The Abdus Salam
International Centre for Theoretical Physics

United Nations
Educational, Scientific and
Cultural Organization

IAEA
International Atomic Energy Agency

1875-3

First Latin American Regional Workshop on Distributed Laboratory Instrumentation in Physics

7 January - 4 February, 2008

Introduction to JAVA
-supporting material-

Carlos Kavka
INFN Sezione di Trieste
Area di Ricerca
Padriciano 99
34012 Trieste
Italy

Introduction to Java

Carlos Kavka*

INFN Sezione di Trieste
Area di Ricerca
Padriciano 99 – 34012 – Trieste
Italia.

Supporting material for the
lectures given at:
The First Latin American Workshop on
Distributed Laboratory Instrumentation Systems
Universidad Austral de Chile
Valdivia, Chile, 7 January – 1 February 2008

LNS

*Carlos.Kavka@ts.infn.it

Abstract

This chapter is intended to serve as an introduction to programming in *Java*. It is by no means a complete treatment of the *Java* language or its Application Programming Interface (API). The readers are encouraged to supplement this material by referring to standard texts and consulting the *Java* Software Development Kit (SDK) and API documentation [1].

After a brief introduction to the features of the language that will be relevant to the topics covered in this Workshop, more advance topics such as multithreading, use of Ant and running *Java* on the TINI (Tiny InterNet Interface) hardware are illustrated with suitable example code.

Appendices contain source code of the examples discussed in this chapter.

[1] http://java.sun.com/reference/api

Contents

1 Introduction

Java is a very powerful programming language that has generated a lot of interest in recent years. It is a general purpose concurrent object oriented language, with a syntax similar to C (and C++), but omitting features that are complex and unsafe.

Its main advantage is the fact that the compiled code is independent of the architecture of the computer. The world wide web has popularized the use of *Java*, because programs written in this language can be transparently downloaded with the web pages and executed in any computer with a *Java* capable browser.

However, *Java* is not limited to Web based applications. In fact, it has been used extensively in other domains, including microcontroller applications.

A *Java* application is a standalone *Java* program that can be executed independently of any web browser. A *Java* applet on the other hand is a program designed to be executed under a *Java* capable browser. In this introduction to *Java*, we will not cover applets.

Java was developed by Sun Microsystem in 1991, as part of a project that was developing software for consumer electronic devices. The current version is *Java* 6.0 (also known as Java SE 6) and Sun provides the JDK (*Java* Development Kit) freely through its web site[1]. This has certainly contributed to the popularity of the *Java* language.

These lecture notes assume that you have some familiarity with C. In fact, usually *Java* can be learnt easily than C or C++ due to the fact that most of the complex aspects of C that can cause errors are not present in *Java*. We will not be covering all aspects of *Java*, and in particular, we will not be covering applets and interface design.

All examples used in these notes are available for experimentation. In fact, they serve to complement these lecture notes and you are encouraged to execute and modify the examples in order to fully understand the concepts involved.

2 The *Java* platform

Java programs are compiled into *Java* byte-codes, a kind of machine independent representation. The compiled program is then executed by an interpreter called the *Java* Virtual Machine (JVM). The *Java* Virtual Machine is an abstract computer with its own instruction set and memory areas. A *Java* compiled program can be executed in any computer system that has a JVM.

The main advantage of this approach is, of course, portability. That is, the same *Java* compiled program can be executed in any computer that has a JVM. The price to pay is slower execution due to the use of an interpreter. More recent Just In Time (JIT) Compilers make *Java* programs to execute at comparable speeds.

[1] java.sun.com

In this way, *Java* byte codes help to make "write once, run everywhere" possible.

3 A first example

This section presents a short example, the usual Hello World application. The program when compiled and run just prints the message "Hello World!".

```
/**
 * Hello World Application
 * Our first example
 */

public class HelloWorld {

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

This example, though short, involves a lot of concepts. We will just be considering the main aspects now. The rest will be covered in more detail in later sections.

The application consists of only one class. Its name is `HelloWorld` and it has to be defined in the file `HelloWorld.java`. The Java compiler requires that the name of the class must be the same as the name of the file without the extension.

The class defines only one method, called `main`, that has to be defined exactly as shown in the example:

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

The `main` method receives an array of strings as argument and returns nothing. This is the place where the execution will begin.

The class `System` is defined in the *Java* API (Application Programming Interface) and it is used to provide access to the system functionality. The class variable `out` is a member of the class `System` and is used to create the stream object `System.out` which allows access to the standard output stream. The method `println` is called in order to print the string passed as an argument on the standard output:

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

There exists two types of standard comments for documenting programs, and they are ignored by the compiler. Single line comments begin with the characters `//` and multi-line comments are defined between the characters `/*` and `*/` like in C.

There exists a third type of comments that are used by the special documentation utility `javadoc`. They are defined between the characters `/**` and `*/`, like in the example:

```
/**
 * Hello World Application
 * Our first example
 */
```

HTML and special commands can be included within this type of comments, and they are interpreted by `javadoc` in order to automatically generate documentation that can be displayed on a web browser.

4 Development cycle for *Java* applications

To develop a *Java* application we have to follow three steps: creation of the source file, compilation and execution. The following example is based on the use of Sun Microsystems JDK.

Creation of the source file : This can be done with any text editor. The name of the file must have the same name as the class that is going to be defined, with the extension `.java`. *Java* is case sensitive and the capitalization of the words will be recognized by the compiler. One possibility is the use of the `emacs` editor as shown below (# is the UNIX command prompt):

```
# emacs HelloWorld.java
```

Compilation : The *Java* compiler, when invoked using the command `javac`, will translate the source file into a file containing bytecodes that can be executed by a *Java* Virtual Machine:

```
# javac HelloWorld.java
```

This will create a file with the same name and the extension `.class`:

```
# ls
HelloWorld.java
HelloWorld.class
```

Execution : The class file containing the bytecodes will be interpreted by the *Java* Virtual Machine. In order to execute the application, we have to call the program `java` with the name of the class as argument (without extension):

```
# java HelloWorld
Hello World!
```

The `javadoc` utility can be used to automatically generate documentation for the class, and for all of its components. The following command creates a set of HTML files that describe the class `HelloWorld`.

```
# javadoc HelloWorld
```

The main output file is `HelloWorld.html` and it is shown in figure 1.

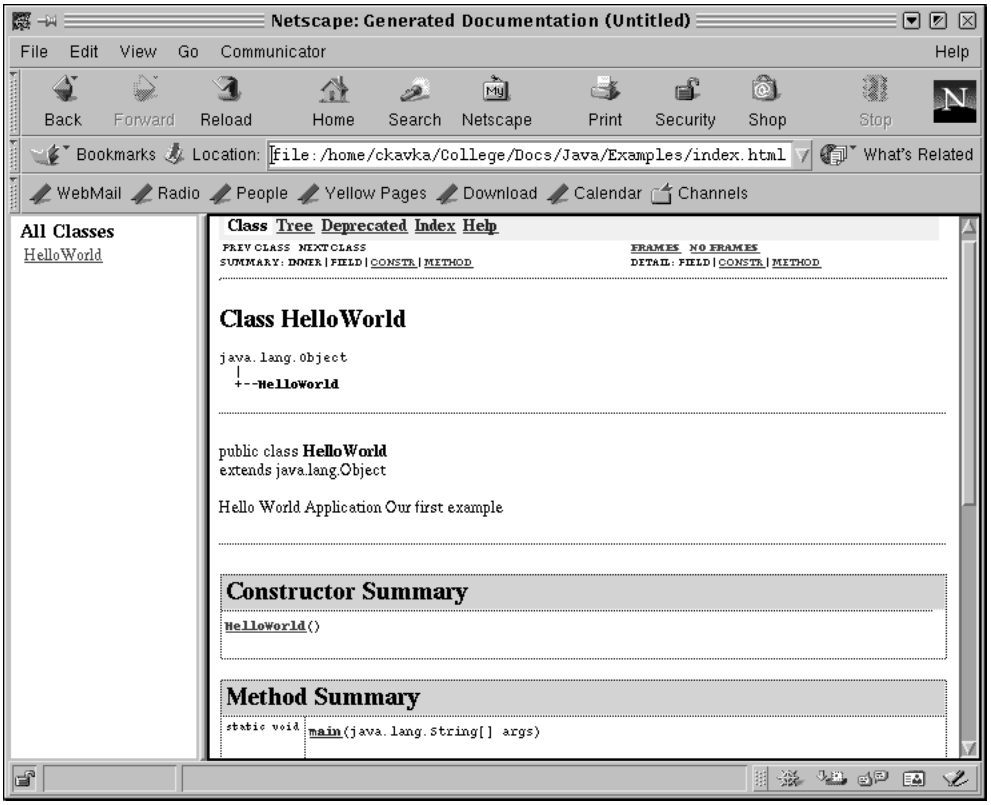

Figure 1: Documentation generated by javadoc

5 Fundamental data types

Java provides ten primitive data types: four types of integers, two types of floating point numbers, characters, booleans, the special type `void` and strings. Table 1 presents for some data types, their minimum and maximum values and the size of the objects of these types.

The type `boolean` consists of two values: `true` and `false`. There is no equivalence with integer values like in C.

The type `void` is used as the return type of a method that returns nothing, like the method `main` declared in the first example:

type	size	min value	max value
byte	8 bits	-128	127
short	16 bits	-2^{15}	$2^{15}-1$
int	32 bits	-2^{31}	$2^{31}-1$
long	64 bits	-2^{63}	$2^{63}-1$
float	32 bits	$1.4\text{E}-45$	$3.45\text{E}38$
double	64 bits	$4.9\text{E}-324$	$1.7\text{E}308$
char	16 bits	unicode 0	unicode $2^{16}-1$

Table 1: Some characteristics of Java primitive data types

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

The type `String` specifies sequences of characters, and it is not related to arrays like in C.

The type `char`, representing characters, is 16 bits long, and allows to work with the standard set of ASCII characters, plus an enormous amount of multilingual characters.

We will consider more details of the types in following sections.

6 Variables

Variables can be declared by specifying its type and name. They can be initialized at the point of declaration, or a value can be assigned later with the assignment expression, as shown below:

