University of Central Florida

Engineering Data Structures
EEL 4851

JAVA
LABORATORY MANUAL
(Revised edition, 2002)

Maureen Murillo
Avelino González

EEL 4851 – Engineering Data Structures

JAVA Laboratory Manual

Introduction to Java – the Language

Java has quickly emerged as one of the most popular languages of the late 1990's. It has transitioned from a special-purpose language used in Internet applications to a general purpose one used in many cases where C, C++ were once used. This laboratory manual will teach the student the basics of Java, both as a general-purpose language as well as the Internet language. It includes this Introduction Section and 12 laboratory sessions. The most important part of learning any computer language is to use it. This manual forces the reader to do just that. If the reader is not able or willing to implement the laboratory sessions, then he/she should look for an alternative learning instrument.

What has made Java so popular is its ability to run on any hardware platform. Other high-level languages that have become standardized (C, C++, Lisp, Fortran and several others) do in fact have standard source code that, when compiled with a compiler specially designed for a particular hardware platform and operating system, it will run on that platform. When the program is to be ported to another platform, the source code must be re-compiled with a compiler for the new platform. While this avoids having to rewrite the program in a different way if the language were not standard, in practice, however, a long and complex program written in one standard language will require some modifications when compiled for another platform and/or operating system. But object code compiled under one compiler will not run on a platform for which that compiler was not designed. This is because the microprocessors and/or the operating system are designed such that they do not operate exactly the same way. But Java object code will run on (almost) any microprocessor. This is important from the standpoint of the Internet, where one does not always know the type of computer platform and operating system on which an application will be asked to run.

Java originated from an effort in the early 1990's at Sun Microsystems to write a language that could be run on all home appliances, such as microwave ovens, VCR's, etc. This is actually a difficult task, as the microprocessors found on such devices are inexpensive and of many variations. Thus, it had to be compatible with many processors. A few years later, it was determined that the approach taken by the developers of Java would be more useful in an Internet environment. The rest is history.

Java embodies portability by "compiling" the source code, not to machine language, but to *byte code*. Byte code is an intermediate level representation that represents the way in which most microprocessors do things. Such a representation is called the *Java Virtual Machine*, and is similar to most common microprocessors. Each family of microprocessor (Intel, Motorola, etc.) has a specific way in which they do things – their instructions, memory organization, registers, etc. However, as a whole, they do things much more similarly than they do differently. Java takes advantage of this fact. The byte code is actually representative of most microprocessors – not exactly any one of them, but very similar to all of the most common ones. Thus, only one Java compiler is necessary, as its output will be the same (byte code) regardless of on which machine the Java code will be executed.

But, of course, byte code will not by itself run on *any* microprocessor. It requires an additional step – the *interpretation* step. As the byte code is executed, an interpreter will interpret every instruction and convert it into the actual machine representation for that platform. The advantage is that while compilers are highly complex and expensive systems, byte code interpreters are much easier to write and maintain, since most of the work is already done by the Java compiler. If every computing platform has its own Java byte code interpreter, then Java object code can be run seamlessly.

The drawback is that being an essentially interpreted language, Java can be slower than the fully compiled and optimized C and C++ programs. Other traditionally interpreted languages such as Basic and Lisp lost popularity when commercial applications were being developed partially due to their interpreted nature resulting in slow execution. However, the Java interpretation is faster than that of those older languages because the line of source code is not being interpreted, but rather, an intermediate, and rather low level implementation.

The Java syntax is suspiciously similar to C and C++. In fact, Java is easy to understand if you already know those languages. However, there are several significant differences:

- Java is completely object-oriented. C++, although it allows (and encourages!) the use of classes, does not require the programmer to do so. Java does.
- Java does not allow pointers. While this greatly reduces the power of the programmer to dwell at the memory levels, it does provide a level of safety and a program structure that is better in the end. Students everywhere cheer this aspect of Java after they learn C/C++.

This laboratory manual is written with the assumption that the student already knows C++ to the point that he/she can write non-trivial programs in C++.

There exist two types of Java programs: *applications* and *applets*. Applications are standalone programs such as would be written in C and C++. This is where the general-purpose aspect of Java comes in. Programs that were previously written in C and C++ can now be written wholly in Java, notwithstanding the accompanying reduction in execution speed.

Applets are similar to applications, but they don't run standalone. Instead, applets adhere to a set of conventions that lets them run within a Java-compatible browser (for example, Netscape or Internet Explorer). This is the special purpose nature of Java (for which it was intended originally).

In this course we will cover both of these aspects of Java. We will begin by showing you how to write and run a simple Java standalone program. This is the same as the “Hello World” that is commonly run in C/C++ courses as the first example. It will teach the student how to write a simple Java program and what the different lines of code mean. In next chapters we will show you the basic differences about the main programming aspects between Java and C++. In chapter 9, we will cover simple applets, with some GUI components. The last chapter introduce you in the topic of data structures, showing you how to build a stack, a queue, a list and a binary search tree.

Chapter 1

Introduction to Java Programming

The "Hello World" Application that you are going to create leads you through compiling and running a stand-alone application-- a Java program that executes independently of a browser. The lab also introduces some general Java techniques: how to define a class and how to use supporting classes and objects.

This lab describes how to create the application using the Java *Development Kit* (JDK). Let's first run a simple application – "Hello, World" in Java – and then compile and run it. Don't worry about what the statements mean – we shall deal with that a bit later. Just write it, compile and run it. Java is case-sensitive, so make sure you copy the code exactly as indicated below.

1. Creating a Java source file

Using a text editor, create a file named **HelloWorldApp.java** with the following Java code:

```
/**
* The HelloWorldApp class implements an application that
* simply displays "Hello World!" to the standard output.
*/
class HelloWorldApp {
        public static void main(String[] args) {
           System.out.println("Hello World!"); //Display the string.
        }
}
```

2. Compiling the source file

Compile the source file using the Java compiler. Type in the command line of the operating system:

```
javac HelloWorldApp.java
```

If the compilation succeeds, the compiler creates a file named **HelloWorldApp.class** in the same directory (folder) as the Java source file written above. This class file contains Java byte codes, the platform-independent representation interpreted by the Java runtime system.

If the compilation fails, make sure you typed in and named the program exactly as shown above, using the capitalization shown.

3. Running the application

Run the program using the Java interpreter. You should see "Hello World!" displayed. Type in the command line:

```
java HelloWorldApp
```

The argument to the Java interpreter is the name of the class to run, not the name of a file. In this simple example, they are the same, but this may not be the case other times. Be sure to capitalize the class name exactly as shown above.

4. Understanding your "HelloWorld" application

Now that you've written, compiled and run a simple Java application, let's discuss what you did so that you might better understand it. What you wrote is a Java application – a stand-alone Java program. This is a program that is capable of running on a computer independent of any other program such as an Internet browser. This is equivalent to a C++ program.

4.1. Comments in Java code.

The Java language supports three kinds of comments:

1. **/* text */**
2. **/** documentation */**
3. **// text**

In the first one, the compiler ignores everything from /* to */. This is as in C.

The second one, a slight variation of the first one, indicates a documentation comment (*doc* comment, for short). The compiler ignores this kind of comment, just like it ignores comments that use /* and */. The JDK *javadoc* tool uses doc comments when preparing automatically-generated documentation.

The third comment format directs the compiler to ignore everything from // to the end of the line. This is as in C++.

4.2. Defining a class.

In object oriented terminology, a function that is a member of a class is called a *method.* Since Java is completely object-oriented and stand-alone functions are not permitted, all functions are referred to as methods. Furthermore, all variables exist within a class or an object (an instance of a class). The Java language does not support global functions or variables. Thus, the skeleton of any Java program is a class definition.

A class is the foundation of any object-oriented language such as Java. A class is a template that describes all associated with the instances of that class – data members as well as its behavior indicated by the methods. When you instantiate a class you create an object that resembles other instances of the same class, although with different values.

Defining a class in Java is very similar to C++:

```
class name {
    . . .
  }
```

As in C++, the *class* keyword begins the class definition of a class named *name*. All variables and methods are defined within the brackets that form its definition. The "Hello World" application has no variables and only one single method named **main**.

4.3. The *main* method.

The entry point of every Java application is its **main** method. This is the equivalent of the **main()** function in C/C++. When running an application program, the name of the class to be run must be specified. The interpreter invokes the **main** method defined within that class. The **main** method kicks off the execution, and runs any other methods defined in the program as needed. Thus, to begin the Java execution, the name of the class that contains the **main** method must be invoked. If the class invoked does not contain the **main** method, the Java interpreter will not execute the program. Once again, it is basically running the **main()** function in C/C++, except that in Java, it must be contained within a class.

Therefore, every Java application must contain a **main** method. Its format is as follows:

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

- As in C++, the *public* keyword means that any object in the program can call the **main** method.

- The keyword *static* indicates that the **main** method is a *class method*. This means that the method does not change objects of its own class – yet, the method must be invoked tied to the class and not to an object of that class.

- The keyword *void* means, similarly to C/C++, that the main method doesn't return anything.

The **main** method accepts a single argument: an array of elements of type String.

```
…. main(String[] args)
```

This array is the mechanism through which the runtime system passes information to your application. Each String in the array is called a command-line argument. Command-line arguments let users affect the operation of the application without recompiling it. For example, a sorting program might allow the user to specify that the data be sorted in descending order with this command-line argument:

-descending

This application does not make use of command line arguments, so this is not a good example with which to explain this concept. It will be covered at a later time.

4.4. Using classes and objects.

But the **main** method, while necessary, should not be the only one used in a Java program. This would be like writing a C program with only the **main()** function. While possible, it is not really desirable for large, useful programs. The "Hello World" application does use another class, the *System* class. This class can be found in the API (*application programming interface*) provided with JDK.

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

The construct *System.out* is the full name of the *out* variable in the *System* class. Notice that the application never instantiates the System class and that *out* is referred to directly from the class name. This is because *out* is a class variable--a variable associated with the class rather than with an instance of the class. You can also associate methods with a class--*class methods*.

To refer to class variables and methods, you join the class name and the name of the class method or class variable together with a period (".").

Methods and variables that are not class methods or class variables are known as instance methods and instance variables. To refer to instance methods and variables, you must reference the methods and variables from an object.

While System's out variable is a class variable, it refers to an instance of the PrintStream class (a class provided with the Java development environment) that implements the standard output stream.

When the System class is loaded into the application, it instantiates PrintStream and assigns the new PrintStream object to the out class variable. Now that you have an instance of a class, you can call one of its instance methods:

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

As you can see, you refer to instance methods and variables similarly to the way you refer to class methods and variables. You join an object reference (out) and the name of the instance method or variable (println) together with a period (".").

5. Writing your own program

Now that you have a very basic idea of how to write a Java program, let’s get you to practice writing your own simple program. The specification of this program is to request input from a user. The program should first display:

```
What is your name?
```

The user then types in this/her name from the keyboard, and the systems captures it.

```
Jennifer
```

Then, knowing the user's name, it should ask how old the user is.

```
How old are you, Jennifer?
```

Jennifer then states she is 21.

```
21
```

When the user enters his/her age, the program simply says "*Thanks a bunch*". In the case of Jennifer,

```
Thanks a bunch.
```

Then exits.

In order to do this, you will need to use an input class. This input class has been loaded into your hard drive, and it is called *MlmIn.* This input class is not a primitive class in Java, but was written by the authors specifically for this book. Java has no primitive class to do simple inputs. This is another disadvantage when compared to C/C++, which enjoy many primitive input functions.

As you use the command *#include* in C++ to employ libraries and to increase the functionality of your program, in Java you have to use the command *import* followed by the name of the class and a semicolon in order to utilize supporting classes or methods of them.

Some of the input methods that this class *MlmIn* has are: *readChar(), readInt()* and *readString()*. These methods are static, so you can call them by typing the name of the class, followed by a period and by the name of the method. The result of each method should be assigned to an appropriate variable.

You may need to know that Java has a primitive type called *int* (for integer variables) and the type *String* (for lines of text). This last type implements a whole class that facilitates the manipulation of text in Java, in contrast with the necessity in C++ of declaring a character array in order to store a string. To declare a variable, just write the type of the variable and then the name you select for your variable.

The last aspect of Java you may need in order to implement this program is that the method for displaying information on the screen can accept more than one parameter, for example, a string of text and a variable. All the parameters within the parenthesis should be concatenated with the symbol +. For example,

```
System.out.println("The product of " + n1 + " and " + n2 + " is " + n1*n2);
```

where *n1* and *n2* are integer variables.

Chapter 2

Classes and Objects in Java

As mentioned in Chapter 1, Java shares much of its syntax with C++. This is good if you are a C++ programmer. Bad if you are not. However, Java seems to make programming easier than with the overly complex C++ syntax. This chapter will discuss how to create classes in Java, and how it is similar or different from C++.

1. Class definitions in Java

You have already seen how to create a class in Java. As you already should know, the basic structure of a class is:

```
access class name {
   ...
 }
```

where *access* is the optional label `public`, and the *name* is any label chosen for the class by the programmer. This is very similar to C++ except for the existence of the *package* class access. This is the topic of discussion here.

`Private` or `protected` access does not make sense for classes, so they can only have either `public` or `package` access. `Public` access in a class means that it can be used by methods of any other object or class in the entire program. This type of access must be declared explicitly with the modifier `public`. Classes without the `public` access modifier can only be used by classes or objects within the same package (a package is a collection of classes grouped under the same name). This is the `package` access type.

Therefore, the default access is `package.` The `public` access specifier must be explicitly used whenever `public` access is desired. Yet, an explicit `package` access label is illegal. If `package` access is desired, you must leave the access label field blank. A Java source file can only have one `public` class (at most), and as many `package` classes as necessary. Remember that the name of the file and the name of the `public` class must match exactly.

We will next discuss how to define data and procedural members in Java classes.

2. Defining data members in a Java class

There are two types of variables in Java:

1. Primitive variables
2. Object variables

Primitive variables are simply the variables that we all know and love from other programming languages. They are of a *primitive* data type. The primitive data types in

Java are the same as those in C++ - c*har* for character variables, *int* for integer variables, and *float* and *double* for floating point numbers. Object variables, on the other hand, are data members of one class whose type is another class.

You also should know by now that everything in Java is based on classes. A class, as in C++, consists of data members and procedural members (methods).

But, unlike C++, where all `public` or `private` members are in the same pre-labeled "section" of the class, the Java class members must specifically and individually indicate whether they are *`public`*, *`private`* or *`protected`*. Lack of an access specifier indicates that the data member or method's access is `package`.

The meaning of these terms is very similar to those in C++. `Public` data members may be accessed and modified by methods in any other object or class. `Public` methods may be called from within any other class or object. As in C++, data access methods are typically `public` to allow outside objects to access the value of data members. They are very visible to any user. `Private` members, on the other hand, limit their visibility to members of its own class. As in C++, data members are typically `private`, while some methods may be also labeled as `private`.

