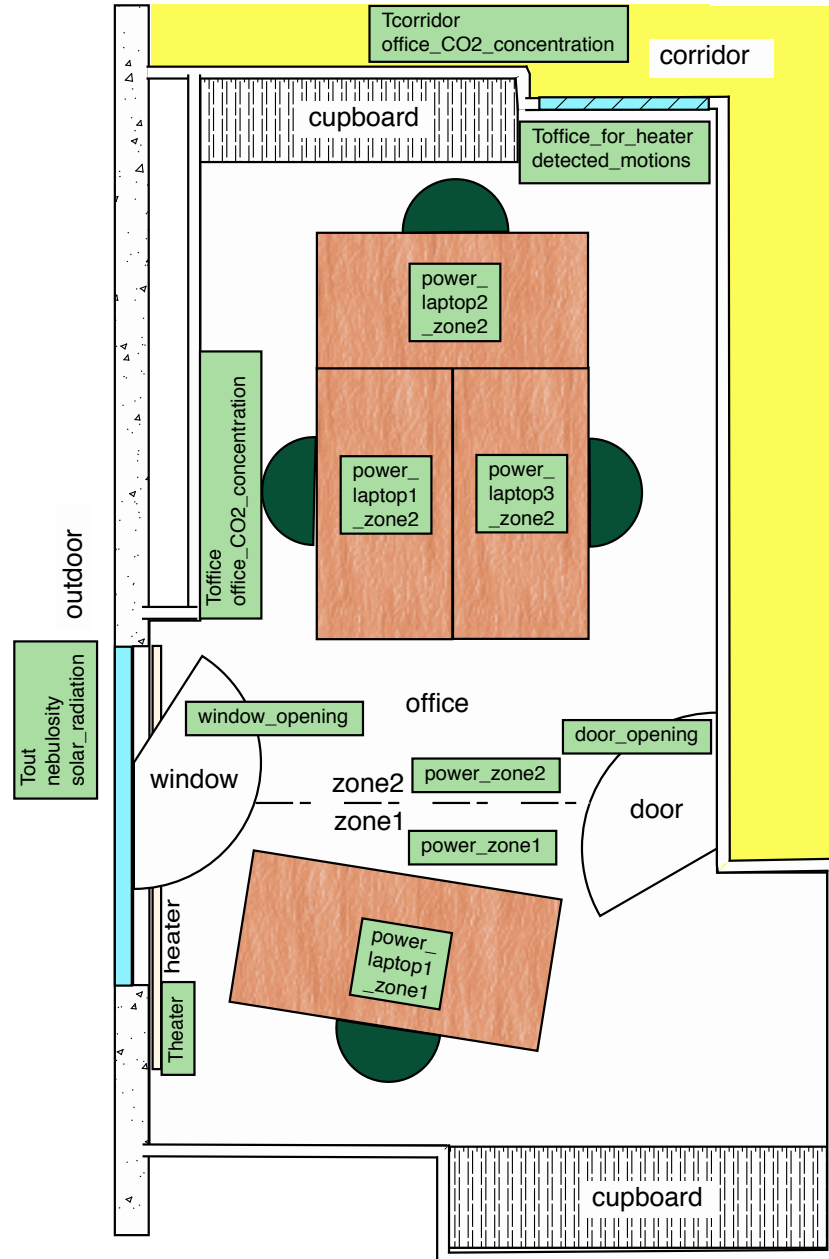


Figure 15.1 Plan of the office under consideration

Sensor names appear into green squares. The corresponding measurements covering the period April, 1st to June, 30th 2015 are given in the zip file ‘occupancy-help.zip’, in file named ‘office.csv’. Data have been sampled with a 1 hour time resolution. The comma-separated values (see https://en.wikipedia.org/wiki/Comma-separated_values) text file format has been chosen because it is easy to read and widespread. Even LibreOffice and Excel can read it. In subsection 15.2.1, an easy way to read

Occupancy estimator based on laptop power consumption

Occupancy estimator based on laptop consumption is very easy to do. The consumption of each one of the 4 laptops is measured by a power meter (variable 'power_laptopX_zoneY'). If the average consumption is greater than the standby consumption (15W), it is considered that someone is working on the computer.

Occupancy estimator based on motion detections

Estimator based on motion detections is also easy to do. It is considered that the number of motions detected within a hour is proportional to the number of occupants in the office. In order to determine the best proportional coefficient, a simple dichotomy can be done, considering the laptop consumption based estimations as the actual occupancies. Scale the coefficient to minimize the error between laptop consumption and motion detection based occupancy estimators. Algorithm is given in 15.2.3.