```
int x;                    // not initialized
double f = 0.33;
char c = 'a';
String s = "abcd";

x = 55;                   // value assigned
```

7 Literals

The integer values can be written in decimal, hexadecimal, octal and long forms, as shown in the next example:

```
int x = 34;               // decimal value 34
int y = 0x3ef;            // hexadecimal value 3ef
int z = 0772;             // octal value 772
long m = 240395922L;      // long value 240395922
```

The floating point values are of type `double` by default. In order to specify a `float` value, we have to add the letter `F` at the end, as shown below:

```
double d = 6.28;          // 6.28 is a double value
float f = 6.28F;          // 6.28F is a float value
```

The character values are specified in the standard C notation, with the exception that unicode values can be specified with `\u`:

```
char c = 'a';             // character lowercase a
char d = '\n';            // newline character
char e = '\u2122'         // unicode character (TM)
```

The boolean values are `true` and `false`. They are the only values that can be assigned to `boolean` variables:

```
boolean ready = true;     // boolean value true
boolean late = false;     // boolean value false
```

8 Constants

The declaration of constants is very similar to the declaration of variables. It has to include the *Java* keyword `final` in front. The specification of the initial value is compulsory, as shown in the examples below:

```
final double pi = 3.1415;          // constant PI
final int maxSize = 100;           // integer constant
final char lastLetter = 'z';       // last lowercase letter
final String word = "Hello";
```

Of course, once declared, their values cannot be modified.

9 Expressions

Java provides a rich set of operators in order to use in expressions. Expressions can be classified as arithmetic, bit level, relational, logical, and specific for strings. They are detailed in the following subsections.

9.1 Arithmetic operators

Java provides the usual set of arithmetic operators: addition (+), subtraction (-), division (/), multiplication (*) and modulus (%). The following application provides some examples.

```
/**
 * Arithmetic Application
 */
class Arithmetic {

  public static void main(String[] args) {
    int x = 12;
    int y = 2 * x;
    System.out.println(y);
    int z = (y - x) % 5;
    System.out.println(z);
    final float pi = 3.1415F;
    float f = pi / 0.62F;
    System.out.println(f);
  }
}
```

The output produced by the execution of the application is:

```
24
2
5.0669355
```

The last section of this application shows that the variables can be declared at any point in the body of a method. They can then be used to store a value from this point up to the end of the block in which they were defined.

Java provides several compound assignment operators that are composed of the assignment operator and a binary operator. These can be used for abbreviating assignment expressions. The next application presents some examples:

```
/**
 * Shorthand operators Application
 */
class ShortHand {

  public static void main(String[] args) {
    int x = 12;
    x += 5;                    // x = x + 5
    System.out.println(x);
    x *= 2;                    // x = x * 2
    System.out.println(x);
  }
}
```

The output produced by the execution of the application is:

```
17
34
```

A usual operation is to increment or decrement the value of a variable. The operators ++ and −− are provided for that. There are two versions of these operators, called prefix and postfix. For pre-increment and pre-decrement operators, the operation is performed first, and then the value is returned. For post-increment and post-decrement operators, the value is returned, and then the operation is performed.

The following application presents some examples:

```
/**
 * Increment operator Application
 */

class Increment {

 public static void main(String[] args) {
  int x = 12,y = 12;

  System.out.println(x++); // x is printed and then incremented
  System.out.println(x);

  System.out.println(++y); // y is incremented and then printed
  System.out.println(y);
 }
}
```

The output produced by the execution of the application is:

```
12
13
13
13
```

9.2 Relational operators

Java provides the standard set of relational operators: equivalent (==), not equivalent (!=), less than (<), greater than (>), less than or equal (<=) and greater than or equal (>=). The relational expressions always return a boolean value.

The following example shows the value returned by some relational expressions:

```
/**
 * Boolean operator Application
 */
class Boolean {
  public static void main(String[] args) {
    int x = 12,y = 33;
```

```
    System.out.println(x < y);
    System.out.println(x != y - 21);

    boolean test = x >= 10;
    System.out.println(test);
  }
}
```

The output of the program is:

```
true
false
true
```

9.3 Bit level operators

Java provides a set of operators that can manipulate bits directly. Some operators such as *and* (&), *or* (|) and *not* (~) perform boolean algebra on bits. There are others to perform bits shifting: shift left (<<), shift right with sign extension (>>) and shift right with zero extension (>>>).

The binary bitwise operator *and* (&) performs a boolean "and" operation between the bits of the two arguments. The binary bitwise operator *or* (|) performs a boolean "or" operation between the bits of the two arguments. The unary bitwise operator *not* (~) performs a boolean "not" operation on the bits of its argument.

The binary bitwise left-shift operator (<<) shifts the bits of the first argument as many positions as indicated by the second argument inserting 0s from the side of the least significant bit. The binary bitwise right-shift operator (>>>) shifts the bits of the first argument as many positions as indicated by the second argument inserting 0s from the side of the most significant bit. The binary bitwise right-shift operator with sign extension (>>) shifts the bits of the first argument as many positions as indicated by the second argument inserting 0s or 1s from the side of the most significant bit, maintaining the sign of the first argument. This means that 0s are inserted if the number is positive, and 1s are inserted if the number is negative.

These operators operate on integral types. If the argument is a `char`, `short` or `byte`, it is promoted to `int` and the result is an `int`.

The following example shows the value returned by some boolean algebra bit level expressions:

```
/**
 * Boolean algebra bit level operators Application
 */

class Bits {
```

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

  int x = 0x16;             // 00000000000000000000000000010110
  int y = 0x33;             // 00000000000000000000000000110011
  System.out.println(x & y);// 00000000000000000000000000010010
  System.out.println(x | y);// 00000000000000000000000000110111
  System.out.println(~x);   // 11111111111111111111111111101001

  x &= 0xf;                 // 00000000000000000000000000000110
  System.out.println(x);    // 00000000000000000000000000000110

  short s = 7;              // 0000000000000111
  System.out.println(~s);   // 11111111111111111111111111111000
 }
}
```

The example shows that compound assignment operators are also possible by combining the assignment operator and the binary boolean operators (& and |). The comments specify the binary representation of the values obtained at each stage. The last two comments show that even if the value of the argument to "not" is a `short`, the result is an `int`.

The following example shows the value returned by some bit level shift expressions:

```
/**
 * Bit level operators Application
 */
class Bits2 {

 public static void main(String[] args) {
  int x = 0x16;              //00000000000000000000000000010110
  System.out.println(x << 3);//00000000000000000000000010110000

  int y = 0xfe;              //00000000000000000000000011111110
  y >>= 4;                   //00000000000000000000000000001111
  System.out.println(y);     //00000000000000000000000000001111

  x = 9;                     //00000000000000000000000000001001
  System.out.println(x >> 3);//00000000000000000000000000000001
  System.out.println(x >>>3);//00000000000000000000000000000001

  x = -9;                    //11111111111111111111111111110111
  System.out.println(x >> 3);//11111111111111111111111111111110
  System.out.println(x >>>3);//00011111111111111111111111111110
 }
}
```

9.4 Logical operators

Java provides the logical operators *and* (&&), *or* (||) and *not* (!). The logical operators can only be applied to boolean expressions and return a boolean value.