`Protected` members represent a compromise between the two and are applicable in the case of inheritance, where only derived classes can access the protected members of the parent class. This will be covered later in the inheritance section. Classes in the same package than the class with protected members can also access them, even if they are not derived classes.

Data members must be declared, just as in C++. Therefore, a label indicating the type of data it represents must always precede the variable name. Thus, Java, like C and C++, is a strongly typed language. Preceding the type of the variable, you can write the kind of access you desire for the variable (`public`, `protected` or `private`). The default access of a member if you don't indicate it is the `package` access, which allows visibility of the variable only to other classes in the same package.

As in C++, a `static` data member is a variable that is shared by all the instances of the class. It's like a global variable for all the objects of the class. So it can be used without making reference to a specific object.

Write a short Java program that has one class that has the main method and two public data members, *month* and *day*. Both variables are integers. They should have a value assigned within the class itself (we will complicate that a bit later). Assuming that every month has 30 days, calculate the day of the year that a particular date consisting of day and month represents. Print out "The day of the year is *xxx*" using **System.out.println().**

Because the main method must be static (i.e., it is a *class method* and cannot be called tied to an instance of the class), it cannot make reference to nonstatic data members. So in order to make the program to work, you have to declare the data members of the class as static variables and initialize them in the same statement where you declare them. For example:

```
public static char letter = 'M';
```

3. Constants in Java

Just as constants can be defined in C++ by preceding the variable with the word **const**, the same can be done in Java. The way to do it is by preceding the declaration of the variable with

```
public static final
```

This means that the value of the constant cannot be changed. `Public` and `static` are not strictly necessary, but it is common to use them to denote public access and static scope.

4. Defining methods in a Java Class

Methods in Java are the member functions in C++. Unlike C++, however, which encourages the separate definition of the member functions, Java requires that the methods be defined inside the class definition. This eliminates the need for the scope definition operator commonly used to define member functions in C++ outside the class definition brackets.

As in C++, the methods must have an access specifier. Typically, methods are `public`, but they may be `private` or `protected` in some cases.

The format of a method, as we saw in the previous chapter, is as follows:

<access> <method flag - optional> <returns> <name> (<parameters>)

- As in C++, the *access* keyword indicates who can access that method. It is typically set to `public` for a method. The differences between the distinct access modifiers (`public`, `private` and `protected`) for methods are the same than those for the data members, including the fact that if you don't specify the access keyword the default access is `package`.
- The *method flag* is optional and when `static` is placed therein, it will indicate that the method is a *class method.* Class methods cannot operate on objects, and instance methods must only operate on objects. Since numbers are not objects, they must be manipulated only by class methods. If static is not used, then it is assumed that the method is not a class method.
- The *returns* field indicates what the method returns. This is similar to functions defined in C and C++. *void* means that the method doesn't return anything.
- The *name* field is the label for that method.
- The declaration of *parameters* works in the same way than in C++. You declare a list of parameters separated by comas and where each parameter is preceded by its type.

To the class defined above, add a public method that calculates the day of the year and sets the value of a third variable, a `float` called *`doy`*, to the result. Then print out the value of that variable. Again, because the main method is static, it can only make reference to static data members and to static methods. So you need to declare the second method as static.

5. The `new` operator

The new operator makes an instance (an object) of a class. Of course, the class must be already defined, and a variable of the type of the class must already be declared.

```
<class name>  <object name>;
<object name> = new  <class name> ( );
```

This can be alternatively done in one line as follows:

```
<class name>  <object name> = new  <class name> ( );
```

But this is more important than it seems. Variables of a primitive type are locations in memory where the value of the variable is held. However, an object instantiated by `new` does not hold the object itself, but the address in memory where the object is located. Yes, a pointer! This is very similar to the operator of the same name in C++ that returns a pointer to a dynamically allocated block of memory.

Define a class called `DayOfYear` that contains the data members and the method to compute the day of the year as done previously in the **main** method. In this case, define the class and instantiate an object of the same class from the **main** method. After instantiating it, call its method that calculates the day of the year and that prints it out. This method and the data members of the class have to be nonstatic because now you are going to instantiate the class in the **main** method and you will call the method that calculates the result tied to the object.

6. Constructors

The theory of constructors in Java is basically the same than in C++. Their purpose is to initialize the data members of the class. If the constructor does not have any parameters, it is called the *default constructor*.

A class can have more than one constructor, and the name of all of them must match the name of the class. As in C++, the constructors don't return any value and you don't have to include the word `void` in its definition.

A difference with C++ is that the constructor of a class is called in Java only when you use `new` to create an instance of the class. For example:

DayOfYear obj = new DayOfYear ( ) ;

Note that if you don't define any constructor for a class, Java calls a default constructor that does nothing, but it allows the creation of the object, like in the previous example. But if you decide to define at least one constructor for a class with at least one parameter, you have to define the default constructor too, because if you try to use the default constructor Java does not use its default one. If you write any constructor, Java assumes that now you are in charge of all of them, including the default one.

For the class `DayOfYear` defined before, implement a default constructor that asks the user for the day of the month and for the month. The rest of the program is the same, it should print the day of the year.

7. Method Overloading

Overloading of methods is the same as overloading of functions in C++. It depends on the arguments passed to it, the number and the type. This applies to the constructors of a class, which can have more than one constructor.

Include in the class `DayOfYear` another constructor that receives the day and the month as parameters and that initializes the data members. In the **main** method declared two objects. The first one has to use the later constructor and initializes its members with the last day of year (December 31). The second object has to be initialized with the values entered by the user. Display both days of the year.

8. Call-by-value and Call-by-reference

Unlike C and C++, Java does **not** give the programmer the option of how to pass the variables to a method. It basically dictates that primitive variables are always passed by value, while object variables are always passed by reference. This is in agreement with the fact that primitive variables are locations in memory that contain the value of the variable, while object variables are addresses of where the variables are in memory. Arrays are treated as objects by Java, and thus, also passed by reference.

Type the following code, run it and see what happen.

```
import MlmIn;

public class test
{
  public static void main(String[] args)
  {
    int num1 = 4;
    myobj num2 = new myobj();

    System.out.println ("Value of num1 before: " + num1);
    System.out.println ("Value of num2 before: " + num2.data1);
    change.add(num1, num2);
    System.out.println ("Value of num1 after: " + num1);
    System.out.println ("Value of num2 after: " + num2.data1);
  }
}
```

```
class myobj
{
   public int data1 = 3;
}

class change
{
  public static void add(int x, myobj y)
  {
        x = x + 2;
        y.data1 = y.data1 + 2;
  }
}
```

Chapter 3

Control Structures and Memory Management in Java

This chapter deals with defining the control structures in Java. Since they are very similar to those of C and C++, this chapter constitutes a review and a confirmation of them. Related to the control structures, we will mention different types of operators that are usually necessary in the construction of control structures. Finally, we will cover a topic that usually constitutes a source of errors in a program: pointers, and how to finalize objects.

1. Operators

Relational operators:

< : less than
<= : less than or equal to
> : greater than
>= : greater than or equal to
= = : equal to
!= : different to

Logical operators:

&& : and
|| : or
^ : exclusive or
! : not

Arithmetic operators: + , **-** , ***** , / and % (modulus).

Assignment operators: += , **-=** , ***=** , /= and %= . These operators carry out the specified operation and the assignation in a single step. For example,

num += 2;

assigns to the variable *num* the addition of the value in *num* and the value 2.

Incremental and Decremental operators: let *num* be an integer variable,

++num : pre-increment
num++ : post-increment
--num : pre-decrement
num-- : post-decrement

2. if/else – Selection structure

The **if-else** structure works in the same way than it does in C++. Its format is:

```
if (condition) {
    ...
 }
else {
    ...
}
```

The **condition** is a logical expression that returns true or false and has to be written within parenthesis. The **else** section of the structure is optional and it works as the default selection. You can have nested **if-else** within the body of any other **if-else**. The braces "{ }" are necessary if the body of the **if** or the **else** is compound of more than one statement.

There is another way to implement a simple **if-else**, and it is with the operator **? :** called **ternary operator**. This operator has three operands: the first one is a boolean expression, the second one is the statement to be executed if the logical expression returns true, and the third one is the statement to be executed if the expression results false. See the following example, assuming that *grade* is an integer variable:

```
System.out.println (grade >= 7 ? "Approved" : "Fail");
```

For the class `DayOfYear` that you defined in the previous chapter, implement a private method that checks whether the date given by the user is valid. Use in the implementation of the method the selection structure **if-else**. The return type of the method has to be `boolean`, i.e., it has to return `true` or `false`. This method should be called from the default constructor. If the date entered by the user is invalid the constructor has to display a message indicating it. This will be improved in the next sections of this chapter.

3. switch – Selection structure

The **switch** structure is an alternative way of implementing an **if-else**. It is especially useful in showing clearly the logic of the program when you have many alternatives in the selection. Its format is:

```
switch (variable) {
     case value1 :
             ...
        break;
     case value2 : case value3 :
             ...
        break;
     ...
     case valueN :
        ...
        break;
     default:
        ...
        break;
}
```

Notice that the body of each case doesn't need braces to delimit its beginning and its end, but it needs a **break** at the end of each one, otherwise the other cases could be evaluated simultaneously. Another interesting element is that you can combine more than one case in the same line if they have the same body to be executed, for example in the case with *value2* and the case with *value3*. The **default** case is optional and it would be executed if none of the other cases were evaluated to true.

We will combine the **switch** with the **for** structure in the next section to improve the class `DayOfYear`.

4. for – Repetition structure

The **for** structure doesn't have any change respect to C/C++. Its format is:

```
for  (initialization_control_var;  boolean_expression;
     update_control_var) {
    ...
}
```

An example of this structure is:

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

Improve the accuracy of the method that calculates the day of the year in the class `DayOfYear`, including a **for** loop that goes from 1 (January) to the month before the month specified by the user in order to accumulate the correct number of days for each month. Inside the **for** loop you have to include a **switch** structure that selects the number of days to add depending on the month.

5. while – Repetition structure

The format of the **while** structure is:

```
while (condition) {
      ...
}
```

With the **while**, the body of the loop will be executed as long as **condition** is true. It could happen that the statements in the body will never be executed if the condition is false since the first time that is evaluated.

6. do/while – Repetition structure

The **do-while** structure is very similar to the **while**, except that the condition is evaluated at the end of the loop, what causes that the body of the loop will be executed at least once. The repetition finishes when the condition is false. Its format is:

```
do {
      ...
}
while (condition);
```

Modify the default constructor in the class `DayOfYear`, and include a repetition structure that keeps asking to the user a date until he/she enters a valid date. You can use either a **while** or a **do-while**. Remember that the **do-while** executes its body at least once.

7. Pointers and finalization of objects

When we talk about pointers in Java we could say that they don't exist in this programming language or we could say that everything in Java is a pointer. The reason for this duality is that all the object variables in Java are references to object values stored in other memory locations. This contrasts with C++, where the object variables hold object values, and if we want a variable to hold the address of an object value we have to specify it explicitly with the pointer symbol *. An object variable being an address to an object in Java, we don't have to worry about pointers, we are manipulating them implicitly when we work with the object variables. This is the reason why we can say that the pointers don't exist in Java, because in practical terms they don't exist for the programmer.

An important issue at this point is how to free all the memory allocated to object variables. Well, this is simpler in Java than in C++. In C++ you need take care of all the objects you create, and destroy them when they are not longer needed (usually through the destructors), in order to free all the system resources they have assigned. In Java you don't have to worry about this. Java has a **garbage collector** that is called automatically by the system, and whose task is to free the memory that objects that are no longer needed have. The objects that are candidates for garbage collection are those that the system determines that go out of scope in the program, and those that the programmer explicitly sets to null.

The garbage collector has another useful characteristic. Sometimes as part of the destruction of an object you may wish to free other system resources besides memory such as files, or you want to mark other objects for garbage collection. You can do this through the method **finalize** that is called automatically. The purpose of this method is to clean up the object before the garbage collector destroys it. The method **finalize** is inherited by all the objects from the class call `Object`, which is the parent of all the classes in Java, so you can override it for a specific class. This method must be declared as follows:

```
protected void finalize() throws Throwable {
     ...
     super.finalize();
}
```

It is recommended to include the last line of the method (super.finalize( );), because this cleans up all the resources that the class may have obtained through inheritance. A negative aspect of the garbage collector and the method **finalize** is that you never know when they are going to be executed, or even worse, you don't know if they will be executed.

Chapter 4

Inheritance in Java

Inheritance is a powerful technique for designing and reusing code, which makes the programmer's work easier when it is well understood by her/him. Inheritance is an element of object-oriented programming that is implemented by the programming languages in different ways. Java implements it by mean of extending classes and using interfaces as a way of superseding the multiple inheritance that Java doesn't support. This chapter deals with these issues and others related to it within the scope of the Java language.

1. Extending a class

Inheritance in Java is made through the extension of a class. You can find in different sources of information different terminology, depending on the author, to make reference to the elements involved in inheritance. For example, the class that is extended is called *base class*, *parent class* or *superclass*, and the class that is extending another class is called *derived class*, *child class* or *subclass*. From this chapter on we will use the terms *superclass* and *subclass*, to be consistent with the syntax of Java.

The basic structure to extend a class is shown in the following line:

```
access class subclass_name extends superclass_name {
        . . .
}
```

All classes in Java are the extension of another class, even if they don't include the term *extends*. In such a case, the class is an extension of the class `Object`, which is the root of the hierarchy of classes in Java.

You can see that in Java there is only one type of inheritance, you don't have to specify any inheritance type as you do it in C++. In C++ you would declare a subclass following the format:

```
class subclass_name : public|protected|private superclass_name {
...
}
```

The behavior of a subclass in Java is the same as a subclass with public inheritance in C++. This means that the **public members** of the superclass are inherited as public in the subclass; the **protected members** of the superclass remain protected in the subclass, and the **private members** of the superclass are accessible in the subclass only through the use of non-private methods of the superclass. Finally, with respect to the additional type of access in Java, the **package members** of the superclass are accessible in the subclass only if both classes (subclass and its superclass) are in the same package.

Define a class `ECE_Person` that has:

Private data member: `SSN` (Social Security Number)
Protected data member: `name`
Public methods: `setName`, `setSSN`, `getName`, `getSSN`

Define a class `Student` that inherits from `ECE_Person` and that has:

Private data member: `GPA`
Protected data member: `major`
Public methods: `setGPA`, `setMajor`, `getGPA`, `getMajor`, `displayInfo`

Define a class `Staff` that inherits from `ECE_Person` and that has:

Private data members: `salary`
Public methods: `setSalary`, `getSalary`, `displayInfo`

In the `main` method, make an instance of the class `Student` and another instance of the class `Staff`. Set the values of the objects asking the user to enter the information. Then, display the information of both persons.