Occupancy estimator based on CO2 concentration

The CO2 based estimator relies on an air mass balance. Let's call Q_{out} , the renewal air flow exchanged from indoor and outdoor. It can be decomposed into $Q_{out} = Q_{out}^0 + \zeta_{window} Q_{out}^{window}$ where Q_{out}^0 stands for air infiltration, Q_{out}^{window} for the average constant renewal air flow going through the windows when it is opened and ζ_{window} , the window opening. It is worth 0 when it is always closed and 1 when it is always opened during a considered time period. An intermediate value represents the fraction of the period the window was opened. This value corresponds to the variable 'window_opening' of the CSV file.

In the same way, the renewable air flow through the door is given by: $Q_{corridor} = Q_{corridor}^0 + \zeta_{door} Q_{corridor}^{door}$. ζ_{door} corresponds to the variable 'door_opening' whose values are coming from a contact sensor.

Obviously, Q_{out} and $Q_{corridor}$ are time-dependent because of openings ζ_{window} and ζ_{door} .

The air mass balance leads to:

$$V \frac{d\Gamma_{office}}{dt} = -(Q_{out} + Q_{corridor}) \Gamma_{office} + Q_{out} \Gamma_{out} + Q_{corridor} \Gamma_{corridor} + S_{breath} n$$

with:

- Γ , the CO2 concentration and $Q_{\text{out}} \approx 395\text{ppm}$, the outdoor average CO2 concentration
- V , the room volume i.e. 55m^3 for the office
- S_{breath} , the average CO2 production per occupant. It is estimated at $4\text{ppm.m}^3/\text{s}$
- n , the number of occupants in the office

Considering average values over a time period of $T_s = 3600$ seconds, the differential equation can be solved to get a recurrent equation (X_k means average value of X during period $[kT_s, (k+1)T_s)$):

$$\Gamma_{\text{office},k+1} = \alpha_k \Gamma_{\text{office},k} + \beta_{\text{out},k} \Gamma_{\text{out},k} + \beta_{\text{corridor},k} \Gamma_{\text{corridor},k} + \beta_{n,k} n_k$$

with:

- $\alpha_k = e^{-\frac{Q_{\text{out},k} + Q_{\text{corridor},k}}{V} T_s}$
- $\beta_{\text{out},k} = \frac{(1-\alpha_k) Q_{\text{out},k}}{Q_{\text{out},k} + Q_{\text{corridor},k}}$
- $\beta_{\text{corridor},k} = \frac{(1-\alpha_k) Q_{\text{corridor},k}}{Q_{\text{out},k} + Q_{\text{corridor},k}}$
- $\beta_{n,k} = \frac{(1-\alpha_k) S_{\text{breath}}}{Q_{\text{out},k} + Q_{\text{corridor},k}}$

The estimator comes directly from that equation. It yields:

$$\hat{n}_k \left(Q_{\text{out}}^0, Q_{\text{out}}^{\text{window}}, Q_{\text{corridor}}^0, Q_{\text{corridor}}^{\text{door}} \right) = \frac{\Gamma_{\text{office},k+1} - \alpha_k \Gamma_{\text{office},k} - \beta_{\text{out},k} \Gamma_{\text{out},k} - \beta_{\text{corridor},k} \Gamma_{\text{corridor},k}}{\beta_{n,k}}$$

$Q_{\text{out}}^0, Q_{\text{out}}^{\text{window}}, Q_{\text{corridor}}^0, Q_{\text{corridor}}^{\text{door}}$ are constant values that should be determined using a simple simulated annealing optimization algorithm given in 15.2.4. The problem is to find the best values, considering the laptop consumption based estimations as the actual occupancies. Following initial values are proposed:

- $Q_{\text{out}}^0 = \frac{25}{3600} \text{m}^3/\text{s}$
- $Q_{\text{corridor}}^0 = \frac{25}{3600} \text{m}^3/\text{s}$
- $Q_{\text{out}}^{\text{window}} = \frac{150}{3600} \text{m}^3/\text{s}$
- $Q_{\text{corridor}}^{\text{door}} = \frac{150}{3600} \text{m}^3/\text{s}$

Note down and discuss the 4 estimated constant air flows.

15.2 Clues

15.2.1 Reading a comma-separated values (CSV) from Java

Here is a simple example (it cannot be used in this way because it is not object oriented):