The following example shows the value returned by some logical expressions:

```
/**
 * Logical operators Application
 */

class Logical {

  public static void main(String[] args) {
    int x = 12,y = 33;
    double d = 2.45,e = 4.54;

    System.out.println(x < y && d < e);
    System.out.println(!(x < y));

    boolean test = 'a' > 'z';
    System.out.println(test || d - 2.1 > 0);
  }
}
```

The output produced by the execution of the application is:

```
true
false
true
```

9.5 String operators

Java provides a complete set of operators on Strings. We will leave most of them for a later section, and we will now consider just the concatenation operator (+). This operator combines two strings and produces a new string with characters from both arguments.

A useful behavior happens when an expression begins with a `String` and uses the + operator. In this case, the next argument is converted to `String` if necessary.

The next program shows some examples:

```
/**
 * Strings operators Application
 */

class Strings {
```

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

    String s1 = "Hello " + "World!";
    System.out.println(s1);

    int i = 35,j = 44;
    System.out.println("The value of i is "  + i +
                       " and the value of j is " + j);
  }
}
```

The output produced by the execution of the application is:

```
Hello World!
The value of i is 35 and the value of j is 44
```

Due to the fact that the expression between parenthesis starts with a `String` and the operator `+` is used, the values of `i` and `j` are converted into strings, and then concatenated:

```
System.out.println("The value of i is "  + i +
                   " and the value of j is " + j);
```

9.6 Casting

Java performs an automatic type conversion of the values when there is no risk of data loss. This is the usual case for *widening* conversions, as the following example shows:

```
/**
 * Test Widening conversions Application
 */

class TestWide {

  public static void main(String[] args) {
    int a = 'x';        // 'x' is a character
    long b = 34;        // 34 is an int
    float c = 1002;     // 1002 is an int
    double d = 3.45F;   // 3.45F is a float
  }
}
```

In order to specify conversions where data can be lost (*narrowing* conversions) it is necessary to use the cast operator. It consists of just the name of the type we want to convert to, between parenthesis, as the following example shows:

```
/**
 * Test Narrowing conversions Application
 */

class TestNarrow {

  public static void main(String[] args) {
    long a = 34;
    int b = (int)a;       // a is long, narrowing conversion
    double d = 3.45;
    float f = (float)d;   // d is double, narrowing conversion
  }
}
```

These conversions must only be used when we are certain that no data would be lost.

10 Control structures

Programs are built using the three fundamental blocks:

- sequence.
- selection.
- repetition and iteration.

Java provides the same set of control structures as C, with the main difference being that the conditional expression must be a *boolean* value, and cannot be an integer. Next sections describe them.

10.1 Selection control statements

The basic selection mechanism is the statement `if`, which decides, based on the value of a boolean expression, the statement that has to be executed. It has two forms:

```
if (boolean-expression)
  statement
```

and:

```
if (boolean-expression)
  statement
else
  statement
```

A *statement* can be replaced by one instruction or by a compound statement consisting of a set of instructions surrounded by curly braces.

The following application presents an example of the use of the `if` selection. The application prints the words “letter”, “digit” or “other character” depending on the value of the variable `c`:

```
/**
 * If control statement Application
 */

class If {

  public static void main(String[] args) {

    char c = 'x';

    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
      System.out.println("letter: " + c);
    else if (c >= '0' && c <= '9')
      System.out.println("digit: " + c);
    else {
      System.out.println("other character:");
      System.out.println("the character is: " + c);
      System.out.println("it is not a letter");
      System.out.println("and it is not a digit");
    }
  }
}
```

The output produced by the execution of the application is:

```
letter: x
```

10.2 Repetition control statements

Java provides the standard `while` and `do-while` repetition control statements to implement indeterminate loops. They allow the repetition of a statement (or compound statement) while a *boolean* expression evaluates to `true`. Their forms are:

```
while (boolean-expression)
  statement
```

and:

```
do
  statement
while (boolean-expression);
```

Note that the `while` structure can execute the statement zero or more times, and the `do-while` structure can execute the statement one or more times, depending on the value of the boolean expression.

The following example prints the number of times it is necessary to increment a variable in a certain step from an initial value till it goes over a limit:

```
/**
 * While control statement Application
 */

class While {

  public static void main(String[] args) {
    final float initialValue = 2.34F;
    final float step = 0.11F;
    final float limit = 4.69F;

    float var = initialValue;
    int counter = 0;

    while (var < limit) {
      var += step;
      counter++;
    }
    System.out.println("It is necessary to increment it "
                       + counter + " times");
  }
}
```

The output produced by the execution of the application is:

```
It is necessary to increment it 22 times
```

Java provides also a control structure that supports iteration: the `for` loop. Its form is:

```
for(initialization;boolean-expression;step)
  statement;
```

The initialization expression is executed first, and then the statement is executed while the boolean expression evaluates to true. Before the evaluation of the boolean expression, the step expression is evaluated.

The next example performs the same computations as the previous one, but using the `for` loop:

```
/**
 * For control statement Application
 */

class For {

  public static void main(String[] args) {
    final float initialValue = 2.34F;
    final float step = 0.11F;
    final float limit = 4.69F;

    int counter = 0;

    for (float var = initialValue;var < limit;var += step)
      counter++;

    System.out.println("It is necessary to increment it "
                       + counter + " times");
  }
}
```

The scope of the variable `var` defined in the first expression of the `for` loop is the body of the loop, the boolean expression and the step expression. The output of the application is, of course, the same as the output of the previous example.

10.3 break and continue

The statements `break` and `continue` provide control inside loops. `break` quits the loop whereas `continue` starts a new iteration by evaluating the boolean expression again. In the case of the `for` loop, the step expression is executed first.

The next example illustrates the use of `break` and `continue`:

```
/**
 * Break and Continue control statement Application
 */

class BreakContinue {

  public static void main(String[] args) {
    int counter = 0;

    for (counter = 0;counter < 10;counter++) {

      // start a new iteration if the counter is odd
      if (counter % 2 == 1) continue;
```

```
      // abandon the loop if the counter is equal to 8
      if (counter == 8) break;

      // print the value
      System.out.println(counter);
    }
    System.out.println("done.");
  }
}
```

The output produced by the execution of the application is:

```
0
2
4
6
done.
```

Note that the boolean expression of the first `if` statement evaluates to `true` when the value of the counter is odd. In this case, the `continue` statement finishes the current execution of the body of the loop, executing the step expression (`counter++`), and evaluating again the boolean expression (`counter < 10`). If this expression evaluates to `true`, the body of the loop is executed again.

The `break` statement breaks the execution out of the loop when the counter reaches the value 8, causing the control to be transferred to the last statement of the program.

Although `break` and `continue` can be used with a label, we will not cover such usage in these lecture notes.

10.4 Switch control statement

The `switch` control structure selects blocks of code to be executed based on the value of an integral expression. Its structure is as follows:

```
switch (integral-expression) {

  case integral-value: statement; [break;]
  ...
  case integral-value: statement; [break;]

  [default: statement;]
}
```

The square brackets surround optional statements. The integral expression is evaluated and the statement that has a value that matches the value of the

expression is executed. If the `break` statement is present, the `switch` statement is abandoned. if not, the statements that follow are executed independently of their integral value, till a `break` statement is found, or the end of the `switch` body is reached.

The integral expression can be any expression that returns a value convertible to `int`. This means that it can be `char`, `short`, `byte` or `int`.

The following example counts the number of days in a year. Note that the answer would be completely different if `break` statements are removed.

```
/**
 * Switch control statement Application
 */

class Switch {

  public static void main(String[] args) {
    boolean leapYear = true;
    int days = 0;

    for(int month = 1;month <= 12;month++) {

      switch(month) {
        case 1:          // months with 31 days
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:

        case 12:
          days += 31;
          break;

        case 2:          // February is a special case
          if (leapYear)
            days += 29;
          else
            days += 28;
          break;

        default:         // it must be a month with 30 days
          days += 30;
          break;
      }
    }
```

```
    System.out.println("number of days: " + days);
  }
}
```

The output produced by the execution of the application is:

```
number of days: 366
```

11 Arrays

In *Java* it is possible to declare arrays that can be used to store a number of elements of the same type. The following are some examples of declarations of arrays:

```
int[] a;        // an unitialized array of integers
float[] b;      // an unitialized array of floats
String[] c;     // an unitialized array of Strings
```

The declaration does not specify a size for the array. In fact, the declaration does not even allocate space for them.

The size can be specified by initializing the arrays in the declaration:

```
int[] a = {13,56,2034,4,55};            // size: 5
float[] b = {1.23F,2.1F};               // size: 2
String[] c = {"Java","is","great"};     // size: 3
```

Other possibility to allocate space for arrays is using the operator `new`. In this case the size of the array can be computed even at execution time.