2. Inheritance of constructors

An instance of a subclass is an object of both the subclass and all the superclass(es) in the hierarchy. So when you declare an object of a subclass, Java executes the constructor of the subclass and the constructor(s) of the superclass(es). Remember that a class can have more than one constructor. If you don't call any specific constructor, Java will call the default one. To explicitly call a superclass' constructor that is not the default, you must include the call to that constructor in the definition of the subclass constructor. This must be done by including the call as the first line of the body of the subclass' constructor. You can use the word `super()` to make reference to the superclass, with the parameters of `super()` providing the determination of the correct constructor. If no parameters are included, then it will call the default constructor. You might ask the obvious question: "why explicitly call the default constructor when it will be called anyway?" – For code readability. For example,

```
class Student extends Person {
   ...
   public Student()   //default constructor of Student
   {
      super(); //call to the default constructor of Person
      ...
   }
}
```

The call to the superclass' constructor must be the first line in the subclass' constructor. If you include it in another line, you will get an error. You can call any constructor of the superclass, not necessarily the default one. But if you don't call any of the superclass' constructors at all, Java will call the default one.

Now define the constructors for the classes `Person`, `Student` and `Staff`. Each class has the responsibility of initializing its own data members, so the constructors have to assign default values to the members. In the main method, after creating the instances of `Student` and `Staff`, display the information of the objects to confirm that the constructors assigned the expected values. Then ask the user for new values and display the objects again. The suggested default values for the data members are:

`name` = not assigned yet
`SSN` = 000-00-0000
`GPA` = -9
`major` = not defined yet
`salary` = -999

3. *Final* classes and methods

We saw in a previous chapter that we can declare variables that are *final*, i.e., they are constants and they cannot be modified in the program. In the same way, we can declare *final* methods and *final* classes.

A final method is a method that cannot be overridden in a subclass. The static and the private methods are implicitly *final*. The structure for declaring a final method is:

```
<access> final <returns> <name> (<parameters>) {
    ...
}
```

Modify the code you wrote for the last exercise and override the method `setName` of the class `Person` in the class `Student`, writing a method in `Student` called `setName` that assigns to `name` the string "JOE". Compile and run the program. Then declare the method `setName` of the class `Person` as a final method. Try to compile it again. You should get an error.

Besides final variables and final methods, you can declare final classes. These classes cannot be extended by other classes, which means that they cannot be the superclass of any other class. All the methods of a final class are implicitly final. The format of a final class is:

```
<access> final class <name> {
    ...
}
```

Declare the class Person as a final class and try to compile the program. It shouldn't work.

4. *Abstract* classes and methods

An abstract class in the context of inheritance can be seen as the opposite of a final class. The usual purpose of an abstract class is to be extended by another class, i.e., to be a generic superclass. An abstract class cannot be instantiated, so you cannot declare an object whose type is an abstract class. Its format follows:

```
<access> abstract class <name> {
    ...
}
```

An abstract class can have both abstract and non-abstract methods. An abstract method is a method without implementation, and whose purpose is to be the interface for the subclasses that must implement it. Any abstract method must belong to an abstract class. The structure of an abstract method is:

```
<access> abstract <returns> <name> (<parameters>);
```

Any non-abstract class that is derived from an abstract class has to implement the abstract methods of the superclass; otherwise you would get an error.

The methods in the abstract class that are concrete (non-abstract), and that have been defined, represent an uniform behavior for the subclasses that might be overridden if desired.

Modify the code of the exercise 2 and declare the class `ECE_Person` as an abstract class, given that every person in the ECE Department has to be classified as a student or as part of the staff. Declare, too, an abstract method in `Person` called `displayInfo`. This method was already implemented in the subclasses, so it should work. If you delete the method `displayInfo` in any of the subclasses you will get an error.

5. Interfaces

In contrast to C++, Java does not support multiple inheritance. But it supports interfaces that can be used as an alternative way of implementing multiple inheritance.
A Java interface is a data type similar to a class, which contains a set of method declarations without definition. It can contain declaration of constants, too. It is similar to the abstract classes that we just described. An abstract class that only has abstract methods and constants can be converted to an interface. The structure of an interface declaration is:

```
<access> interface <name> extends <list_interfaces> {
    ...
}
```

The words **extends** ***<list_interfaces>*** are optional, and they indicates that an interface can extend other interfaces, inheriting the definitions of the abstract methods and constants. The list of interfaces is a list of names separated by commas.

An interface can be implemented by any class, which must then define all the methods of the interface. At the same time, a class can implement more than one interface. The format of a class that implements interfaces is:

```
<access> class <name> implements <list_interfaces> {
    ...
}
```

An example of an interface and a class that implements it is presented here:

```
public interface Figure {
    public static final int THREE_SIDES = 3;
    public static final int FOUR_SIDES = 4;

    double getArea();
    double getVolume();
}

public class Square implements Figure {
    double getArea() {
         ...
    }
    double getVolume() {
         ...
    }
    ...
}
```

An advantage of interfaces over abstract classes is that a class can implement more than one interface in addition to extending a class by inheritance. This is the way in which multiple inheritance can be replaced in Java.

Define an interface called `Employee` that includes constants for the base salary ($10 per hour), for a bonus ($2 per hour) and for extra hours ($15 per hour). Also, include in the interface the definition of a method for getting the information of the employee and other one for calculating the monthly salary. Define two classes: `Secretary` and `Manager`, which implement the interface `Employee`. Since all employees must receive a salary, the interface Employee has a definition for a method that calculates it. But because the salaries of different kinds of employees are obtained in different ways, each class has its own implementation.

The class `Secretary` should have data members for: number of hours per month and number of years of work. The secretaries cannot work extra hours, but if they have worked more then 5 years, they get an extra 10% of the monthly salary, and if they work more than 150 hours per month, they get a bonus for each hour after 150.

The class `Manager` should have the same data members (number of hours per month and number of years of work) and a data member for extra hours. The monthly salary of a manager is the base salary multiplied by 2 times the number of hours per month, plus the extra hours times the extra hour rate. For each year of work they get a 2% of the salary.

In your main method declare an instance for `Secretary` and another one for `Manager`. Ask the user for the values of each object and calculate the monthly salary of each one.

Chapter 5

Exception handling in Java

Exceptions to what is expected happen in all aspects of life. Since computers attempt to solve real life problems, programmers have always had to deal with exceptions to the norm. In previous times, handling exceptions has been done with conditional structures that test for the abnormal condition, and direct the control to other instructions that take the appropriate alternate action. There are advantages, however, to providing formal means of handling exceptions. This makes for more clear, robust code. C++ is in the process of formally instituting an exception handling mechanism, proposed in 1990 by Koenig and Stroustrup, which interestingly enough, is very similar to (and precedes) that used in Java. This chapter discusses this formalism.

1. Exceptions in Computing

Say we want to calculate the average cost of travel per day of one particular trip, let's say to Paris. A program would ask for the total amount of Francs spent on the trip, and the length of the trip in number of days from departure to arrival. Naturally, since the trip was taken, the number of days must not have been 0. However, input errors do take place, and they would represent an exception. Dividing any number by zero results in a run-time error, and a program crash in most cases. So, some lines of code should be placed prior to the division so as to "catch" this exception, and execute some alternative instructions, which would allow the program to recover from the error and not crash. Such code would include a statement that says: "This cannot be possible. Please re-enter the length of trip in days". This could be reflected in the following Java snippet:

```
System.out.println("How many Francs did you spend on your
trip?: ");
francs = MlmIn.readDouble();
System.out.println("What was the length of your trip (in
days)?: ");
days = MlmIn.readInt();

while (days < 1)   // "Catching" the exception
{
  System.out.println("This cannot be possible.");
  System.out.println("Please reenter the length of trip
in days: ");
  days = MlmIn.readInt();
}

System.out.print("The average cost of travel per day to
Paris is ");
System.out.println(francs/days + " Francs.");
```

Please write, compile and run the above code. You will be making changes to it very soon, and you need to ensure that it runs correctly before making changes.

2. Exception in Java – the `try` block and the `throw` statement

In order to formalize the handling of exceptions, Java provides a set of procedures that facilitate this process. It is called the *try, catch* and *throw* operators. They are basically block identifiers that tell the interpreter to execute the instructions therein only when an exception has been detected.

The `try`-block contains the lines of Java code that should be executed when the situation is as expected. The name *try* implies that you are not absolutely certain that you want to run this sequence of instructions, even though it represents the normal conditions. Its format is as follows:

```
try
{
     code statements to try
}
```

The `try`-block should also contain the statements that monitor the situation for the existence of an exception. It is typically a conditional structure that identifies the trigger situation and "throws" the control to another block if the exception has been detected.

The `throw` operator is used as a statement within the `try`-block to redirect the flow to the alternate set of lines of code. The `try`-block now looks more like:

```
try
{
     <code statements to try>
     if (exception is present)
       throw new Exception("Exception: No days");
     <add'l code statements to try if no exception>
}
```

`Exception` is a predefined class in Java. The `throw` statement, through the `new` operator, forms a new object of that class (`Exception`), and throws it to the exception code, which *catches* it. The code inside the `try`-block, but underneath the `throw` statement, is what would have been executed if all had been normal, but now cannot be in the face of the exception. The program continues to execute all statements below the `try`-block unless execution terminated by the exception code. The format for the `throw` statement is as follows:

```
throw new Exception("<exception name string>");
```

The `exception name string` is a somewhat deeper issue than just a label, but we leave that for later.

It should be noted that once an exception is thrown, the `try`-block terminates. There is no going back to it through a `return` statement. This means that any objects instantiated in this block no longer exist and cannot be referenced.

Without compiling the code snippet (yet!), modify it to include the `try`-block and the `throw` statement.

<u>3. The `catch` block</u>

When the control is "thrown" due to an exception, it must certainly be caught by something. As the name suggests, the `catch`-block does the catching. The execution of the `catch`-block is referred to as *catching the exception*.

The `catch`-block is placed under the `try`-block, and has the following format:

```
catch(Exception e)
{
 <alternate code to execute when exception present>
}
```

Although it sure looks like the definition of a method, with the parenthesis and argument, it is definitely not one. But it sure does act like one.

The identifier "e" is called the *`catch`-block parameter*. It serves as a label for the exception that is caught, so that the `catch`-block can contain code to remedy that exception. The inclusion of the class name (`Exception`) in front of it serves to specify the *type* of exception that the `catch`-clock is able to catch. Any legal identifier can be used in lieu of "e", but "e" is traditional. This identifier is tied in with the label used when the throw statement creates a new object of the Exception class. We did say that there was more to it than met the eye at the time.

Turns out, the labeling string in the throw statement is really an argument for the constructor of the object being created through the `new` operator. The newly-created object stores that string label within it so that it can be used by the `catch`-block. When an exception is thrown, the type of exception it is becomes important. If the wrong type of exception is thrown, it will not be caught. The class `Exception` defines the type of exception thrown. It turns out that, for now, there is only one type of exception, and that is represented by the class `Exception`. The object created in the throw statement replaces the "e" in the catch-block parameter, and the block is executed.

The reason it does this is to provide a quick and specialized print statement to the screen by simply allowing the newly-created object of the class `Exception` to print out the string placed as an argument for the constructor. Every instance object of the class `Exception` has a methods called `getMessage()`. This method captures the string label in the throw statement, and can be printed through any output method. Once the exception is caught, the `try`-block terminates in that no other code from it will be executed. The control will now go to the `catch`-block, and upon completion, it will continue below it.

A `catch`-block only executes when the nearest `try`-block above it with the appropriate exception class in the throw statement throws an exception. This means that the thrown exception will search bellow it for catch-blocks of the same exception class.

This completes the basics of exception handling in Java.

Complete the exercise by writing an appropriate `catch`-block for the zero travel days exception. Compile the problem and execute it.

Please discuss in a paragraph or two what you think the purpose of the exception class is.

4. Predefined Exception classes

Up to this point, we have only discussed one exception class, the `Exception` class. However, that is not the only one that can exist. Some predefined classes in Java have their own predefined exception class for their methods. Some of these predefined calls methods throw exceptions, and these are predefined exception classes.

If you use any of these methods with the exceptions, you should place the method invocation within a `try`-block. Of course, a `catch`-block should follow it in order to catch the exception. The `catch`-block should contain the predefined exception class and should immediately follow the `try`-block. Additional code can be added to the `catch`-block if deemed necessary by the programmer.

For example, let's say that we define a method that reads a single character and returns it. This method makes a call to a method of the system that reads from the standard input device (keyboard), and that might throw an exception of a predefined exception class called `IOException`. The following snippet of code will give you an idea of how this is done.

```
public static char readChar()
{
        int current_char=-999;

        try
        {
           current_char = System.in.read();
        }
        catch(IOException e)
        {
           System.out.println(e.getMessage());
           System.out.println("Error reading character.");
           System.exit(0);
        }

        return (char)current_char;
  }
```

The predefined exception class has the information needed to allow the programmer to know what to do.

5. Defining and using your own exception class

A programmer/user can define his/her own exception class. All user-defined exception classes must be derived from a predefined exception class, most likely, but not necessarily the class `Exception`. When defining an exception class, the constructors are typically the only methods it will contain, other that whatever may be inherited from the parent class. So, most of the time, only the constructor is to be defined in the class.

So, why then, would you want to define your own exception class? Well, actually, it is only done to predefine the string that will be displayed in the output stream (typically, the screen). This specializes the message and tells the user of this class what and why this exception exists. It also provides a "label" for situations where there may be more than one exception thrown by a method. Each exception will correspond to a user-defined class. The Java exception handler will ensure that a thrown exception will be caught by the appropriate exception class (`catch`-block).

If there are more than one `catch`-block belonging to the same exception class below the `try`-block, then it is imperative on the user/programmer to determine the order so that the most appropriate one appears first. This resembles a `switch` block in C/C++ and Java, except there is no `break` statement. If no `catch`-block exists to catch an exception thrown, the application terminates (if not a GUI).