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadCSV { // be careful: this is not object oriented

    public static void main(String[] args) {
        String fileName = "office.csv";
        BufferedReader bufferedReader = null;

        try {
            bufferedReader = new BufferedReader(new FileReader(fileName));
            String line;
            while ((line = bufferedReader.readLine()) != null) {
                String[] tokens = line.split(", ");
                for(String token: tokens)
                    System.out.print(token + ", ");
                System.out.println();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        } finally {
            if (bufferedReader != null) {
                try {
                    bufferedReader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        System.out.println("Done");
    }
}
```

15.2.2 Possible architecture for the application

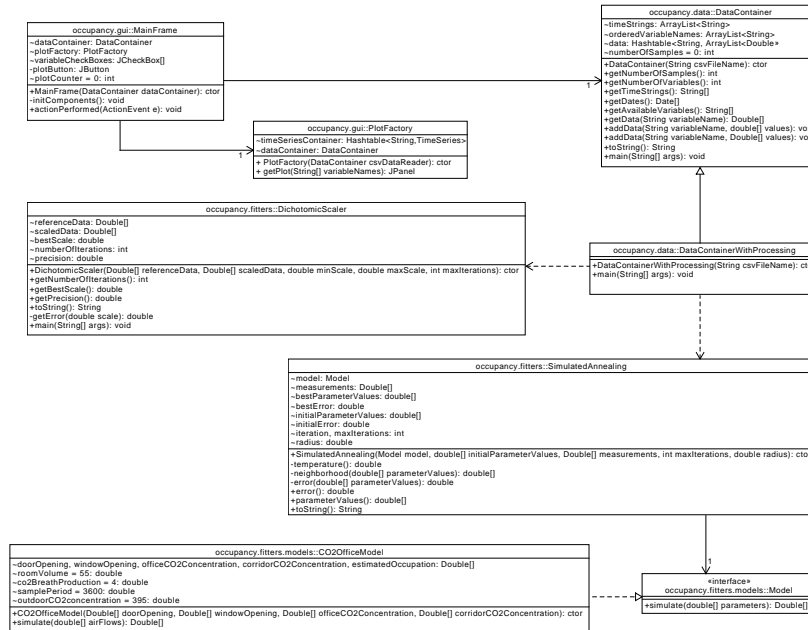


Figure 15.3 Class diagram of a possible architecture for the application

15.2.3 Dichotomic search algorithm

Finding the minimum of a 1-variable convex function can be done by dichotomy. Consider a function $f(x)$ defined on $[m, M]$. The algorithm is given by:

```

 $l \leftarrow m$  [ $l$  for lower]
 $u \leftarrow M$  [ $u$  for upper]
for  $i \leftarrow 0 \dots i_{max}$  do
   $c \leftarrow \frac{l+u}{2}$  [ $c$  for central]
  if  $f(l) < f(u)$  and  $f(c) < f(u)$  then
     $u \leftarrow c$ 
  else if  $f(c) < f(l)$  and  $f(u) < f(l)$  then
     $l \leftarrow c$ 
  else
    break
  end if
end for
 $x^* \leftarrow c$ 
precision  $\leftarrow \frac{u-l}{2}$ 
  
```

15.2.4 Simulated annealing algorithm

Simulated annealing (https://en.wikipedia.org/wiki/Simulated_annealing) is an multi-dimensional improvement optimization algorithm with diversification. Let's consider a function $f(X)$ of n variables $X = x^0, \dots, x^{n-1}$ that must be minimized. The algorithm performs random jumps in a neighborhood. Let's define the neighborhood random jump function $J(X_k) \rightarrow X_{k+1}$:

```

for  $i \leftarrow 0 \dots n-1$  do
     $x_{k+1}^i \leftarrow x_k^i (1 + (2 \times \text{random}(0, 1) - 1)r)$ 
end for

```

where $r \in (0, 1)$ is the radius defining the neighborhood as a fraction of the current variable value x_k^i , k is the current iteration and $\text{random}(0, 1)$ is a random value between 0 and 1.

Then, if the new variable values lead to a better $f(X_{k+1})$ result, it can be kept for a next random search and forget otherwise.

But simulated annealing is more than that. Always trying to improve a result can lead to local minimal. To get around this problem, a diversification mechanism is added. It means that sometimes, even if the result is worse, it can be kept, expecting latter better results. A temperature function is used to allow more worse results at start than at the final iterations. Therefore, the temperature must decrease with the iteration number. It must also decrease if the current best value for $f(X)$ is decreasing. We suggest the following simple temperature function $T(k, f^r) = \beta f^r \frac{k_{max}-k}{k_{max}}$ where β is a freely tunable parameter that impacts the convergence ($\beta = 1$ is a good start). Note that f^r is the current reference function for stepping and f^* is the current minimum function value.

The simulated annealing algorithm can then be given:

```

 $X^c \leftarrow X_0$  [ $c$  for candidate]
 $X^r \leftarrow X_0$  [ $r$  for reference]
 $X^* \leftarrow X_0$  [ $*$  for best]
for  $k \leftarrow 0 \dots k_{max} - 1$  do
     $X^c \leftarrow J(X^r)$ 
    if  $f(X^c) < f(X^*)$  then
         $X^* \leftarrow X^c$ 
    end if
    if  $\text{random}(0, 1) < e^{\frac{f(X^r) - f(X^c)}{kT(k, f^r)}}$  then
         $X^r \leftarrow X^c$ 
    end if
end for

```