```
int i = 3,j = 5;
double[] d;           // unitialized array of doubles

d = new double[i+j];  // array of 8 doubles
```

When the `new` operator is used, the memory is assigned dynamically. The components of the array are initialized with default values: 0 for numeric type elements, `'\0'` for characters and `null` for references (more about that later).

The array can be accessed by using an integer index that can take values from 0 to the size of the array minus 1. For example, it is possible to modify the third element (the one with index 2) of array `a` in the first example with the following assignment:

```
a[2] = 1000;    // modify the third element of a
```

Every array has a member called `length` that can be used to obtain the length of the arrays. The next application shows examples of the use of arrays:

```
/**
 * Arrays Application
 */

class Arrays {

  public static void main(String[] args) {
    int[] a = {2,4,3,1};

    // compute the summation of the elements
    int sum = 0;
    for(int i = 0;i < a.length;i++)
      sum += a[i];

    // create an array of floats with this size
    float[] d = new float[sum];

    // assign some values
    for(int i = 0;i < d.length;i++)
      d[i] = 1.0F / (i + 1);

    // print the values in odd positions
    for(int i = 1;i < d.length;i += 2)
      System.out.println("d[" + i + "]=" + d[i]);

  }
}
```

The output produced by the execution of the application is:

```
d[1]=0.5
d[3]=0.25
d[5]=0.16666667
d[7]=0.125
d[9]=0.1
```

It is also possible to declare multidimensional arrays with a similar approach. As an example, the following line declares a matrix of integers that can be used to store 50 elements, organized in 10 rows of 5 columns.

```
int[][] a = new int[10][5];
```

12 Command line arguments

We have seen that the method `main` has to be defined as follows:

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

It takes one argument that is defined as an array of strings. Through this array, the program can access the command line arguments typed when the program is submitted to the java virtual machine for execution. The following application prints all of its command line arguments:

```
/**
 * Command Line Arguments Application
 */

class CommandArguments {

  public static void main(String[] args) {

    for(int i = 0;i < args.length;i++)
      System.out.println(args[i]);
  }
}
```

Sample executions of the application follows:

```
# java CommandArguments Hello World
Hello
World
# java CommandArguments
# java CommandArguments I have 25 cents
I
have
25
cents
```

Even if, in the last example, the argument 25 is an integer, it is considered as the string "25", which is stored in `args[2]`. It is possible to convert a string that contains a valid integer into an `int` value by using method `parseInt` that belongs to the class `Integer` (more details on that later).

The following application accepts two arguments in the command line. They must be integers. The application prints the result of the addition of the two arguments.

```
/**
 * Add Application
 */

class Add {

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

```
    if (args.length != 2) {
      System.out.println("Error");
      System.exit(0);
    }
    int arg1 = Integer.parseInt(args[0]);
    int arg2 = Integer.parseInt(args[1]);

    System.out.println(arg1 + arg2);
  }
}
```

Sample executions of the application follows:

```
# java Add 2 4
6
# java Add 4
Error
# java Add 33 22
55
```

Note the use of the method `exit` that belongs to class `System`, which is used to terminate the execution of the application. Note also that the conversion from string type to integer type is performed by using the method `parseInt` as discussed below.

13 Classes

A class is defined in *Java* by using the `class` keyword and specifying a name for it. For example, the code:

```
class Book {

}
```

declares a class called `Book`. New instances (or objects) of the class can be created (or instantiated) using the keyword `new`, as follows:

```
Book b1 = new Book();
Book b2 = new Book();
```

or in two steps, with exactly the same meaning:

```
Book b3;

b3 = new Book();
```

As you can imagine, this class is not very useful since it has an empty body (ie. it contains nothing).

Inside a class it is possible to define data members, usually called *fields*, and member functions, usually called *methods*. The fields are used to store information and the methods are used to communicate with the instances of the classes.

Let's suppose we want to use instances of the class `Book` to store information on the books we have, and particularly, we are interested in storing the title, the author and the number of pages of each book. We can then add three fields to the Book class as follows:

```
class Book {
  String title;
  String author;
  int numberOfPages;
}
```

Note that the above definition is a kind of template or blueprint that defines the class `Book`.

Now, each instance of this class will contain three fields. The fields can be accessed with the dot notation, which consists of the use of a dot (.) between the name of the instance and the name of the field we want to access.

The next application shows how to create an instance and how to access these fields:

```
/**
 * Example with books Application
 */

class Book {
  String title;
  String author;
  int numberOfPages;
}

class ExampleBooks {

  public static void main(String[] args) {
    Book b;

    b = new Book();                // default constructor
    b.title = "Thinking in Java";
    b.author = "Bruce Eckel";
    b.numberOfPages = 1129;

    System.out.println(b.title + " : " + b.author +
```

```
                        " : " + b.numberOfPages);
  }
}
```

The output produced by the execution of the application is:

```
Thinking in Java : Bruce Eckel : 1129
```

13.1 Constructors

The constructors allow the creation of instances that are properly initialized. A constructor is a method that has the same name as the name of the class to which it belongs, and has no specification for the return value, since it returns nothing.

The next application provides a constructor called `Book` (there is no other option) that initializes all fields of an instance of `Book` with the values passed as arguments:

```
/**
 * Example with books Application (version 2)
 * that shows the use of constructors
 */

class Book {
  String title;
  String author;
  int numberOfPages;

  Book(String tit,String aut,int num) {      // constructor
    title = tit;
    author = aut;
    numberOfPages = num;
  }
}

class ExampleBooks2 {

  public static void main(String[] args) {
    Book b;

    //  create an instance of a book

    b = new Book("Thinking in Java","Bruce Eckel",1129);


    System.out.println(b.title + " : " + b.author +
```

```
                        " : " + b.numberOfPages);
  }
}
```

The constructor is called when the instance of a book is created. The output produced by the execution of this application is:

```
Thinking in Java : Bruce Eckel : 1129
```

Java provides a default constructor for the classes. This is the one that was called in the example `ExampleBooks` before, without arguments:

```
b = new Book();
```

This default constructor is only available when no other constructors are defined in the class. This means, that in the last example `ExampleBooks2` it is not possible to create instances of books by using the default constructor.

It is possible to define more than one constructor for a single class, only if they have different number of arguments or different types for the arguments. In this way, the compiler is able to identify which constructor is called when instances are created.

The next application adds one extra field for books; the ISBN number. The previously defined constructor is modified in order to assign a proper value to this field. A new constructor is added in order to initialize all the fields with supplied values. Note that there is no problem in identifying which constructor is called when instances are created:

```
/**
 * Example with books Application (version 3)
 * that defines more that one constructor
 */

class Book {
  String title;
  String author;
  int numberOfPages;
  String ISBN;

  Book(String tit,String aut,int num) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = "unknown";
  }

  Book(String tit,String aut,int num,String isbn) {
    title = tit;
```

```
    author = aut;
    numberOfPages = num;
    ISBN = isbn;
  }
}

class ExampleBooks3 {

  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    System.out.println(b1.title + " : " + b1.author +
        " : " + b1.numberOfPages + " : " + b1.ISBN);


    b2 = new Book("Thinking in Java","Bruce Eckel",1129,
        "0-13-027363-5");
    System.out.println(b2.title + " : " + b2.author +
        " : " + b2.numberOfPages + " : " + b2.ISBN);
  }
}
```

The output of the execution of the application is:

```
Thinking in Java : Bruce Eckel : 1129 : unknown
Thinking in Java : Bruce Eckel : 1129 : 0-13-027362-5
```

13.2 Methods

A *method* is used to implement the messages that an instance (or a class) can receive. It is implemented as a function, specifying arguments and the type of return value. It is called by using the dot notation.

The following is the same application as the one defined before, but with a method to get the initials of the author's name from an instance of a `Book`:

```
/**
 * Example with books Application (version 4)
 * that defines one method
 */

class Book {
  String title;
  String author;
  int numberOfPages;
  String ISBN;
```

```
  Book(String tit,String aut,int num) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = "unknown";
  }

  Book(String tit,String aut,int num,String isbn) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = isbn;
  }


  public String getInitials() {
    String initials = "";

    for(int i = 0;i < author.length();i ++) {
      char currentChar = author.charAt(i);
      if (currentChar >= 'A' && currentChar <='Z') {
        initials = initials + currentChar + '.';
      }
    }
    return initials;
  }
}

class ExampleBooks4 {

  public static void main(String[] args) {
    Book b;

    b = new Book("Thinking in Java","Bruce Eckel",1129);
    System.out.println("Initials: " + b.getInitials());
  }
}
```

The output of the execution of the application is:

```
Initials: B.E.
```

The prototype of the method `getInitials()` is:

```
public String getInitials()
```

The method is defined `public` so it can be called from other classes (more details on that later). It takes no arguments and returns a String.