The user-defined exception class is defined as follows:

```
public class UserDefinedExceptionClass extends Exception
{
     public UserDefinedExceptionClass()   //constructor
     {
          super("Error: Dividing by zero days");
     }

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

Note that this represents an overloaded constructor, one that uses a built-in message for the exception class, and a second one that allows the user/programmer to define his/her own message. Both use the base class constructor, in the case, that of the `Exception` class.

Define your own exception class for the trip daily average program. Call it `DenominatorExceptionClass`, and have it send the following predefined message in the default constructor:

```
"Error: denominator cannot be zero."
```

But this is a dry message. We want to personalize it a bit for our purposes. Therefore, define another constructor that receives a message, and uses it to send to it from the main program the message:

"You have not entered any days. Surely you did take the trip, right?"

This overloads the constructor. Make the class a child of `Exception`.

6. Multiple `throws and catches`

A `try`-block is not limited to throwing only one possible exception. There may be several `throw` statements within its invoked methods. Some of these exceptions may be predefined, or user-defined. But while each `catch`-block can only catch one type of exception determined by the exception class, there may be more than one `catch`-block following the `try`-block. When an exception is thrown from the try-block, each of the catch-blocks is tried in order to see which is the first one that matches in type of exception (the exception class). The exception will be caught by the first catch-block that satisfies the type of exception thrown. Therefore, if an exception can be caught by more than one catch-block, the order in which the catch-blocks are listed is of importance. The key is that exceptions of a derived class will also be caught by catch-blocks belonging to its superclass. Thus, if an exception derived from Exception is placed after Exception, it will never be used. The trick is to put the higher-level classes lower in the sequence to catch all exceptions not caught otherwise.

Develop a second user-defined exception class and with its accompanying throw statement to throw an exception whenever the trip is less than 3 days in length. The message should say:

```
"Why bother to go to Paris for only 2 days?"
```

7. The `finally` block

After the `catch`-blocks have been placed in the code, a `finally`-block can be put to include code that should be executed regardless of whether an exception has been thrown. It is sort of the else statement. There may not be a need for one such block, so it is optional, but it does exist as a way of gracefully completing the process. Its format is:

```
finally
{
  ...
}
```

Put a set of code to print out the solution of the average cost problem. This code will be run regardless of whether an exception is thrown or not.

8. Passing the buck

If a call to the method A might throw an exception (or you throw it, after checking its returned value), the method B that is making the call should catch it. Nevertheless, if for any reason you don't want to catch the exception within method B, method B can pass this responsibility to whoever uses method B. This is done through the clause in method B's heading, which follows:

```
<access> <returns> methodB (<parameters>) throws
<exception_name>
{
      ...
      try {
         methodA; // methodA might throw an exception
or you
                // throw it with "throw new
               exception_name()"
      }
      ...         // NO catch-block for exception_name
}
```

Doing this, method B does not need to include any catch block for the exception of type `exception_name`. In summary, if a method makes use of other method(s) that might cause exceptions, the method has to catch them or has to pass them through the clause *throws* in its heading.

For example, if method C calls method B, and in method B, method A throws an exception that method B doesn't catch, method B will be finished as soon as the exception occurs and the exception is passed to method A that should catch it or pass it to whoever uses method A.

You are going to redesign the program that we have been working on. To read a denominator is a common task that is performed in many applications. So it's more useful to define a method that reads a number and throws an exception if it is zero. Write a public class called `Denominator`, with a static method called `readDenominator`. This method has to read a number, and if it is zero the method has to throw an exception. Note that it doesn't have to catch it, only throw it and pass it. Use the exception class `DenominatorExceptionClass` that you defined previously. Then use the method `readDenominator` for reading the number of days of the trip in the main program. Notice that now you don't need an if statement in the main program to check if the number of days is zero and you don't need either to throw a new exception because that is done in the method `readDenominator`. But you still need the catch-block to reenter the number of days.

Chapter 6

Streams and File I/O in Java

Input/output (I/O) can be performed on the keyboard/screen or on a file. This chapter deals with file I/O. I/O in Java is handled by streams. A stream is nothing more than an object that delivers data to a destination (output stream) or that takes data from a source (input stream). For example, `System.out` is an output stream.

There are two basic types of files: binary and text files. Binary files are processed as a sequence of binary digits. Text files on the other hand are handled as a sequence of characters and that can be read using a text editor. The classes for processing files in Java are numerous. We present here the most common ones.

1. Writing to binary files

Before describing the output to files, it's necessary to indicate that I/O operations might throw exceptions of type `IOException`. For now, we will assume that everything is going to be OK and we won't catch these exceptions. We will cover this topic later. But for now you need to include in the methods definition heading the clause `throws IOException`. For example, if the method A does I/O operations, its heading would be:

```
public void methodA throws IOException
{
      ...
}
```

The most common class used for writing to a binary file is `DataOutputStream`. This class is defined in the library `java.io`, so you have to import this library before doing any file I/O operation.

The first step for writing to a file is opening it. In order to open a binary file you have to declare a stream object of type `DataOutpuStream` that will handle the file. In the same declaration of the stream object you have to connect it to the actual file name. However, the constructor of `DataOutputStream` does not accept a file name as an argument, it only accepts another output stream.

Fortunately for us, there is a class called `FileOutputStream` that we can use to return an output stream. So when you create an object of type `DataOutputStream`, you have to send to its constructor a new object of type `FileOutputStream`. The constructor of this last class does accept a file name as an argument.

So, why do we not just use File`OutputStream` objects? It is because streams of this type don't provide output methods as nice as those provided by `DataOutputStream` objects.

The structure for creating an output stream is:

```
DataOutputStream <outputStreamName> =
  new DataOutputStream (new FileOutputStream(<fileName>));
```

This statement creates a stream called `outputStreamName`, and a new empty file called `fileName`. If the file already exists, it will be overwritten. Now you can use the just created output stream to write values of primitive types to the file. Some methods of the class DataOutputStream are:

```
public final void writeInt(int n) throws IOException
public final void writeLong(long n) throws IOException
public final void writeDouble(double n) throws IOException
public final void writeFloat(float n) throws IOException
public final void writeChar(int n) throws IOException
public final void writeBoolean(boolean x) throws IOException
public final void writeUTF(String s) throws IOException
public void close()throws IOException
```

The format for calling one of these methods is:

```
outputStreamName.methodName(<parameter>);
```

Remember that you should close the file as soon as you finish writing to it. This may avoid possible problems with the file if, for example, the program ends abnormally.
Write a program that asks the user the temperature of each day of a week and save them in a binary file. Remember that you cannot read the file with a text editor. In the next section we will see how to read binary files.

2. Reading from binary files

Now you need to read what you saved in a binary file. Reading from a binary file is very similar to writing to it. You have to declare an input stream whose class is DataInputStream, and you have to connect this stream to the name of the file you want to read from. The structure to open a binary file is:

```
DataInputStream <inputStreamName> =
  new DataInputStream (new FileInputStream(<fileName>));
```

The file you are trying to read from should exist, otherwise the last statement shown will throw a `FileNotFoundException`. Once you declared the input stream, you can use analogous methods to those for writing, but now for reading. You should know the format of the file's content in order to use the appropriate methods for reading. For example, if you wrote to a binary file a float number and next an integer, you should read them in the same order, first a float number and then an integer.
Some methods in the class `DataInputStream` are:

```
public final int readInt() throws IOException
public final long readLong() throws IOException
public final double readDouble() throws IOException
public final float readFloat() throws IOException
public final char readChar() throws IOException
public final boolean readBoolean() throws IOException
public final String readUTF() throws IOException
public void close() throws IOException
```

If any of these methods attempts to read a value from the file and the pointer of the file is already at the end of it, then an `EOFException` is thrown.

After reading the file you have to close it as you do it when you are writing to it.

Write another program that reads from the file the temperatures and calculates the average temperature of the week. At the beginning, the program has to ask the name of the file to the user. Don't forget to close the file.

3. Exception Handling with File I/O

Specific exceptions can be thrown when you are doing file I/O. It's a good practice to catch them in order to display an informative message, and to end the program or to correct the error, for example, asking the user for a new input. Most of the file I/O methods may throw an `IOException`. Some of the reasons for an `IOException` are to attempt to open a file for reading or writing without permission and opening a file for writing when no disk space is available.

All the methods shown here may throw an exception of this type. In addition, the methods that try to write data to a file may throw an `EOFException` if they reach the end of the file and don't success fully finish the writing process. Also, if you try to open a file for reading that does not exist, the method will throw a `FileNotFoundException`.

The classes `EOFException` and `FileNotFoundException` are subclasses of the class `IOException`. That means that if you only include in your program a catch-block for `IOExceptions`, the `EOFExceptions` and `FileNotFoundExceptions` will be caught by that block. However, it's better if the program has specific catch-blocks for those types of exceptions. The order in which you list the catch-blocks is important, because if the first catch-block is for an `IOExceptions`, this block will catch even the `EOFExceptions` and `FileNotFoundExceptions` before the other blocks attempt to catch them. So the best is to list the catch-blocks from more specific to more general.

The `EOFException` is useful for checking the end of the file when reading from it and you don't know how many times you have to read. The code snippet that follows shows how to do it:

```
try
{
  while (true)
  {
    n = inputStream.readInt( );
    System.out.println(n);
  }
}
```

```
catch (EOFException e)
{
   System.out.println("End of reading from file.");
}
```

Modify the programs written before and catch the three types of exceptions. For `IOExceptions` display the messages: "Problem reading from the file" and "Problem writing to the file" respectively in both programs. Also, for the same exception display the default message of the exception using `getMessage()`. For `EOFException` display the message "Error: end of file reached", and for `FileNotFoundException` display the message "Error: File not found. Are you sure you have that file?". Run the program several times and enter errors to check that the catch-blocks are working. For example, modify your code and attempt to read more data than the file has. See what happens.
Delete the *throws* statement in the heading of the class' method. You don't need it now.

4. Checking properties of the files – The `File` class

There is a useful class in Java called `File` that we can use for checking some properties of the files. What you usually have to do is to declare an object of type `File` whose constructor accepts a file name, and apply to it some of the methods defined for that class. Once you know the properties of the file (it exists and it can be read, for example), you declare a stream of one of the types explained in the previous sections to read from and to write to.

Some methods that you can apply to a `File` object are:

- `public boolean exists()`

Checks whether a file exists with the name associated to the object when it was created.
- `public boolean canRead()`

Checks whether the program can read from the file.
- `public boolean canWrite()`

Checks whether the program can write to the file.
- `public boolean delete()`

Tries to delete the file and returns the result of the operation (success or fail).
- `public long length()`

Returns the length in bytes of the file.
- `public String getName()`

Returns the name of the file.
- `public String getPath()`

Returns the path where the file is located.

Modify the program that asks the user the temperatures of the week and that saves them in a file. Check if the file that the user indicates already exists. If so, ask the user if she/he wants to overwrite it. If her/his answer is negative, then the program ends.

5. Writing to text files

Text files can be written and read using any text editor. To operate with them in a Java program you need special classes and methods. One of the most common classes used for writing to a text file is `PrintWriter`. The structure for declaring and opening a text file for output is as follows:

```
PrintWriter <outputStreamName> =
  new PrintWriter (new FileOutputStream(<fileName>));
```

This statement may throw an `IOException`. The way of writing to a file of this type is the same than displaying information to the screen. In fact, the methods are the same, but they send the data to the files. Some methods of the class `PrintWriter` are:

```
public final void println(Object outputData)
public final void print(Object outputData)
public void close()
```

Remember that the class `Object` is the root of the whole hierarchy of classes in Java. This means that at the end of the chain all the objects in Java are of type `Object`. For example, assuming that *num* is an integer variable, a valid output to a file handled by the stream *myfile* is:

```
myfile.println("The number is " + num);
```

Now let's write another version of the temperatures program. Modify the program that writes to a binary file and make it works with text files. The resulting file should have sentences like:

```
Temperature of day 1:
80
Temperature of day 2:
77
...
```

Note that the number should be in a separate line. Run it and verify with a text editor that the information is correct.

6. Reading from text files

The classes and methods used to write to and those to read from binary files are very similar to each other. However, such is not the case for the methods and classes for reading from and those for writing to text files.

We will use the class `BufferedReader` to declare an input stream for text files. The constructor of this class accepts as argument an object of type `Reader`. So we will make use of the class `FileReader` that takes as argument the name of a file and returns a `Reader` object. In summary, the structure for declaring an input stream for text files follows:

```
BufferedReader <inputStreamName> =
  new BufferedReader (new FileReader(<fileName>));
```

This statement may throw a `FileNotFoundException` if the file that is trying to open doesn't exist. Some methods of the class `BufferedReader` are:

```
public String readLine() throws IOException
public int read() throws IOException
public void close() throws IOException
```

The method `readLine()` reads a line of text from the file and returns it. If the method goes beyond the end of the file, it returns *null*, and does not throw an `EOFException`. The second method `read()` reads a single character from the file and returns it as an *int* value. Again, if the method goes beyond the end of the file, it returns the value –1 and does not throw an `EOFException`.

Note that if you want to read a character from a file and save it in a *char* variable, let's say in the variable *symbol*, you will have to type cast it, doing something like:

```
char symbol = (char)(inputStreamName.read());
```

Also, as you can see, there are no methods for reading numbers or other types. So if you want to read a number from a text file you have to read it as a *String* or as a *char* in the form of an *int*, and then you have to convert it. The way of converting a *String* to an *int* requires some explanation. There is a class called `Integer`, which has a static method called `valueOf` that takes a *String* and converts it to an object of type `Integer`. Since we want a value *int* and not a value `Integer`, we have to do one more conversion. We take the *String* returned by `valueOf` and we apply to it the method `intValue`, which returns an *int* value. For example, if we want to save in the *int* variable "n" a number read from a file, we would write:

```
int n =
Integer.valueOf(inputStreamName.readLine()).intValue();
```

Similarly, we do the same for the types *double*, *Long* and *Float*, replacing the class `Integer` with `Double`, `Long` or `Float`, and changing the method `intValue` to `doubleValue`, `longValue` or `floatValue`.

Let's modify the program that reads from a file the temperatures of a week and that calculates the average. Make it read from a text file. Remember to close the file when you are done with reading from it.

Chapter 7

Non-Primitive Data Types in Java

Frequently, in real applications we need to manipulate large amounts of data, and in many cases, the data is of the same type. Thus, it becomes indispensable to have a structured way to keep this information. This chapter presents some non-primitive data types that are able to hold groups of values: arrays and vectors.

1. Arrays

An array is a variable that holds the address of a collection of values of the same type (base type). The base type can be a primitive type in Java or any class type. There are three basic ways for declaring an array:

```
base_type[] array_name = new base_type[array_size];
    or
base_type[] array_name;
array_name = new base_type[array_size];
    or
base_type array_name[] = new base_type[array_size];
```

You remember that in C++ you indicate the size of the array in the same statement where you declare the array. For example:

```
double myarray[10];
```

You cannot do this in Java. First, you declare the array without any size, and then you assign memory to it using the method *new*. This is, in fact, a dynamically allocated array.

As in C++, the indexes of arrays start with 0. Thus, `myarray[0]` is referencing the first element of the array. The last element of the array is the value that corresponds to the length of the array minus 1. The length of the array can be obtained using the instance variable `length` that is part of array objects, and which holds the size of the array. The instance variable `length` is used as any other instance variable of an object, as shown in the following example:

```
size = myarray.length;
```

You, as the programmer, should not try to access elements with indexes that are out of bounds. If you try to access an element with an index equal to or beyond the number returned by `length`, you will get the exception `ArrayIndexOutOfBoundsException`. If this occurs, this is an error of design or implementation and not an error entered by the user. Thus, the program shouldn't catch it, because the program should be free of this type of exceptions.

Once the array has been declared, it has to be initialized assigning values to each element of the array. There are different ways for doing this. The typical way is to assign value by value to each element of the array after declaring and creating the array variable. For example:

```
int[] myarray = new int[3];
myarray[0] = 8;
myarray[1] = 24;
myarray[2] = 5;
```

A more concise way for doing the same is to initialize the array using curly brackets at the same time that it is declared. The next example shows this idea and it is equivalent to the previous example:

```
int[] myarray = {8, 24, 5};
```

Notice that with this last structure, the array is created automatically to hold three elements. It is not necessary to use `new` or to indicate explicitly the length of the array. A third way for initializing an array can be used when the initial values are all the same (for example, zero), or they can be obtained through the same calculations (for example, the value of element `i` of the array is the value of element `i-1` plus 2), or they are entered by the user. If this is the case, usually it is possible to use a `for` loop for the initialization.

Create a program that reads a word entered by the user, and that determines whether it is a palindrome or not. A palindrome is a string that reads the same forward and backwards, such as `radar`. You have to read in character by character and save each one in an array of characters.

Suggestions:

- Note that when reading a character from the keyboard, the user types the character followed by the ENTER key, that in reality is the newline character. The `char` variable holds only one character, in this case, the first one that is entered by the user. The newline character is kept and caught by the next read operation. Nevertheless, you are not interested in the new line character, so you can throw it away. This can be done including just after the read operation the following command:

  ```
  System.in.skip(2);
  ```

  This command skips the two bytes corresponding to the new line character. You need to import the library java.io.* and you might need to ignore the `IOException`s that that method might throw.
- Ask the user the length of the word.
- Use two arrays: one holds the word forward and the other backwards.

Test the program with palindromes such as radar, rotor, rotator, kayak, level, racecar and madam. Can you find more?

2. Multidimensional Arrays

So far, we have discussed arrays of one dimension (linear arrays). But we might want to define an array with more than one dimension. The most common one is two-dimensional array, usually called *matrixes*. We discuss here two-dimensional arrays, which can be extended easily to n-dimensional arrays.
A two-dimensional array is created and initialized very much like a one-dimensional array. You can declare and create an array using any of the following structures:

```
base_type[][] array_name =
        new base_type[num_rows][num_cols];
    or
base_type[][] array_name;
array_name = new base_type[num_rows][num_cols];
    or
base_type array_name[][] =
        new base_type[[num_rows][num_cols];
```

For initializing a two-dimensional array you can use any of the three basic ways explained for one-dimensional array. One of these is to use two nested `for` loops, for example when the user has to input the values. The second one is to assign value by value to each element of the array, for example:

```
int[][] table = new int[2][3];
table[0][0] = 6;
table[0][1] = 1;
table[0][2] = 5;
table[1][0] = 4;
table[1][1] = 3;
table[1][2] = 8;
```

The last way for initializing a two-dimensional array is to assign the values in the same declaration of the array using curly brackets. For example:

```
int[][] table = {{6, 1, 5}, {4, 3, 8}};
```

This statement will create an array called `table` with two rows and three columns, which is equivalent to the previous example. We will use the convention of calling `row` to the first dimension or index, and `column` to the second dimension or index.

Notice that an array of two dimensions can be seen as an array whose base type is another array. Following with the example of the variable `table`, we can say that `table` is a linear array whose base type is another array of integers. This means that each element of the array `table` is an array of integers. Thus, what do you think the following statement will print?:

```
System.out.println(table.length);
```

It will print the number of rows (first dimension) of `table`, i.e., 2. If we want to print the number of columns (second dimension), we have to make reference to any of the elements of `table`, which is another array, so we will get the second dimension, i.e., 3. We do it as follows:

```
System.out.println(table[0].length);
```

We can use any other valid index of table instead of 0. We just need to get the length of any of the elements of `table`.
Create a program that reads the grades of a specific number of students and calculates the final grade. Each student has a midterm test, a final test and a project. All the grades are given in the scale from 0 to 100. The evaluation is as follows:

Midterm test = 35%
Final test = 40%
Project = 25%

Keep these grades and the final grade of all the students in a two-dimensional array. The user has to input all the grades of the students 1 through *n* (the user specifies at the beginning of the program how many students there are). Define a method called `getFinal` that calculates the final grade of one student. This method will be called once per each student. The method `getFinal` receives as argument a one-dimensional array, which contains the data of one student. Remember that a two-dimensional array is a one-dimensional array whose base type is another array. For example, if `students` is a two-dimensional array, the reference to `students[0]` is a one-dimensional array. The method returns the final grade. Your program must print all the final grades of the students after reading the grades of all the students.

3. Ragged Arrays

Continuing with the idea of that a two-dimensional array is an array of arrays, we can create the array in several steps. First, we declare and create the linear array:

```
int[][] rag_array;
rag_array = new int[3][];
```

To this point, we have created a linear array with three rows, where each row is an array of integers that have not yet been created. We can create these arrays separately. This allows us to define different array's sizes for each row. For example:

```
rag_array[0] = new int[2];
rag_array[1] = new int[10];
rag_array[1] = new int[5];
```

These arrays, where different rows can have different number of columns, are called `ragged arrays`.

Using as base the program that you created for getting final grades, define a new program that reads the final grades of the courses taken by some students and that calculates the GPAs. Since different students may have taken different number of courses, you should use a two-dimensional ragged array to keep the information. The program can ask the user how many students are going to be processed, and for each student how many courses he/she has taken. After asking the information of all the students, the program has to display the list of GPAs.

4. Vectors

A limitation of arrays is that once the size has been assigned to them in the program, we cannot make them bigger or smaller. The class `Vector` overcomes this limitation. An object of type `Vector` is a kind of array that can change its size while the program is running. Nevertheless, a vector has other restrictions. Vectors are less efficient than arrays, and the elements in the vector must be objects, i.e., they cannot be of primitive types.

Fortunately, Java provides wrapper classes, which are classes equivalent to the primitive types. The wrapper classes for the primitive types `int`, `long`, `double`, `float`, `char` and `boolean` are `Integer`, `Long`, `Double`, `Float`, `Character` and `Boolean`, respectively.

These wrapper classes have a method to convert the object of a wrapper class to its respective primitive type. For example, if *num* is an object of type `Integer`, and *n* is an `int`, we can use the method `intValue` of the class `Integer` to convert the value of *num* to a value of type `int`. This looks like:

```
int n = num.intValue();
```

There are similar methods for the other wrapper classes (`longValue`, `charValue`, etc.).

Coming back to vectors, the definition of the class `Vector` is in the package `java.util`, so you need to import this package (`import java.util.*`).
This class has three constructors, which are explained with examples. The statement

```
Vector v = new Vector();
```

is using the default constructor, which creates a vector called *v*, with an initial capacity for 10 elements. We can define the initial capacity of the vector specifying it in the constructor:

```
Vector v = new Vector(30);
```

This creates a vector *v* that can hold 30 elements. In either of these two cases, if we exceed the capacity of the vector adding elements, the capacity of the vector will be increased automatically to its double. There is a third constructor, with which we can specify both the capacity and the increment in elements that we want to be added to the vector when its capacity is exceeded. This is shown in the following example:

```
Vector v = new Vector(30, 10);
```

The next question is: what kind of elements can be held in a vector? The base type of the elements of a vector is the class `Object`. Remember that this class is the root of all the classes in Java. This means that any instance of any class is of type `Object`. Thus, a vector can hold all types of objects.
When we want to add an element (object) at a specific location of the vector, we use the method `setElementAt`. The structure is shown:

```
    vector_name.setElementAt(object, index);
```

This is illustrated with the following example:

```
    v.setElementAt("Jennifer", 0);
```

As with traditional arrays, the first position of a vector is 0.
The method `elementAt` returns the object saved at a determined position. Its format is:

```
    vector_name.elementAt(index);
```

Since the base type of the elements of a vector is the class `Object`, the elements returned by `elementAt` are of that type, even if we saved previously objects of a particular class. Thus, we might want to type cast the value returned by the method, as shown in this example:

```
    String name = (String)v.elementAt(0);
```

The `Vector` class has some methods that could be useful for you. We show here some of them:

public final void addElement(Object newElement)
    Adds the new element to the end of the vector.

`public final void insertElementAt(Object newElement, int index) throws ArrayIndexOutOfBoundsException`

    Inserts the new element at the specified position. It doesn't replace the last value at that position, it shifts the elements.

`public final void removeElementAt(int index) throws ArrayIndexOutOfBoundsException`

    Removes the element at the specified position and shifts the elements with greater index.

`public final boolean removeElement(Object element)`

    Removes the first occurrence of the element and shifts the elements with greater index.

`public final void removeAllElements()`

    Removes all elements in the vector.

`public final boolean isEmpty()`

Returns *true* if the vector is empty; otherwise, it returns *false*.

`public final int size()`

Returns the number of elements in the vector.

`public final int capacity()`

Returns the current capacity of the vector.

`public final void trimToSize()`

Trims the capacity of the vector to its current size.

`public Object clone()`

Returns a copy of the vector.

Modify the program of palindromes, and instead of using an array to save the letters of the word, use a vector. Create the vector with an initial capacity for 5 elements, since many palindromes have 5 letters. Also, instead of asking the user for the length of the word, the user has to type a dot to indicate the end of the word. Don't save the dot in the vector. Your program has to read the letters of the word, determine if it is a palindrome, display the word backwards and indicate whether it is a palindrome or not. Also, the program must ask the user if she/he wants to enter another word (when doing this, you can delete all the elements of the vector to start with another word).

Chapter 8

Windows Programming in Java

This chapter introduces the topic of windows programming. It is only an introduction to Graphical User Interfaces and presents the basics for defining simple visual interfaces.

1. Graphical User Interfaces (GUI) in Java

It's very important to design a user-friendly interface that allows the user to interact easily with a program. The most popular type of user interface is the graphical user interface (GUI). This kind of interface contains graphical components or GUI components, which are visual objects that the user interacts with.

Some of the classes that define GUI components in Java are `Window` (area that contains GUI components), `Label` (for displaying read-only text), `Button` (for being clicked and triggering an action), `Checkbox` (to select and unselect an option), and `TextField` (for displaying and entering text). There are others such as `Choice`, `List` and `Panel` that won't be covered in this chapter.

In order to clarify the understanding of these classes, it's important to note that there are two basic types of GUI components: those that are just components, and those that are containers, besides being components too. A container object can hold several components. For example, `Window`, `Frame`, `Panel` and `Applet` are classes that derive from the class `Container`, and that can include components such as `Button`s, `Checkbox`es, `Label`s and `TextField`s, which derive from the class `Component`. The reason why this is explained is that these classes inherit some common methods. For example, all the classes that derive from `Component` have the methods `paint` and `repaint`, and all the classes that derive from Container have the method `add`. These methods will be explained later.

The definition of the classes for GUI objects is in the package `java.awt` (Abstract Windowing Toolkit), so you have to import all the classes of this package (`java.awt.*`) in your program.

When we are designing GUIs, we usually base our design on one or more windows, which contain the graphical components of our interface. Thus, the first step is to define a class that implements a container area (a window) where the GUI components will be displayed.

Java provides a predefined class called `Window`. Nevertheless, this is so simple a window that it is not very helpful. Java also provides the class `Frame`, which is derived from `Window`, but which implements more functionality.

When we define a new class derived from `Frame`, the AWT system calls automatically a method of the class `Frame` that paints the window. This method, called `paint`, receives an argument of type `Graphics`. This argument is the "paintable" area of the screen. Although the method `paint` is called automatically, you usually need to define it, for example, for displaying some text on the window. Every object of type `Graphics` has a method called `drawString` that writes text at the specified position. The structure of this method is:

```
    graphicsObj.drawString(text, xCoordinate, yCoordinate);
```

Thus, this method can be invoked within the method `paint`, whose format is:

```
    public void paint (Graphics objName)
    {
        ...
    }
```

Besides `paint` - a method that you don't call but you do define - the class `Frame` has other methods that you don't have to define but that you have to call explicitly. Some of these methods are:

`void setSize(int width, int height)`
Sets the size of the window, given in pixels.

`void setTitle(String title)`
Sets the title of the window.

`void setBackground(Color c)`
Sets the background color of the window.

`void setForeground(Color c)`
Sets the foreground color of the window. For this method and setBackground you can use as arguments the following values: Color.while, Color.gray, Color.black, Color.orange, Color.blue, Color.red, Color.pink, Color.yellow, Color.green, Color.magenta and Color.cyan.

`void setVisible(boolean b)`
Makes the window visible if the argument is true. Otherwise, makes it invisible.

Now you are ready to define your first window (but not to run it!). Define a class called `SimpleWindow` that derives from `Frame`. Your window has to display the message *"Welcome to my first window"*, so you have to define the method `paint` within the class `SimpleWindow`. Remember that you don't have to call this method, it will be called automatically. In the `main` method of your class, instanciate an object of type `SimpleWindow`, which will be in fact your window. Set the appropriate size of the window, the background and foreground color, and set the title with your name. For example, *"Maureen's window"*. The last statement of the `main` method has to be a call to the method `setVisible`. Compile the program but don't run it yet, because there is still something left for closing the window. We will see this in the next section.

2. Event-Driven GUIs in Java

When the user interacts with the GUI components, the components fire events. An event is an object that represents an action, such as pressing a key, clicking a button with the mouse, closing a window, or any other action performed by the user during the interaction with the GUI. This is why it's said that GUIs are event-driven.

Thus, it's necessary to write code for handling these events. The classes related to events are defined in the package `java.awt.event`, so you have to import all the classes of this package in your program.

To process an event requires two main steps:

1. To register an *event listener*: an event listener is an object that listens for specific types of events.
2. To implement an *event handler*: an event handler is a method that is called automatically in response to a specific event.

For each GUI component that might fire an event, we should add one or more event listeners to that component. For example, a button in a GUI can fire an event when it is clicked. So we need to add an event listener to that button. This event listener will receive the event when it occurs. Then, it is expected that the event listener has one or more event handlers. Thus depending on the type of event, the event listener will call automatically the appropriate event handler.

Each type of GUI component behaves differently, and for some of them there is a predefined listener class that implements the event listener and its event handlers. To this point, we have only worked with frames (or windows). There is a listener class for `Window` and `Frame` components called `WindowAdapter`. This class can be used to create a new event listener. The class `WindowAdapter` already has predefined methods (event handlers) that respond to different kind of events. These methods are `windowOpened`, `windowClosing`, `windowClosed`, `windowIconified`, `windowDeiconified`, `windowActivated` and `windowDeactivated`.

Usually we don't need to write the definition of these methods, they already implement the expected behavior, and, if we registered the event listener, they will be called automatically when the event occurs.

Nevertheless, sometimes we need to override a method. The most basic event that a window should handle is closing the window. The class `Frame` already includes the closing button in the component, but we need to write the definition of the handler for this event. Thus, the way for doing this is to define a new class that derives from `WindowAdapter` inheriting all the methods (handlers). In this new class we can redefine the method `windowClosing`, which must follow this format:

```
public void windowClosing(WindowEvent e)
{
      ...
}
```

This method (event handler) will be called automatically when the user presses the closing button of the window. As you can see, this method, as well the others of the class `WindowAdapter,` receives an argument of type `WindowEvent`, which has the information of the event. In the body of the method `windowClosing` you can include a statement to finish the program, such as:

```
System.exit(0);
```

Once that you have defined the new listener class, you just have to declare in your program a listener object of that class and add it to the window object. The `Window` and `Frame` classes have the method `addWindowListener` that registers an event listener associated to the window. For example, if you have an object ***w*** whose type is a derived class of `Frame`, and you defined a new class called `myWinListener` derived from `WindowAdapter`, in your program you will include the following lines:

```
myWinListener listener1 = myWinListener();
w.addWindowListener(listener1);
```

Now you can complete the previous exercise. Define a listener class called `WindowListener` that derives from `WindowAdapter`. In this class redefine the method `windowClosing` to finish the program when the user presses the closing button. Add to main method of the class `SimpleWindow` that you already defined the lines for registering the event listener. Then, run the program.

3. Adding components

Now we want to add some graphical components to our window. All the classes that derive from the class `Container`, such as `Frame`, have the method `add`. This method adds to the container object the specified component. For example, if `mywin` is an object whose type is any derived class from `Frame`, and we want to add a component to that window, we have to write:

```
mywin.add(componentObject);
```

The argument `componentObject` can be any object whose type is any class derived from `Component`. One of these classes is `Button`, which implements the push buttons of GUIs. When we create a new button we can specify the label that it will show. The general structure for creating an object of type `Button` and for adding it to the window is:

```
Button buttonName = new Button(label);
add(buttonName);
```

Note that the method `add` was called without being tied to any window object. This is done when the previous statements are written inside the constructor of the class derived from `Frame`.

However, before adding a button or any other component to the window, it's necessary to associate an event listener to the button. Remember that a frame fires events of type `WindowEvent`. And in the other hand, a button fires events of type `ActionEvent`. The event listeners that can handle this last type of event are of type `ActionListener`.

ActionListener is a predefined interface that Java provides. This interface has an event handler (a method) called actionPerformed, which receives an event of type ActionEvent as argument. This method is in charge of handling the event. Note that ActionListener is an interface and its methods are not implemented. Thus, your class, besides extending the class Frame, has to implement the interface ActionListener. Remember that a class that implements an interface must define the methods of the interface. Thus, your class has to define the method actionPerformed.

In summary, first you create the button. Then, you register or associate a listener to that button, and finally you add the button to your window. There is a method called addActionListener that associates a listener to the button. In the next lines you can see the three steps (suppose that the statements are written inside the constructor of the class):

```
Button buttonName = new Button(label);
buttonName.addActionListener(this);
add(buttonName);
```

The second line requires more explanation. We are adding to buttonName a listener using the method addActionListener. The confusing part is that we are registering as the listener the argument "this". Note that your class that is extending Frame, let's call it SimpleWindow, is implementing at the same time the interface ActionListener. So, in fact, your class SimpleWindow is a listener. When we write in the constructor of the class

```
buttonName.addActionListener(this);
```

"this" is making reference to the object instantiated of type SimpleWindow.
If we add more buttons to the window, we are using as the listeners of all the buttons the same object of type SimpleWindow. So it's necessary to find a way in which the event handler (actionPerformed) can know which button fired the event.

As we said, actionPerformed receives an argument, let's say "e", of type ActionEvent. This argument has the information of which component fired the event. Well, "e" can use the method getActionCommand to return the label or string associated to the button. This label will be used as the identification of the button. With this information we can have in the method actionPerformed a branching statement, such as an "if", and we can determine what to do depending on the button that produced the event.

For example, if our window has two buttons, one with the label "continue" and the other with the label "stop", the event handler will look like:

```
public void actionPerformed(ActionEvent e)
{
     if (e.getActionCommand().equals("continue"))
     {
          ...
```

```
        }
        else if (e.getActionCommand().equals("stop"))
        {
             ...
        }
        repaint();
    }
```

`repaint` is a method that you have to call if you change something in your window that needs to be redrawn. For example, if you change the color or the text displayed on the window, `repaint` will call automatically the method `paint` that you defined.
In summary, the format of your class (the parts concerning to adding a button) is:

```
    public class WindowClass
        extends Frame implements ActionListener
    {
      public WindowClass()
      {
        ...
        Button buttonName = new Button(label);
        buttonName.addActionListener(this);
        add(buttonName);
        ...
      }

      public void actionPerformed(ActionEvent e)
      {
        ...
      }

      public static void main(String[] args)
      {
        WindowClass mywin = new WindowClass();
        mywin.setVisible(true);
        ...
      }
      ...
    }
```

Define a new class called `ColorWindow`, which will contain three buttons. One has to say "`Exit`" and it will have the same function as the closing button of the window. So your window will have several ways to exit the program. A second button has to say the name of a color, for example "`Green`". When the user presses that button the background color of the window changes to that color, and a particular message associated to that color has to be displayed on the window, for example, "`The grass is green`". The third button has to say another color, for example "`Red`". When the user presses that button, the background has to change to that color and a new message has to be displayed related to the color, for example, "`My dog Dots is red`".

Your class should have a data member that holds the current text that is displayed on the window. In this way, you can use that data member in the method paint to display the message instead of the literal, and you can change the value of the data member in the `actionPerformed` method.

In the constructor of your class, include the following line. It will be explained in the next section:

```
setLayout(new FlowLayout());
```

4. Layout manager

As we saw, the method `add` simply throws objects into a window. We might wish to arrange those objects in a particular way. A *layout manager* is an object that positions components inside a container object. There are some predefined layout managers in Java that arrange the components into a window. Also, there is a method that sets a particular layout manager for our window, and which we can call from the constructor of our class. The format of this method is:

```
setLayout(new LayoutManagerClass());
```

This method can be called from the constructor of our class that implements the window. `LayoutManagerClass` can be anyone of the predefined classes that Java provides for layout managers. We will mention only three of them.

The most basic layout manager is the class `FlowLayout`. This manager arranges the objects on the window line by line. The manager starts positioning the objects in the first line, and when it is full, the manager continues with the following line, putting the objects in the order that the program adds them to the window. The only line that we need to add to the constructor is:

```
setLayout(new FlowLayout());
```

Another layout manager is BorderLayout, which places the objects according to five main areas: the North (top) border of the window, the South (bottom) border, the East (right) border, the West (left) border, and the center of the window. In this case, after setting this layout manager we have to specify which area we want to use for each component that we add to the window. For example,

```
setLayout(new BorderLayout());
Button mybutton1 = new Button(label1);
add(mybutton1, “North”);
Button mybutton2 = new Button(label2);
add(mybutton2, “East”);
Button mybutton3 = new Button(label3);
add(mybutton3, “Center”);
```

A third layout manager is the class `GridLayout`, which arranges the objects in rows and columns. This manager is similar to `FlowLayout`. The `GridLayout` divides the window into a grid with the specified number of rows and columns. Each piece or cell of the window has the same size. As the program adds components to the window, the `GridLayout` manager starts to place the objects in order from left to right, and from top to bottom. The components will have the same size as each cell of the grid. For example, if you have a grid of 2 rows and 3 columns, your window will have probably big buttons. But if you have a grid of 5 rows and 7 columns, the buttons will be smaller. As with the `FlowLayout` manager, you only have to set the manager (suggestion: in the constructor), and you have to specify the size of the grid, as shown in the next line:

```
    setLayout(new GridLayout(numRows, numColumns));
```

Modify the class `ColorWindow` that you defined in the last exercise and set different layout managers.

Chapter 9

Applets in Java

In Java we can create different types of programs. The most common Java programs are applications and applets. Up to this point we have created applications, which are standalone programs. Applets are similar to applications, but they don't run standalone. Instead, a Java applet is a program that adheres to a set of conventions that allows it to run within a Java-compatible browser (for example, Netscape or Internet Explorer). Applets are designed to run from a document on the Internet. This chapter describes how to create applets in Java.

1. A simple applet: the HelloWorld applet

This section describes the basics for writing an applet. Every applet must define a subclass of the `Applet` class. Applets inherit a great deal of functionality from the `Applet` class, ranging from communication with the browser to the ability to present a graphical user interface (GUI).

In order to use the `Applet` class, it is necessary to import the package `java.applet.*`. Besides this package, you will also need the AWT library, so you have to import `java.awt.*` and `java.awt.event.*`.

Every applet must implement at least one of the following methods: `init`, `start`, or `paint`. Unlike Java applications, applets do not need to implement a `main` method. The methods `init`, `start`, or `paint` are called automatically, so you don't need to invoke them. The `paint` method has the same functionality as we saw in the last chapter. It is typically used to display some text on the window through the method `drawString` that writes text at the specified position (see chapter 8 to review how to use it).

The `start` method is called every time the user of the browser returns to the HTML page that contains the applet.

The `init` method serves a similar purpose as the constructors, given that applets do not normally use constructors. In the `init` method you place all the initializing actions, such as setting color and adding GUI components to the applet. This method has no parameters.

Besides the `init`, `start`, and `paint` methods, applets can implement two more methods that the browser calls automatically when a major event occurs: `stop` and `destroy`. The `stop` method is called when the applet stops executing, for example when the user leaves the applet's page. The `destroy` method is called when the applet is removed from memory, for example when the user closes the browser.

Applets can implement any number of other methods, as well.

In contrast to graphical applications in Java, applets do not need the `setVisible` method, the `show` method, the `setTitle` nor the `setSize` method, because applets are embedded in HTML documents, which take care of these functions. Also, when the HTML document is closed, it will automatically close the applet, so the applet does not need listeners such as `WindowDestroyer` for this purpose.

Now you are ready to write the source code file for an applet that displays the string "Hello World". In order to do this, define a subclass deriving from `Applet` called `HelloWorldApplet`. This class should implement just one method, the `paint` method, where you will display the message. Compile the file as you normally do, but don't run the applet because you need to write the HTML file first, as it will be explained in the next section.

2. Creating an HTML file

As we already said before, applets are not standalone programs. They are meant to be run using a Java-compatible browser, such as Netscape or Internet Explorer. Thus, applets should be embedded in HTML (Hypertext Markup Language) pages, which can be read through a web browser.

An HTML document consists of commands that determine how the page will be seen on the browser. Most of the HTML commands are composed of two delimiters, one that marks the beginning of the text to which the command applies and another one that marks the end of the text. Each command's delimiter is written within the symbols "<" `and` ">". The format of a command is:

```
<command>
     Text to which the command applies
</command>
```

An HTML file is a text file that can be created using any text editor. Its name should end with `.html`. The beginning and the end of the entire document is enclosed in the pair `<HTML>` and `</HTML>`. Every HTML document has a head that contains information that is used by the browser. The delimiters for the head part are `<HEAD>` and `</HEAD>`. Within the head of the document we can specify the title of the page using the command `<TITLE>` and `</TITLE>`. Between these two tags we specify the title. Up to this point, the HTML file should look like:

```
<HTML>
<HEAD>
<TITLE>
Title of the page
</TITLE>
</HEAD>
</HTML>
```

An HTML file has another part that is the body of the document. This part is enclosed by the tags `<BODY>` and `</BODY>`. Everything that we write between these two tags will be display on the screen. For example:

```
<HTML>
<HEAD>
<TITLE>
Title of the page
</TITLE>
</HEAD>
<BODY>
The output of my applet is:
</BODY>
</HTML>
```

The next step is to include in the HTML file a command that displays and runs our applet. Using the tags `<APPLET>` and `</APPLET>`, you specify (at a minimum) the location of the `Applet` subclass and the dimensions of the applet's onscreen display area. When a Java-capable browser encounters an `<APPLET>` tag, it reserves onscreen space for the applet, loads the `Applet` subclass onto the browser, and creates an instance of the `Applet` subclass. The `<APPLET>` command, as some others, can take some arguments. Using the parameter `CODE`, the `<APPLET>` command will take the name of the file `.class` that contains the applet. To specify the size of the applets's onscreen display area we can use the parameters `WIDTH` and `HEIGHT`. In the following snippet of code you can see how to use this command:

```
<HTML>
<HEAD>
<TITLE>
Title of the page
</TITLE>
</HEAD>
<BODY>
Some text to be displayed
<APPLET   CODE=”HelloWorldApplet.class”   WIDTH=integer
HEIGHT=integer>
</APPLET>
</BODY>
</HTML>
```

Once that you have created an HTML file like that, you are ready to run your applet. In order to do this, open the just created HTML file using any browser such as Netscape or Internet Explorer, and that's it. JDK provides an applet viewing program that can be used instead of these browsers. Usually it is used only for testing purposes. Its name is the Applet Viewer. If you want to use it you have to type in the command line of the operating system:

```
appletviewer HTML_File.html
```

Complete the `HelloWorld` applet creating an HTML file, and run it using the Applet Viewer of JDK and then using a web browser.

3. An applet with GUI components

In this section you will write a more complex applet, which contains some GUI components, specifically a text field and a label. Thus, this exercise complements the chapter "Windows Programming in Java" because what is explained here applies also to applications.

The class that implements a label is `Label`. The format to declare a new label is as follows:

```
Label myLabel;
```

Probably, you will include this statement as part of your class, so *`myLabel`* will be a data member of that class. Then, you need to create the object *`myLabel`* specifying the text that it will display. This can be done in the `init` method, following the next format:

```
myLabel = new Label("text of the label");
```

Similarly, a text field object is declared as a data member of the class that you are defining using the predefined class `TextField`, as it is shown here:

```
TextField myField;
```

The creation of the object in the `init` method will follow the next format, indicating the length of the text field:

```
myField = new TextField(an_integer_number);
```

As with other GUI components, a text field object can fire an event. This event occurs when the user types something in the text field and presses the `Enter` key. As it was explained in chapter 8, we can catch events like those implementing the interface `ActionListener`. Therefore, the class that inherits from the class `Applet`, has to implement this interface and at the same time has to define the method (event handler) called `actionPerformed`.

You can see that this section is a review of the windows programming chapter and an introduction of the GUI components `Label` and `TextField`. Now you have to put together this knowledge into an applet.

You have to write an applet that uses a label and a text field to enable the user to interact with the program. This program inputs integers typed by the user at the keyboard, computes the sum of these values and displays the result. As the user types each integer and presses the Enter key, the integer is read into the program and added to the total.

In order to do this, create a new class called Addition. The data members of this class should be a label, a text field, an integer variable that will hold the value entered by the user in the text field, and an integer variable that will hold the sum of the numbers.

This applet should contain two methods: init and actionPerformed. In the init method create a label directing to the user to enter an integer. Add this component to the applet as we did with applications in the last chapter. In the same method create a text field with a length of 10. Associate to this component a listener through the method addActionListener, as we also did in chapter 8. Then, add the object to the applet. In the init method you have to initialize to 0 the variable that will hold the total sum of the numbers.

When the user types a number from the keyboard into the text field and presses the Enter key, an event will be sent to the applet and automatically the method actionPerformed will be called to process the user's interaction. So you have to implement this method. As you know, this method receives an object of type ActionEvent, let's call it e, as an argument. The first action that you have to do in this method is to extract from the text field object the number that the user entered. The method getActionCommand applied to the event, i.e., to the object e, returns the characters that the user typed as a String. Then, we can use the method parseInt of the wrapper class Integer to convert the string to an integer. For example, if n is a variable of type int, and e is the event received by actionPerformed, the extraction of the number typed by the user from the text field will look like:

```
    n = Integer.parseInt(e.getActionCommand());
```

Thus, the method actionPerformed should take this number, add it to the total, set the text field object with an empty string and show the sum on the status bar of the window. Text fields objects have the method setText, which takes some text as argument. So, to clear the text field you will write setText(“”).

To display information on the status bar of the window, there is a method called showStatus, which takes as an argument a String value that will be shown. Thus, you have to convert the value of the integer variable that holds the sum of the number to a String value. If t is an int variable, you display this value on the status bar with the following Applet's method:

```
    showStatus(Integer.toString(t));
```

After you finish writing the source code of the applet, create the .HTML file, compile the .java source file using the JDK compiler and run your applet.

4. Understanding another applet

This exercise presents a program that is similar to the previous one. It uses if statements to compare two numbers entered into text fields by the user. If the condition in any of these if statements is satisfied, the drawString statement associated with that if is executed. When the user presses the Enter key in the second TextField, the numbers are read into the variables `number1` and `number2`. The comparisons are performed in the `paint` method, which is called when the `repaint()` method is executed.

Write the following program, compile it, write the .HTML file and run it. Also, respond the questions at the end of the exercise.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Comparison extends Applet implements ActionListener
{

   Label prompt1;  // prompt user to input first value
   TextField input1;  // input first value here
   Label prompt2;  // prompt user to input second value
   TextField input2;  // input second value here
   int number1, number2;  // store input values

   public void init()
   {
      prompt1 = new Label("Enter an integer");
      add(prompt1);

      input1 = new TextField(10);
      add(input1);

      prompt2 = new Label("Enter an integer and press Enter");
      add(prompt2);

      input2 = new TextField(10);
      input2.addActionListener(this);
      add(input2);
   }

   public void paint(Graphics g)
   {
      g.drawString("The comparison results are:", 70,75);

      if (number1 == number2)
        g.drawString(number1 + " == " + number2, 100,90);
      else
        g.drawString(number1 + " != " + number2, 100,90);

      if (number1 < number2)
        g.drawString(number1 + " < " + number2, 100,105);

      if (number1 > number2)
        g.drawString(number1 + " > " + number2, 100,120);

      if (number1 <= number2)
        g.drawString(number1 + " <= " + number2, 100,135);

      if (number1 >= number2)
      g.drawString(number1 + " >= " + number2, 100,150);
   }

   public void actionPerformed (ActionEvent e)
   {
      number1 = Integer.parseInt(input1.getText());
      number2 = Integer.parseInt(input2.getText());
      repaint();
   }
}
```

- Why doesn't `input1` have a listener object associated to it?
- Why does this program implement the method `paint` in contrast to the applet created in the exercise 3, which does not implement it?

Chapter 10

Mutimedia in Java: Images, Animation and Audio

In this chapter you will create simple applets that incorporate multimedia elements, such as images, animation and audio. Please copy the following images and audio files to your hard drive for the exercises that follow:

bird.gif burger.gif surfing.gif
T0.gif T1.gif T2.gif T3.gif T4.gif
T5.gif T6.gif T7.gif T8.gif T9.gif
spacemusic.au

1. Images

Your are going to create an applet that loads an image into an applet, displays the image in its original size and displays the image scaled to twice its original width and twice its original height. You have to include in your program statements to:

1.1. Declare in your class a private data member called `myimage`, which is an instance of the class `Image`.

1.2. Define the method `init` for loading your image into the applet; it has to use the Applet's method called `getImage`. For example, if the name of your image is "surfing.gif" (you can use another image), the method `init` must include the statement:

```
myimage = getImage(getDocumentBase(), "surfing.gif");
```

This version of `getImage` takes two arguments: a location where the image is stored and the file name of the image. In the first argument, the `Applet`'s method `getDocumentBase` is used to determine the location of the image on the Internet (or on your computer if that is where the applet came from). We assume that the image to be loaded is stored in the same location as the HTML file that invoked the applet. The method `getDocumentBase` returns the location of the HTML file on the Internet as an object of class URL (Uniform Resource Locator), which is a standard format for an address of a piece of information on the Internet. Java currently supports two image formats: Graphics Interchange Format (GIF) and Joint Photographic Experts Group (JPG).

1.3. Define the method `paint` for displaying the image; it has to use the `Graphics`' method `drawImage` as explained next.

1.3.1. Display the image in its original size, including in the method `paint` the following statement, assuming that `g` is the `Graphics` object received by `paint`:

```
g.drawImage(myimage, 1, 1, this);
```

The first argument of the method `drawImage` is a reference to the `Image` object in which the image is stored. The second and third arguments are the *x* and *y* coordinates of the upper-left corner where the image should be displayed on the applet. The last argument is a reference to an `ImageObserver` object, which normally is the object on which the image is displayed (`this` is used to indicate the applet).

1.3.2. Obtain the width and the height of the image, including in the method `paint` the following statements:

```
int width = myimage.getWidth(this);
int height = myimage.getHeight(this);
```

1.3.3. Display the scaled version of the image, including in the method `paint` the following statement:

```
g.drawImage(myimage,1,height+1,width*2,height*2,this);
```

The second and third arguments of the method are the *x* and *y* coordinates where the image should be displayed, and the fourth and fifth arguments specify the width and height of the image for display purposes.

1.4. Compile your program using JDK, write the HTML file and run your applet using any browser.

2. Animations

The next applet demonstrates a simple animation. The applet uses the same technique to load and display images as used in the last program. This applet maintains an array of images that are loaded in the applet's `init` method. Include in your program statements to:

2.1. Declare in your class a private data member called `myImageArray`, which is an array of instances of the class `Image`.

2.2. Declare the following three integers as private data members of your class:

```
totalImages = 10   // total number of images
currentImage = 0   // current image subscript
sleepTime = 150    // milliseconds to sleep
```

2.3. Define the method `init` to load the images into the applet.

2.3.1. First, assign memory to the array, creating it with a size corresponding to `totalImages`.

2.3.2. Then, load all the images into the array using a `for` loop. The names of the images are "t0.gif", "t1.gif", etc. Each name is constructed concatenating its parts using the symbol "+". For example, to load the image "t2.gif" into the array, you use the following statement:

```
myimagearray[2] =
   getImage(getDocumentBase(), "t" + 2 + ".gif");
```

2.4. As you know, applets always begin with a series of three method calls: `init`, `start` and `paint`. The predefined startup sequence of method calls made by the browser for every applet is always `init`, `start` and `paint`. Define in your class the public method `start`, which does not receive any argument, and initialize within this method the variable `currentImage` with a value of 0. This indicates that the applet always starts with the first image.

2.5. The work of displaying the animation in this example is performed by the Applet's `paint` method. The instance variable `currentImage` keeps track of the image to display. Define the method `paint` including statements for:

2.5.1. Displaying one frame, i.e., the image of the array indicated by the variable `currentImage`, as you displayed the image in the exercise 1.

2.5.2. Updating the value of the `currentImage` variable to make it to reference to the next image in the array. Note that when `currentImage` is making reference to the last image, it has to change to the first one of the array. This calculation is obtained with the following statement:

```
currentImage = (currentImage+1) % totalImages;
```

2.5.3. Making the applet sleep for a fraction of a second, so the animation can be seen clearly:

```
try {
     Thread.sleep(sleepTime);
}
catch (InterruptedException e)  {
     showStatus(e.toString());
}
```

In Java, a thread designates a portion of a program that may execute concurrently with other threads. Threads make concurrency primitives available to the personal computer applications programmer. One of the methods of the class `Thread` is called `sleep`. This method is called with an argument specifying how long the currently executing thread should sleep (in milliseconds); while a thread sleeps, it does not contend for the processor, so other threads can execute.

The code that may generate an exception is enclosed in a `try` block followed by a `catch` block.

2.5.4. Calling the method `repaint`, which in turn invokes the `paint` method to display the next image.

2.6. Compile your program using JDK, write the HTML file and run your applet using any browser.

3. Sounds

Java programs can manipulate and play audio clips. Java provides two mechanisms for playing sounds in an applet: the `Applet`'s `play` method and the `play` method from the `AudioClip` interface. The `Applet`'s method `play` loads the sound and plays it once. For example:

```
play(getDocumentBase(), "hi.au");
```

The second technique is more flexible. It allows the audio to be stored in the program so the audio can be reused throughout the program's execution. This is the technique that is used in the next program.

This program uses three components GUI (buttons) to interact with the user. These buttons are used to play the sound once, to play the sound in a loop, and to stop the loop. You need to have the file "spacemusic.au" in the same directory as your source code. You have to do the following in your applet:

3.1. Your class has to implement the interface `ActionListener`.

3.2. Declare a data member called `sound` whose type is the class `AudioClip`.

3.3. Declare 3 buttons called `playSound`, `loopSound` and `stopSound`.

3.4. In the `init` method make the variable `sound` reference the file "spacemusic.au" using the following statement:

```
sound =
getAudioClip(getDocumentBase(),"spacemusic.au");
```

3.5. In `init` create the three buttons with the following labels respectively: "Play", "Loop" and "Stop". Assign listeners to them and add them to the applet.

3.6. Assuming that the method `actionPerformed` receives as a parameter an event called `e`, define this method with the following code:

```
if (e.getSource() == playSound)
     sound.play();
else if (e.getSource() == loopSound)
     sound.loop();
else if (e.getSource() == stopSound)
     sound.stop();
```

3.7. Compile the applet, create the .HTML file and run it using any browser.

Chapter 11

Visual J++ (based on version 6.0)

This chapter is an introduction to Visual J++. Its purpose is that you become familiar with its environment for the development of different types of Java programs. Thus you will create very simple programs that will show you the basic features of the tools for creating console applications, windows applications and applets.

1. Creating a console application

Console applications are applications that do not use windows to display information and to interact with the user. They are designed to run from the DOS command or the UNIX prompt.

You will create a simple console application in order to become familiar with the Microsoft Development Environment (MDE) of Visual J++.

In Visual J++, in order to create a program you have to build a project that is a collection of source files. Let's do it following the next steps:

- In the MDE select *New Project* from the *File* menu.
- The *New Project* dialog box lists some folders. In the *Visual J++ Projects* folder, choose the *Applications* folder and then select *Console Application* (with one click only).
- In *Location*, write the path of YOUR OWN DIRECTORY where you want to store your programs.
- In *Name*, write a name for your project changing the default name. Note that your project will be saved in a subfolder with the same name as the project.
- Finally, choose the button *Open* to build your project.

Now you have a project. You can use the tool *Project Explorer* to look at the resulting project. If you already don't see a window in the MDE showing the *Project Explorer*, choose *Project Explorer* from the *View* menu. To view the files in your project, click the plus sign next to the project name. As you can see, there is a single file called *Class1.java*. This file will contain your source code. Double-click this file to view the empty class definition that Visual J++ generated for you.

In the main method write a statement to display a message on the screen (any message you want).

The rest is easy. To compile your application, choose *Build* from the *Build* menu. You should see in the status bar the phrase *"Solution update succeeded"*; this means that your program compiled properly.

Now run the program choosing *Start* from the *Debug* menu. Note that your application will run in a MS-DOS window, which will be displayed and closed quickly and you could not see the output. You can fix this problem after studying the next section.

2. Using the WFC I/O package

Visual J++ has some features that are characteristic to this programming language and that are not recognize by the standard Java. This is the case of the library WFC (Windows Foundation Classes), which is a set of classes unique to Microsoft Windows and to Microsoft Visual J++. Note that if you use any of the classes of the WFC in your program, your program will compile only with Visual J++ and not with other compilers. In this section we will present briefly one of the package of the WFC, which is the input/output package. In order to use any of the classes defined in this package you have to import *com.ms.wfc.io.**.

The most basic input/output is to access the keyboard for input and the screen for output. You can use the class *Text* defined in the I/O package to do the standard I/O. The *Text* class has the public static members *in*, and *out*, which allow us to perform the standard I/O.

Text.out is an object that handles character output to the screen through the *writeLine()* and *write()* methods. In the other hand, *Text.in* is an object that handles character input from the keyboard using the method *readLine()*, which returns a *String* object. In this way, if you want to ask the user for his/her name, you will write some code like:

```
Text.out.writeLine("Please enter your name:");
String name = Text.in.readLine();
```

Remember that for converting a *String* to an *int* value, you have to convert the *String* to an *Integer* and then to an *int*, as we did in chapter 6.

Write an application that asks the user for 5 integer numbers and that displays the average of them.

3. Creating a windows application

In previous chapters we built windows applications using the Abstract Windowing Toolkit (AWT). In this chapter we will learn how to use the Windows Foundation Classes for Java (WFC) provided by Visual J++ to create a simple windows application quickly and easily.

- Select *New Project* from the *File* menu.
- Choose *Applications* from the *Visual J++ Projects*.
- Select *Windows Application*.
- Enter a name for your project.
- Choose *Open*.

This creates a project with a single source file, called by default *Form1.java*. If you double-click this file for editing, you will see the *Forms Designer*, which allows to design the application graphically.

To start to build your application you need two more windows: the *Toolbox* and the *Properties* windows. To open the *Toolbox* select *Toolbox* from the *View* menu. By clicking the WFC tab, this window shows the WFC controls that you can include in the form or window that you are creating.

The first control that you have to include in your form is a label asking for a file name. In order to do this, follow these steps:

- Click the *Label* control in the *Toolbox*.
- Click a spot within the *Forms Designer*.
- Enter the text `File Name` in the label that is displayed in the *Forms Designer*.

As soon as you do this, the *Properties* window appears. This window shows all the properties of the control selected in the *Forms Designer*. Using this window you can change any of the properties of the control, such as color, font, position and the actions that the control will perform.

Now, you have to add two *Edit* controls, one for containing the file name and the other one for showing the content of the file. Once you add the *Edit* control for the file name next to the label you already added, select the *Properties* window to change some properties of this control. In the list there is a property called *text* that has the phrase *edit1*. Since you don't want the *Edit* control to show this default phrase as the initial value, delete this text.

Include another *Edit* object, which will show the content of the file. This control has a property called *multiline*. Set it to *true*. This will allow you to resize the control vertically and horizontally. Do the same with the *text* property as you did for the other *Edit* control.