It is called by using the dot notation:

```
System.out.println("Initials: " + b.getInitials());
```

Note that no arguments are passed. In object oriented terminology, we must say that the message "getInitials" is sent to the object "b". The object "b" is the receptor of the message.

The method is implemented as follows:

```
  public String getInitials() {
    String initials = "";


    for(int i = 0;i < author.length();i ++) {
      char currentChar = author.charAt(i);
      if (currentChar >= 'A' && currentChar <='Z') {
        initials = initials + currentChar + '.';
      }
    }
    return initials;
  }
```

All references to `author` correspond to references to the field called `author` in the receptor of the message, in this case, the instance `b`.

The method creates an empty string in the variable `initials`, and traverses the field `author` searching for uppercase letters. If an uppercase letter is found, it is added to the string `initials` together with a dot.

Note the use of the methods `length()` and `charAt(int)` of class String, that can be used to get the length of a string, and the character in a specified position in the string.

The next example defines an array of books, and initializes it with data pertaining to three books. After that, the method `getInitials` is called on the three instances. This should clarify the fact that even if the method `getInitials` processes data stored in `author`, it corresponds to the specific field of the receptor of the message.

```
class ExampleBooks5 {

  public static void main(String[] args) {
    Book[] a;

    a = new Book[3];
    a[0] = new Book("Thinking in Java","Bruce Eckel",1129);
```

```
    a[1] = new Book("Java in a nutshell","David Flanagan",353);
    a[2] = new Book("Java network programming",
                    "Elliotte Rusty Harold",649);

    for(int i = 0;i < a.length;i++)
      System.out.println("Initials: " + a[i].getInitials());
  }
}
```

The output of the execution of the application is:

```
Initials: B.E.
Initials: D.F.
Initials: E.R.H.
```

13.3 Equality and equivalence

The usual operator for testing equality (==) can be a bit confusing when it is used to compare objects. The next application defines two books with the same values and then compares them:

```
class ExampleBooks6 {

  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = new Book("Thinking in Java","Bruce Eckel",1129);

    if (b1 == b2)
      System.out.println("The two books are the same");
    else
      System.out.println("The two books are different");
  }
}
```

The output of the execution of the application is:

```
The two books are different
```

The fact is that the equivalent operator (==) checks for the equivalence of the objects, ie. if the two objects passed as arguments are the same object, but not if they have the same values. Things are different if we write the application as follows:

```
class ExampleBooks6a {

  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = b1;

    if (b1 == b2)
      System.out.println("The two books are the same");
    else
      System.out.println("The two books are different");
  }
}
```

The output of the execution of the application is:

```
The two books are the same
```

Now, `b1` and `b2` are **references** to the same object. The expression `b1 == b2` returns `true`, because both variables refer to the same object.

In order to have the possibility to test for equality in the sense that two objects are equal if both have the same values, it is necessary to define a method. It is usual practice to call it `equals`. It can be defined inside the class `Book` as follows:

```
  public boolean equals(Book b) {
    return (title.equals(b.title) && author.equals(b.author) &&
            numberOfPages == b.numberOfPages &&
            ISBN.equals(b.ISBN));
  }
```

The method `equals` receives one reference to `Book` as an argument and returns a `boolean` value. This value is computed as the result of an expression that compares each field individually.

The next application tests equality of books:

```
class ExampleBooks7 {

  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = new Book("Thinking in Java","Bruce Eckel",1129);

    if (b1.equals(b2))
      System.out.println("The two books are the same");
    else
```

```
      System.out.println("The two books are different");
  }
}
```

The output of the execution of the application is:

```
The two books are the same
```

13.4 Static fields

In Object Oriented programming classes are used as models to create instances. Every instance of a `Book` has four fields (`title`, `author`, `numberOfPages` and `ISBN`), and they can be used to store values in one instance independently of the values stored in other instances. Hence they are called instance variables.

Static fields (or class variables) are data members that belong to the class and do not exist in each instance. It means that there is always only one copy of this data member, independent of the number of the instances that were created.

The next example defines a static field called `owner` that will be used to store the name of the owner of the books. We assume that all the books that we are going to define in an application will belong to the same person. In this case, it is not necessary to have one field in each instance (book) to store the name, because it must be the same in all of them. The example also defines two methods: `setOwner` and `getOwner` that will be used to set and get the owner of all books respectively. This methods are usually called *accessor* methods.

```
/**
 * Example with books Application (version 8)
 * that defines a static field and accessor methods
 */

class Book {
  String title;
  String author;
  int numberOfPages;
  String ISBN;
  static String owner;  // shared by all instances

  Book(String tit,String aut,int num) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = "unknown";
  }

  Book(String tit,String aut,int num,String isbn) {
    title = tit;
```

```
    author = aut;
    numberOfPages = num;
    ISBN = isbn;
  }

  public String getInitials() {
    String initials = "";

    for(int i = 0;i < author.length();i ++) {
      char currentChar = author.charAt(i);
      if (currentChar >= 'A' && currentChar <='Z') {
        initials = initials + currentChar + '.';
      }
    }
    return initials;
  }

  public boolean equals(Book b) {
    return (title.equals(b.title) && author.equals(b.author) &&
            numberOfPages == b.numberOfPages &&
            ISBN.equals(b.ISBN));
  }

  public void setOwner(String name) {
    owner = name;
  }

  public String getOwner() {
    return owner;
  }
}

class ExampleBooks8 {

  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = new Book("Java in a nutshell","David Flanagan",353);
    b1.setOwner("Carlos Kavka");

    System.out.println("Owner of book b1: " + b1.getOwner());
    System.out.println("Owner of book b2: " + b2.getOwner());
  }
}
```

The main application creates two books, and then set the owner by sending the message `setOwner` to the object `b1` (it could also be `b2`). After that it prints the owner of both books. The output of the execution of the application is:

```
Carlos Kavka
Carlos Kavka
```

It can be seen that even if the owner was set by sending a message to the object `b1`, the owner of `b2` was modified. In fact, there is only one field called `owner` that can be accessed with the methods through all instances of `Books`.

Static fields can be used for communication between different instances of the same class, or to store a global value at the class level.

13.5 Static methods

With the same idea of the static fields, it is possible to define class methods or static methods. These methods do not work directly with instances but with the class. As an example, we want to define a method called `description` to provide information about the class `Book`. In this sense, the information returned by this method must be the same, independent of the instance. The method can be defined inside the class `Book` in this way:

```
  public static String description() {
    return "Book instances can store information on books";
  }
```

Note the word static before the specification of the return value. The application can then call the method as follows:

```
class ExampleBooks9 {

  public static void main(String[] args) {

    Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);

    System.out.println(b1.description());    // to the object
    System.out.println(Book.description());  // to the class
  }
}
```

The output of the execution of the application is:

```
Book instances can store information on books
Book instances can store information on books
```

A static method can be called by sending the message to the class, or by sending the message to any instance.

Static methods can only access static variables. In the above example, the only variable that can be accessed from the method `description` is the variable `owner`.

13.6 A static application

All the examples we have seen till now define a class that contains a static method called main, where usually instances from other classes are created.

It is possible to define a class with only static methods and static fields, as the following example shows:

```
/**
 * All static class Application
 */

import java.io.*;

class AllStatic {
  static int x;
  static String s;

  public static String asString(int aNumber) {
    return "" + aNumber;
  }

  public static void main(String[] args) {

    x = 165;
    s = asString(x);
    System.out.println(s);
  }
}
```

This application defines two static fields `x` and `s`. It also contains two static methods `asString` and `main`.

The method `main` calls the method `asString`. This can be done since both of them are static, and they operate only on static fields. There is no need to create an instance of this class in order to send the messages.

In some sense, when only static fields and methods are defined, the class looks like a standard C program, with functions and global data.

It is interesting to note the way in which the function `asString` converts an integer value to a string. It uses the operator + and the property that when the first argument is a string, the next one is converted to string.

13.7 Fields initialization

All fields in an object are guaranteed to have an initial value. There exists a default value for each primitive type as illustrated in Table 2:

type	default value
byte	0
short	0
int	0
long	0
float	0.0F
double	0.0
char	'\0'
boolean	false

Table 2: Initial values of primitive data types

All references to objects get an initial value of `null`. The following application shows an example:

```
/**
 * InitialValues Application
 */

class Values {
  int x;
  float f;
  String s;
  Book b;
}

class InitialValues {

  public static void main(String[] args) {

    Values v = new Values();

    System.out.println(v.x);
    System.out.println(v.f);
    System.out.println(v.s);
    System.out.println(v.b);
  }
}
```

The output of the execution of the application is:

```
0
0.0
null
null
```

The values can be initialized also in the constructor, or even by calling methods at the declaration point, as the following example shows:

```
/**
 * InitialValues Application (version 2)
 */

class Values {
  int x = 2;
  float f = inverse(x);
  String s;
  Book b;

  Values(String str) {
    s = str;
  }

  public float inverse(int value) {
    return 1.0F / value;
  }
}

class InitialValues2 {

  public static void main(String[] args) {

    Values v = new Values("hello");

    System.out.println(v.x);
    System.out.println(v.f);
    System.out.println(v.s);
    System.out.println(v.b);
  }
}
```

The output of the execution of the application is:

```
2
0.5
hello
null
```

14 The keyword “this”

The keyword `this`, when used inside a method, refers to the receiver object. It has two main uses: 1). it can be used to return a reference to the receiver

object from a method and 2). it can be used to call constructors from other constructors.

For example, the method `setOwner` in the previous `Book` class could have been defined as follows:

```
public Book setOwner(String name) {
  owner = name;
  return this;
}
```

The method returns a reference to `Book`, and the value returned is a reference to the receiver object. With this definition of the method, it can be used as follows:

```
Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
System.out.println(b1.setOwner("Carlos Kavka").getInitials());
System.out.println(b1.getOwner());
```

The message `setOwner` is sent to `b1`. The method returns a reference to the receptor object, which is `b1`. Then the message `getInitials` is sent to `b1`.

The output of the execution of this example is:

```
B.E.
Carlos Kavka
```

The other use of `this` is to call from one constructor another constructor. For example, in the definition of the class `Book` there were two constructors:

```
Book(String tit,String aut,int num) {
  title = tit;
  author = aut;
  numberOfPages = num;
  ISBN = "unknown";
}

Book(String tit,String aut,int num,String isbn) {
  title = tit;
  author = aut;
  numberOfPages = num;
  ISBN = isbn;
}
```

The second one can be defined in a shorter way by calling the first constructor. This can be done as follows:

```
  Book(String tit,String aut,int num,String isbn) {
    this(tit,aut,num);
    ISBN = isbn;
  }
```

The effect is exactly the same. The first constructor is called, and then the value in `isbn` is assigned to the data member `ISBN`.

When the keyword `this` is used in this way, it must be called as the first action of the constructor.

The complete implementation of the `Book` class, with even more methods is included in Appendix A.

15 An example: the complex number class

Let's suppose that we want to define a complex number class that allows us to work with complex numbers in our applications. The following application shows an example of the use of this `Complex` class, that we want to define:

```
/**
 * Test Complex class Application
 */

class TestComplex {

  public static void main(String[] args) {

    Complex a = new Complex(1.33,4.64);
    Complex b = new Complex(3.18,2.74);

    Complex c = a.add(b);
    System.out.println("a+b = " + c.getReal() + " " +
                       c.getImaginary());

    Complex d = c.sub(a);
    System.out.println("c+d = " + d.getReal() + " " +
                       d.getImaginary());

  }
}
```

This application creates two complex numbers `a` and `b` with some initial values in their real and imaginary parts. A complex number `c` is created as the addition of `a` and `b`, and then its real and imaginary parts are printed. After that, a complex number `d` is created as the subtraction of `a` from `c`. This number is also printed.

The output of the execution of this application should be something like:

```
a+b = 4.51 7.38
c+d = 3.18 2.74
```

The `Complex` class should have two data members to store the real and the imaginary parts of the complex numbers. We have to define a constructor that can initialize both parts from the arguments, and methods to get the real and imaginary parts of the number. This can be done as follows:

```
/**
 * Complex Number class
 */

public class Complex {
  double real;         // real part
  double im;           // imaginary part

  /** This constructor creates a complex number from its real
   *  and imaginary part.
   */

  Complex(double r,double i) {
    real = r;
    im = i;
  }

  /** This method returns the real part
   */

  public double getReal() {
    return real;
  }

  /** This method returns the imaginary part
   */

  public double getImaginary() {
    return im;
  }
}
```

We have to define two specific methods in order to implement the addition and subtraction of complex numbers. From the example, we can see that both methods must take one argument; the complex number to be added to or subtracted from the number that is the receptor of the message. For example, in the following expression, the method `sub` must subtract from the complex number `c` the complex number `a`:

```
Complex d = c.sub(a);
```

Note that the methods have to create a new complex number and return it as the result. They do not have to modify the receptor or the complex number passed as argument. This can be implemented as follows:

```
/** This method returns a new complex number which is
 *  the result of the addition of the receptor and the
 *  complex number passed as argument
 */

public Complex add(Complex c) {
  return new Complex(real + c.real,im + c.im);
}

/** This method returns a new complex number wich is
 *  the result of the substraction of the receptor and the
 *  complex number passed as argument
 */

public Complex sub(Complex c) {
  return new Complex(real - c.real,im - c.im);
}
```

Note that we use `new` in order to create a new instance of a complex number. It is initialized by calling the constructor, and then it is returned.

Let's suppose we want to define a method `addReal` that increments just the real part of the receptor of the message with the value passed as argument. Note that this method must modify the receptor, something that methods `add` and `sub` were not doing. An example of its use could be:

```
a.addReal(2.0);
```

By considering our previous example, we should get the values 3.33 and 4.64 as the real and imaginary parts of `a` after the execution of the method. Imagine that we would like to be able to use it also in the following way:

```
a.addReal(2.0).addReal(3.23);
```

In this case we want to add first 2.0 to the real part of `a`, and then 3.23. In this case, we need that the method `addReal` returns a reference to the receptor object (or current object), so the next call to `addReal` can operate on the same complex number.

As `this` is a reference to the receptor object when it is used in a method, this can be done as follows:

```
  /** This method increments the real part by a value
   *  passed as argument. Note that the method modifies
   *  the receptor
   */

  public Complex addReal(double c) {
    real += c;
    return this;    // returns the receptor object
  }
```

We must be careful if we want to create one complex number as a copy of the other, since the assignment expression shown below will not do it:

```
Complex e = a;
```

This will make just `e` to be a reference to the same object referenced by `a` (see section 13.3). This means that if we increment `e`, then `a` will be incremented too.

In order to create a new complex number, we should use a constructor as follows:

```
Complex e = new Complex(a);
```

It is necessary then to define a constructor that takes one complex number as argument. An interesting way to define it is as follows:

```
  /** This constructor creates a complex number as a copy
   *  of the complex number passed as argument
   */

  Complex(Complex c) {
      this(c.real,c.im);
  }
```

Note that this constructor takes a complex number as argument, and calls (through `this`) the constructor defined previously.

The complete implementation of the `Complex` class, with even more methods is included in Appendix B.

16 Inheritance

Inheritance allows to define new classes by reusing other classes. It is possible to define a new class (called subclass) by saying that the class must be “like” the other class (called super class) by using the keyword `extends` followed by the

name of the super class. The definition of the new class specifies the differences with the super class.

Let's suppose we want to extend the definition of the class `Book` we have defined in Section 13 to store information on scientific books. We can add two fields to the definition of the class `Book` in order to store the area of science they cover and a boolean data member to identify proceedings from normal scientific books:

```
class ScientificBook extends Book {
  String area;
  boolean proceeding = false;
}
```

The instances of a `ScientificBook` will have six fields altogether: `title`, `author`, `numberOfPages`, `ISBN`, `area` and `proceeding`; the four inherited from the super class `Book` and the two newly defined fields. Note that by default a scientific book is not a proceeding.

16.1 Constructors

We can define a constructor for the class as follows:

```
  ScientificBook(String tit,String aut,int num,String isbn,
                 String a) {
    super(tit,aut,num,isbn);   // use the constructor defined
    area = a;                  // in the super class
  }
```

The constructor defined above has the same parameters as the constructor of the class `Book` plus one parameter for the area. As there is one constructor that can be used to initialize the first four data member in the super class `Book`, it is not necessary to do it again here. The constructor of the super class can be called through `super`.

By using this constructor, a scientific book can be defined as follows:

```
ScientificBook sb;

sb = new ScientificBook("Neural Networks, A Comprehensive
         Foundation","Simon Haykin",696,"0-02-352761-7",
         "Artificial Intelligence");
```

The method `super` must be the first instruction in the body of the constructor. If it is not used, then the *Java* compiler inserts a call to `super` without parameters. If there is no such constructor, the compiler will indicate an error.

By extending classes it is possible to define a complete hierarchy of classes. Every class can inherit from the class above and can add some fields and methods.

16.2 Methods

A class can have two or more methods of the same name but with different sets of parameters. *Java* is able to distinguish between them by the number and type of their parameters. This property is known as *Method Overloading*.

New methods can be defined in the subclass to specify the behavior of the objects of this class. However, methods defined above in this hierarchy can also be called.

This new method in the subclass can have the same name and the parameters as a method in the super class. This property is known as *Method Overriding*.