The last controls to add are two buttons. Include one and set its *text* property to the phrase *Open File*. Add a second one with its *text* property to the phrase *Cancel*.
Finally, to set the form's title, select the form and change its *text* property to the title you would like.

At this point you have finished the design of how your application will look, even when it doesn't do any important work. Before adding action to your application let's see the code that the Windows Application builder generated for your form.

- Select *Save All* from the *File* menu.
- Choose *Code* from the *View* menu.

Now you can see the results of your work building the project and executing the .EXE file. As you can see, we need to define the functionality of the application. The *Cancel* button should close the application and the *Open File* button should show the content of the file specified by the user.

In order to do this, in the *Forms Designer* double-click the *Cancel* button. This opens the text editor where you have to write the code that will be executed when the user clicks on the *Cancel* button. This code will correspond to the method `button2_click()`, assuming that the Windows Application builder called the *Cancel* button with the name *button2*. These names are assigned by default automatically. Thus modify the method `button2_click()` so that it looks like the following code:

```
private void button2_click(Object source, Event e)
{
     dispose ();
     Application.exit ();
}
```

The method `dispose()` closes all open windows in the application, and the method `Application.exit()` ends the application. Now you can test the functionality of the *Cancel* button. Rebuild and execute it.

When the user clicks the button *Open File* (called by the Windows Application builder with the name *button1*), the method `button1_click()` takes from the first *Edit* object the name of the existing file entered by the user and displays its content in the second *Edit* object that you added.

So in the *Forms Designer*, double-click the button *Open File* and modify the method `button1_click()` with the following code:

```
private void button1_click(Object source, Event e)
{
     // Get the name of the input file.
     String fileName = edit1.getText();

     if (!fileName.equals(“”))
     {
          // Open the file for reading.
          TextReader f = new TextReader(fileName);

          // Read a line at a time and add each line to
          // an array of strings.
          String line;
          String[] sarray = new String[1];
          for (int i=0; (line=f.readLine()) != null; i++)
          {
               if (i >= sarray.length)
               {
                    // Double size of the array.
                    sarray = double_array(sarray);
               }
               sarray[i] = line;
          }

          // Show the lines in the edit control.
          edit2.setLines(sarray);
          }
     }
```

You can see that there is a call to the method `double_array()` in the previous code. This method doubles the size of the array and we have to define it completely. In order to do this, save all the files and close the *Forms Designer*. Then open the .java file (`Form1.java`) with the text editor (not with the *Forms Designer*) to edit the code.

Modify the existing code adding the following method[1]:

```
     /**
      * Double the length of the array passed.
      */
     public static String[] expand(String[] array)
     {
          // Allocate an array twice as big.
          String[] newArray= new String[2*array.length];

          // Null out the array first.
          for (int i=0; i<newArray.length; i++)
          {
               newArray[i] = null;
          }

          // Then copy over members from the source array.
          for (int i=0; i<array.length; i++)
          {

               newArray[i] = array[i];
          }
          return newArray;
     }
```

Finally, through the text editor, add an *import* statement at the top of the `.java` file to import `com.ms.wfc.io.*`, which allows the program to use the `TextWriter` class.

Your application is ready. Save all the files, build the project and execute the `.EXE` file.

4. Creating an applet

The purpose of this section is to provide an introduction to the tools that Visual J++ provides for writing applets. Its intend is not to cover how to write applets (this was done in chapter 9), so we will explain the basic features with the simplest applet: to display a message.

[1] Taken from the book *Programming Microsoft Visual J++ 6.0*, written by Stephen R. Davis.

We have two options for writing an applet in Visual J++: using the *Applet Wizard* and not using it. We will start with the second option, which requires to create an empty project.

- Select *New Project* from the *File* menu.
- Select *Visual J++ Projects* in the left pane of the *New Project* window.
- Choose *Empty Project* in the right pane.
- Enter the project path and name.
- Click *Open*.

In the Project Explorer you can check that you have an empty project without any file. You need to include the .java file.

- With your project selected, choose *Add Item* from the *Project* menu.
- From the *New* tab in the *Add Item* window, select *Class* and enter the class name (the same name as the project).

Now you have a .java file with an empty class. Modify it to look like the following code:

```
import java.applet.Applet;
import java.awt.Graphics;

public class NameOfYourClass extends Applet
{
     public void paint(Graphics g)
     {
          g.drawString("This is just a test for
                         an applet...", 10, 10);
     }
}
```

The previous code doesn't need explanation given that we already covered applets in another chapter.

Remember that we have to create an HTML file for the applet, which will be included as part of the project.

- In the *Project Explorer* select *AddItem*.
- Select *Web Page*.
- Enter a name for the file in the *Name* text box. For example, if your class' name is `MyApplet`, give to the HTML file the name `MyApplet.htm`.
- Click *Open*.

A special text editor for HTML files appears. Note that it has three tabs at the bottom: *Design*, *Source*, and *Quick View*. The *Design* tab allows you to use the tools from the HTML section of the *Toolbox* for "painting" the HTML page. The *Source* tab allows you to write the source code directly. And the *Quick View* tab allows you to view most of the features of the HTML page without opening a browser.

We will use the *Source* tab to modify the default HTML code. Include the necessary statements so that it looks like the following:

```
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="Microsoft Visual
          Studio 6.0">
<TITLE>Test Applet</TITLE>
</HEAD>
<BODY>
<HR>
<OBJECT CODE="NameOfYourClass.class"
     HEIGHT=80 WIDTH=120>
</OBJECT>
<HR>
</BODY>
</HTML>
```

Notice the use of the tag `<OBJECT>` instead of `<APPLET>`. Both have the same function.
The last step is to execute the applet, which requires to set the launch properties:

- Choose *NameOfYourClass Properties* from the *Project* menu.
- In the *When Project Runs*, *Load* drop-down list, select *NameOfYourClass*.htm.
- Choose *OK*.
- From the *Debug* menu select *Build* to compile your project.
- From the *Debug* menu select *Start* to run the program.

Visual J++ opens the browser to execute the .htm file and your applet.
The other option to write applets is using the *Applet Wizard*. The reason we didn't use this tool to write the previous applet is that the wizard generates more default code that we needed for our very simple applet. We won't write again the same applet, we will just check the default code that this tool generates. Follow these steps:

- From the *File* menu, select *New Project*.
- From the *New Project* window, select *Web Pages* under the *Visual J++ Projects* folder.
- Choose *Applet On HTML* in the right-hand pane.
- Enter the project name.
- Click *Open*.

You can see that this creates a project with the `.java` and the `.htm` files. Open both files and check out the default code that the *Applet Wizard* generated for you. This code can be the starting point for any applet you want to create.

Chapter 12

Simple Data Types in Java

Now that you have significant experience with Java, it is time to put all that to work in developing simple data types such as the ones you already did for the lecture part of the course. We are talking about stacks, queues, lists, and simple binary search trees. There is no need to describe these as you should already know what they are and what they do. So, let's get started.

Implementing an abstract data type in Java requires two steps: 1) definition of a Java API (interface) which simply describes the names of the methods that the abstract data type supports. This also includes how these methods are to be declared and used. 2) A class that implements the methods of the interface associated with the desired ADT.

1. The Stack (1)

We will make things easy on ourselves and only require that we include the following three functions in our stack data type representation: top(), size(), push(), and pop(). Of course, we now have to refer to these as *methods*, not functions. The interface to be used for the stack ADT is now as follows:

```
public interface Stack  {
     public Object top();  // returns the top element
          throws StackEmptyException;
     public int size()
     public void push(Object element);
     public Object pop();
          throws StackEmptyException;
}

public class StackEmptyException extends RuntimeException
{
     public StackEmptyException(String err)  {
          super(err);
     }
}
```

Declare an exception class called StackFullException and update the interface to include the exception for pushing into a full stack.

Now we need to declare a concrete class that implements the methods of the interface associated with the Stack ADT. This can be done as follows for an array-based stack that can hold up to 1000 items:

```
public class ArrayStack implements Stack  {
     public static final int CAPACITY = 1000; // default
                         // capacity of stack
     private int capacity; // actual capacity of stack if
                         // not 1000
     private Object S[];  // array that holds the items in
                         // the stack
     private int top = -1;  // the top element of the stack

     public ArrayStack()  {
          this(CAPACITY);  // default constructor
     }

     public ArrayStack(int cap)  {
          capacity = cap;
          S = new Object[capacity];
     }

     public int size()  {
          return (top + 1);
     }

     public void push(Object obj)  {
          if (size() == capacity)
               throw new StackFullException("Stack overflow");
          S[top++] = obj;
     }

     public Object top() throws StackEmptyException   {
          // why is it passing the buck,
          // and to which methods is it passing it???
          if (size() == 0)
               throw new StackEmptyException("Stack is empty");
          return S[top];
     }

     public Object pop() throws StackEmptyException   {
          // why is it passing the buck,
          // and to which methods???
          Object elem;
          if (size() == 0)
               throw new StackEmptyException("Stack is empty");
          elem = S[top];
          S[top--] = null;
          return elem;
     }
}
```

Please implement this stack ADT and write a short program that uses the stack ADT to reverse a string containing characters that spell out your first and last names.

The use of the Object data type is to make the stack generic. That means we can store integers, floating points, or any other user-defined data type in the stack. Any elements stored in the stack are objects of the Object class in Java, which is the superclass of all classes in Java. This is not a problem when we are adding elements to the stack. However, when we retrieve from the stack (using `top()` or `pop()`), then we want to ensure that we are retrieving the correct data type for use in an I/O operation or as a value to be assigned to a variable declared of a specific type. In order to carry this out, we must perform a cast operation. This forces the object to be seen by the program as a member of a specific class. This need became evident in the previous exercise when you attempted to display the contents of the stack containing the reverse of your name. *** must confirm *** Cast the results of the reverse of your name as character elements.

2. The Queue

Do the same for the Queue that you did for the Stack. Implement the Queue ADT using an array.

1) Define an interface called `Queue` that prototypes the following methods:
    - `enqueue()`
    - `dequeue()`
    - `size()`
    - `front()`
    - `full()`
    - `empty()`

2) Define a concrete class called `ArrayQueue` that implements the methods prototyped in the `Queue` interface.

3) Form a queue of aircraft waiting to land at OIA. Each aircraft is identified by a symbol consisting of 2 characters (the airline code) immediately followed by a 3-digit flight number. For example, Delta flight 345 is symbolized as DL345. Write a program that takes aircraft from the keyboard and enqueues them. It dequeues one each minute. Add five flights consecutively and watch the queue grow and then shrink as the aircraft land, one every minute. The flights are: AA123, DL345, UA456, NW678, CO789. Display the queue after each enqueue or dequeue operation.

3. The List (1)

The list is a sequence of elements, where operations can be performed on every element, regardless of its location. This is also called a ranked sequence. Once again, you should be familiar with these already. We will now deal with linked lists, rather than array-based. However, we will be implementing a doubly-linked list in this exercise.

The first class to implement is the node class. This is the class whose instance objects will "carry" the information and compose the elements of the linked list.

```
class DoubleNode  {
     private Object element; // holds the information
     private DoubleNode next; // note that it is NOT a
pointer
     private DoubleNode previous;
     DoubleNode() { this(null,null); }
     Public DoubleNode(Object e, Node p, Node n)  {
          element = e;
          next = n;
          previous = p;
          }
     public void setElement(Object newElem) {element =
newElem;}
     public void setNext(Object newNext)  { next = newNext;
}
     public void setPrevious(Object newPrev) { previous =
           newPrev;}
     public Object getElement()  {  return element;  }
     public DoubleNode getNext()  { return next;  }
     public DoubleNode getPrev() { return previous; }

     .........
     }
```

What other methods are deemed necessary for this class? Please add them to this class.

Next, you will need to develop the API for the list data type:

```
public interface List  {
     public int size();
     public Boolean empty();
     public void insert(Object element, Node before)
     public void delete(Node n);
     .....
}
```

Add any other method that may be needed to this interface.

Now you must include the class `LinkedList` class that implements the `List` API.

```
public class LinkedList implements List  {
     private DoubleNode head;
     private DoubleNode tail;
     private int size;
     public LinkedList   {
          head = null;
          tail = null;
          size = 0;
          }
     ........
}
```

Complete the above class definition. Include the definition of an exception class for an empty list. The methods `delete()` should be defined to throw an exception when the list is empty.

Write a short program in Java that keeps an alphabetically-ordered list of the names of the students in a class. As a new name is added, it is added to the list and put in the proper alphabetical position in the list. Add the following names one by one, starting from an empty list:

Bill, Jim, Karen, Jenny, Sue, John, Alex, Zack, and Alina.

Print out the list at the end of each addition.

4. The Binary Search Tree [(2)]

This section will focus on building a binary search tree data type in Java. We will begin with defining the TreeNode class, which, as the name suggests, defines the nodes to be used in the tree. For the purposes of simplicity, we will only add integers data to the tree. It can be easily changed to be more general by either defining a datatype to include any object type, or by using the `Object` data type available in Java.

```
Public class TreeNode  {
     private int element;
     private TreeNode left, right;
     TreeNode(int newElement)
     {
          element = newElement;
     }
}
```

Remember that Java automatically initializes all objects to `null`, Therefore, there is no need to explicitly initialize the right and left pointers to `null`.

Having the basic building block for the tree, now we can proceed to define a class for the binary search tree. This can be done very simply as follows:

```
Public class BinTree     {
     Private TreeNode root;
     Private TreeNode Insert(TreeNode tree, int newElement)
     {
          if (tree == null)
               return new TreeNode(newElement);
          else if (newElement < tree.element)
               tree.left = Insert(tree.left, newElement);
          else
               tree.right = Insert(tree.right, newElement);
          return tree;
     }
     public TreeNode Insert(newElement)
     {
          root = Insert(root, newElement);
     }
}
```

Note that there are two methods for inserting into the tree. One is public and only has one argument – the element to be inserted. The other one is private and has two arguments, the tree into which to make the insertion, as well as the new element to insert. The reason this is done is to facilitate the recursive traversal of the tree without having to continually have to name the root of the subtree being traversed in the recursive call to Insert(). In the public function, root is assigned the value returned by the private Insert() method. This is to define the root of the main tree and its contents in case it has not yet been defined. ….. needs bette definition

Add a method to the BinTree class defined above to search the tree for a particular key. It should be named `BTSearch()` and have as its argument an integer key to be found.

Build a binary search tree consisting of the following keys:

12, 34, 9, 27, 17, 97, 98, 99, 23, 51, 3, 28, 28, 19

and exercise its search feature by requesting the following searches.

28, 23, 19, 28, 30

---

(1) Java code obtained from Goodrich and Tamassia, Data Structures and Algorithms in Java, New York, NY: John Wiley and Sons, Inc., 1998

(2) Java code obtained from Rowe, An Introduction to Data Structures and Algorithms with Java, London, UK: Prentice Hall Europe, 1998