When a message is sent to an object, the method is searched for in the class of the receptor object. If it is not found then it is searched for higher up in the hierarchy of classes till it is found.

The inheritance can then be used to reuse the code defined in other related classes. In some cases, the behavior of a method has to be changed. In this case, the method can be redefined. As the search of a method starts from the receptor class, the most specific method is always selected.

In our example, we can certainly reuse the method `getInitials` from the class `ScientificBook`, since it works over the data member `author`, which is common to instances from both classes.

Without defining it for scientific books, we can do something like:

```
System.out.println(sb.getInitials());
```

where `sb` is the instance of `ScientificBook` defined before.

We cannot use the method `equals` since in order to check if two scientific books are equal we have to consider now two more fields. However, we can reuse the checking of the four data members from the class `Book` and just write the comparison for the new fields as follows:

```
public boolean equals(ScientificBook b) {
  return super.equals(b) && area.equals(b.area ) &&
      proceeding == b.proceeding;
}
```

The method `equals` compares the fields `area` and `proceeding`. The comparison of the other four fields is done by calling the method `equals` defined in the super class by using `super`.

In this way, this method's `equals` redefines the method with the same name defined in the super class. However, the method `equals` defined in the super class is called as part of the definition of this method. When `super` is used to call super class methods, it can be used in any place of the body of the method.

It should be clear that the method could have been defined in the following way since all fields are accessible from this method:

```
public boolean equals(ScientificBook b) {
  return (title.equals(b.title) && author.equals(b.author) &&
```

```
            numberOfPages == b.numberOfPages &&
            ISBN.equals(b.ISBN) && area.equals(b.area) &&
            proceeding == b.proceeding;
  }
```

Of course, the previous version reuses code defined before.

It is not necessary to call redefined method from the subclass. For example, a method `description` can be defined to return a value independently of the value returned by the method with the same name in the super class. It can be defined as:

```
  public static String description() {
    return "ScientificBook instances can store information" +
           " on scientific books";
  }
```

New methods can also be defined. For example, we can define methods to set the field 'proceeding' and to check it, as follows:

```
  public void setProceeding() {
    proceeding = true;
  }

  public boolean isProceeding() {
    return proceeding;
  }
```

Note that it is possible to send a message `setProceeding` to an instance of the class `ScientificBook` but it is not possible to send it to an instance of `Book`.

The next application is an example of the use of scientific books:

```
/**
 * Test Scientific Book Class
 */

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

   ScientificBook sb1,sb2;

   sb1 = new ScientificBook("Neural Networks, A Comprehensive"+
              " Foundation","Simon Haykin",696,"0-02-352761-7",
              "Artificial Intelligence");
   sb2 = new ScientificBook("Neural Networks, A Comprehensive"+
              " Foundation","Simon Haykin",696,"0-02-352761-7",
              "Artificial Intelligence");
```

```
   sb2.setProceeding();

   System.out.println(sb1.getInitials());
   System.out.println(sb1.equals(sb2));
   System.out.println(sb2.description());
  }
}
```

The output of the execution of the application is:

```
S.H.
false
ScientificBook instances can store information on
scientific books
```

The complete `ScientificBook` class is provided in Appendix C.

16.3 Instanceof keyword and getClass method

The keyword `instanceof` returns a boolean value indicating if an object is an instance of a specified class. The method `getClass` returns the runtime class of the object, which can be printed as a string.

As an example, the following application shows an interesting result in the context of inheritance:

```
/**
 * Test Class Application
 */

class TestClass {

  public static void main(String[] args) {
   Book b1;
   ScientificBook sb1;

   b1 = new Book("Thinking in Java","Bruce Eckel",1129);

   sb1 = new ScientificBook("Neural Networks, A Comprehensive"+
             " Foundation","Simon Haykin",696,"0-02-352761-7",
             "Artificial Intelligence");

   System.out.println(b1.getClass());
   System.out.println(sb1.getClass());
   System.out.println(b1 instanceof Book);
   System.out.println(sb1 instanceof Book);
   System.out.println(b1 instanceof ScientificBook);
```

```
   System.out.println(sb1 instanceof ScientificBook);
  }
}
```

The output of the execution of the application is:

```
class Book
class ScientificBook
true
true
false
true
```

The two calls to `getClass` return the class the receptor objects are instances of, which are printed as strings. When calling `instanceof`, it is true that `b1` is an instance of `Book` and not an instance of a `ScientificBook`, and that `sb1` is an instance of `ScientificBook`.

It is interesting to note that `sb1` is also an instance of `Book`. In fact, every object is an instance of its class, and an instance of all classes that are higher up in the class hierarchy. This is what allows instances of scientific books to accept messages that correspond to methods defined for books.

17 Packages

A package is a structure in which classes can be organized. A package can contain any number of classes, usually related by purpose or by inheritance.

The standard classes in the system are organized into packages. For example, *Java* provides a class `Date` that can be used to work with dates in our classes. It is defined in the package `java.util`. In order to specify to the *Java* compiler that we are interested in the use of this class, we have to use the keyword `import` in a statement as follows:

```
import java.util.Date;
```

It is possible to specify just the name of the package, importing all classes defined in a given package:

```
import java.util.*;
```

An application that prints the current date is the following:

```
/**
 * Test Date Class
 */

import java.util.*;
```

```
class TestDate {

  public static void main(String[] args) {

    System.out.println(new Date());
  }
}
```

The output of the application (when I was executing it) was:

```
Wed Sep 15 11:40:16 ART 2004
```

New packages can be defined by using the keyword `package` with the name of the package we are going to define as argument:

```
package mypackage;  // first line of all source files
```

This must be the first non commented statement in the file. The classes defined in the file belong to the package `mypackage`. There can also be other files that define classes for the same package. They can be imported by other classes with the import statement.

However, not only the package name should be included in each file, the source files themselves should be placed in a special directory, that matches the package name structure. For example, if the file `Instrument.java` is defined as follows:

```
package it.infn.ts;

class Instrument {

...
}
```

It should be placed in the directory `it/infn/ts`. When executing an application that imports this class specifying the full package name, the virtual machine will look for the file by using this path, starting from the base directory or any directory specified in the so called *classpath*. The classpath lists all directories and files that are *starting points* for locating classes. Its value can be set by using the option `-classpath` for the compiler and the virtual machine, or more usually, by setting the `CLASSPATH` environment variable.

18 Access control

It is possible to control the access to methods and variables from other classes with three so called access modifiers: `public`, `private` and `protected`. There

exists a default access control which allows full access from all classes that belong to the same package. That is the one which we have been using in our examples so far.

Full access means that it is possible to access fields and methods from another class. For example, it is possible to set the proceeding condition of a scientific book from the class `TestScientificBook` as follows:

```
sb1.setProceeding();
```

or by just accessing the field:

```
sb1.proceeding = true;   / property exposed
```

Usually we do not want direct access to a field. This is in order to guarantee proper encapsulation. To achieve encapsulation we can use the modifier `private`. This modifier guarantees that the field can be accessed only from methods that belong to this class.

For example, the class `ScientificBook` can be defined in this way:

```
class ScientificBook extends Book {
  private String area;
  private boolean proceeding = false;
}
```

In this case, the direct access to the field `proceeding` is not allowed from other classes, and the condition of a scientific book to be a proceeding can only be asserted by sending the message `setProceeding`.

The same applies to methods. A private method can only be called from other methods in its own class.

Usually most of the data members are defined private, and they can only be modified by methods belonging to the class in which these data members are found. This is in fact the important property of the abstract data types (ADT) called encapsulation.

The `public` modifier allows full access from all other classes without restrictions. This is the usual way in which methods are defined so the messages they implement can be sent to objects of its class from all other classes.

The `protected` modifier allows access to fields and methods from subclasses and from all classes in the same package.

19 final and abstract

Two other modifiers can be used to define the methods and the classes: `final` and `abstract`.

A *final* method cannot be redefined in a subclass. It means that when a method is defined final, it is not possible for the subclasses to redefine its meaning.

A *final* class does not allow subclassing. It means that it is not possible to define subclasses of a final class.

An *abstract* method has no body, and it must be redefined in a subclass. It means that it is possible to define classes that force subclasses to define a specific method.

An *abstract* class is a class that cannot be instantiated. It means that it is not possible to define instances of this class. However, as subclassing is possible, instances can be created of subclasses of abstract classes.

We will now see an example that uses these concepts. Let's suppose we want to use in our application different types of input output boards. In particular, we have a serial board and an Ethernet network board. We have to define two classes, one for each type of board.

However, we can see that there are some data that are common to all input output boards: system name, counter for errors, etc. and some operations that are the same: initialization, reading, writing, close, etc.

A good design option is to define a class called `IOBoard` that contains data members and methods that are common to all types of input output boards. Then subclasses can be defined in order to implement the specific input output boards.

This `IOBoard` class must be abstract, in the sense, that we will not be creating instances of this general input output board, but will create instances of its subclasses.

It is important to note that it is not possible to define the real code for communication in the abstract class `IOBOard` since it is general, and we cannot assume a specific type of hardware. This requires that the implementation dependent code must be defined in the subclasses.

In order to force all subclasses to define methods for the required behavior of an input output board, these methods have to be defined as abstract methods.

The method used to increment the counter of errors in the abstract class `IOBoard` can be defined final, since no subclass must modify its behavior.

The following is the code of the class `IOBoard` defined in line with what is discussed above:

```
/**
 * IO board Class
 */

abstract class IOBoard {
  String name;              // common to all IOBoards
  int numErrors = 0;

  IOBoard(String s) {
    System.out.println("IOBoard constructor");
    name = s;
  }
```

```
  final public void anotherError() {
    numErrors++;
  }

  final public int getNumErrors() {
    return numErrors;
  }
  abstract public void initialize();
  abstract public void read();
  abstract public void write();
  abstract public void close();
}
```

A subclass of `IOBoard` cannot redefine the method `anotherError` since it was declared final. It is not possible to create an instance of `IOBoard` since it was declared abstract. This means that it is not possible to do something like:

```
IOBoard b = new IOBoard("my board");  // wrong !!!!
```

The subclass serial board can be defined as follows:

```
/**
 * IO serial board Class
 */

class IOSerialBoard extends IOBoard {
  int port;

  IOSerialBoard(String s,int p) {
    super(s);
    port = p;
    System.out.println("IOSerialBoard constructor");
  }

  public void initialize() {
    System.out.println("initialize method in IOSerialBoard");
    // specific code to initialize a serial board
  }

  public void read() {
    System.out.println("read method in IOSerialBoard");
    // specific code to read from a serial board
  }

  public void write() {
    System.out.println("write method in IOSerialBoard");
```

```
    // specific code to write to a serial board
  }

  public void close() {
    System.out.println("close method in IOSerialBoard");
    // specific code to close a serial board

  }
}
```

This class defines a constructor that takes the name of the board and a port as arguments. The port corresponds to a field defined in this class, and the name is the value to be stored in the field defined in the super class. Note that the constructor calls the constructor of the super class through `super`.

The respective methods will only print a message identifying themselves and return, since we are not going to define communication code in the method bodies in this example.

The subclass `IOEthernetBoard` can be defined as follows:

```
/**
 * IOEthernetBoard Class
 */


class IOEthernetBoard extends IOBoard {
  long networkAddress;

  IOEthernetBoard(String s,long netAdd) {
    super(s);
    networkAddress = netAdd;
    System.out.println("IOEthernetBoard constructor");
  }


  public void initialize() {
    System.out.println("initialize method in IOEthernetBoard");
    // specific code to initialize an ethernet board
  }

  public void read() {
    System.out.println("read method in IOEthernetBoard");
    // specific code to read from an ethernet board
  }

  public void write() {
```

```
    System.out.println("write method in IOEthernetBoard");
    // specific code to write to an ethernet board
  }

  public void close() {
    System.out.println("close method in IOEthernetBoard");
    // specific code to close an ethernet board

  }
}
```

This class defines a constructor that takes the name of the board and a network address as arguments. The network address corresponds to a field defined in this class, the name is the value to be stored in the field defined in the super class. Note that the constructor calls the constructor of the super class through `super`.

The next application presents an example of their use:

```
/**
 * Test Boards1 class Application
 */
class TestBoards1 {
  public static void main(String[] args) {

    IOSerialBoard serial = new IOSerialBoard("my first port",
                                             0x2f8);

    serial.initialize();
    serial.read();
    serial.close();
  }
}
```

The output of the execution of this application is:

```
IOBoard constructor
IOSerialBoard constructor
initialize method in IOSerialBoard
read method in IOSerialBoard
close method in IOSerialBoard
```

Note the order in which the constructors are executed. Note also that the methods defined in the subclass are executed, and the methods defined in the super class are ignored.

20 Polymorphism

Polymorphism is an important feature in which the appropriate method to act on an object is selected from all the methods that have the same name. We can say that there exists polymorphism when different objects can respond to the same kind of messages. In this way, we can operate with these objects by using the same interface.

In the last example, we can see that instances of the class `IOSerialBoard` and instances of the class `IOEthernetBoard` can respond to the same set of messages. We can say then that there exists polymorphism. This property allows to work with instances of both classes in the same way, as the next example shows:

```
/**
 * Test Boards2 class Application
 */

class TestBoards2 {

  public static void main(String[] args) {

    IOBoard[] board = new IOBoard[3];

    board[0] = new IOSerialBoard("my first port",0x2f8);
    board[1] = new IOEthernetBoard("my second port",0x3ef8dda8);
    board[2] = new IOEthernetBoard("my third port",0x3ef8dda9);

    for(int i = 0;i < 3;i++)
      board[i].initialize();

    for(int i = 0;i < 3;i++)
      board[i].read();

    for(int i = 0;i < 3;i++)
      board[i].close();
  }
}
```

In this application an array of three `IOBoard` instances is defined. A problem that seems to appear is the fact that it is not possible to define instances of this class, since it was declared abstract.

However, as we have seen before, instances of subclasses of `IOBoard` are also instances of `IOBoard` (see section 16.3). So, it is possible to make the assignments shown in the example, that assign one instance of a `Serial` board, and two instances of `IOEthernetBoard` boards to the array.

In order to work with the boards we have to initialize, read and close them. As the interface is the same, we can operate with the input output board instances

stored in the array by just sending the corresponding messages, without considering the specific type of board. This is possible due to polymorphism property.

21 Interfaces

In our last example we have defined an abstract class in order to define the common data and methods we want all input output boards to have and implement. However, we can get a similar behavior by using *interaces*.

An interface defines *what* a class should do, without specifying *how* it should be done. Its definition looks like a class definition, however, it is not a class, it is a specification of a set of requirements that the classes that wants to conform the interface should follow.

An interface looks like a class definition, where all fields are `static` and `final`, and all methods have no body and are `public`. No instances can be created from interfaces.

The fields can represent constant values (being final and static), and the methods can define a behavior, or more specifically as the name says, an interface.

The word `implements` can be used to define classes that implements an interface or in other words, classes that implement all methods defined in the interface.

In our example, if we are not interested in having a name for the boards and an error counter, we could have defined `IOboard` as an interface as follows:

```
/**
 * IO board interface
 */

interface IOBoardInterface {

  public void initialize();
  public void read();
  public void write();
  public void close();
}
```

The class `IOSerialBoard` can then be defined as a class that implements this interface, and not as a subclass of another class:

```
/**
 * IO serial board Class (second version)
 */

class IOSerialBoard2 implements IOBoardInterface {
  int port;
```

```
  IOSerialBoard2(int p) {
    port = p;
    System.out.println("IOSerialBoard constructor");
  }

  public void initialize() {
    System.out.println("initialize method in IOSerialBoard");
    // specific code to initialize a serial board
  }

  public void read() {
    System.out.println("read method in IOSerialBoard");
    // specific code to read from a serial board
  }

  public void write() {
    System.out.println("write method in IOSerialBoard");
    // specific code to write to a serial board
  }

  public void close() {
    System.out.println("close method in IOSerialBoard");
    // specific code to close a serial board

  }
}
```

The next application shows an example of the use of this class:

```
/**
 * Test Boards3 class Application
 */

class TestBoards3 {

  public static void main(String[] args) {

    IOSerialBoard2 serial = new IOSerialBoard2(0x2f8);

    serial.initialize();
    serial.read();
    serial.close();
  }
}
```

A class can implement more than one interface. For example, let's suppose we want to define an interface called `niceBehaviour` that defines methods we consider that all nice classes should implement:

```
/**
 * Nice behavior interface
 */

interface NiceBehavior {

  public String getName();
  public String getGreeting();
  public void sayGoodBye();
}
```

If we are interested in forcing the serial board class to implement all methods in `IOBoardInterface` and all methods in `NiceBehavior`, we can define the serial board class as follows:

```
/**
 * IO serial board Class (third version)
 */

class IOSerialBoard3 implements IOBoardInterface,
                                 NiceBehavior {
  ...
}
```

The *Java* compiler will accept this class definition, only if all methods defined in both interfaces are defined in this class.

We can see that a similar effect can be obtained with abstract classes as with interfaces. In some sense, both of them force other classes to define a specific behavior. However, a class can implement more than one interface, but a subclass cannot inherit from more than one class.

22 Exceptions

The usual behavior when there is a runtime error in an application is to abort the execution. For example:

```
/**
 * Test Exceptions class Application
 */

class TestExceptions1 {
```

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

    String s = "Hello";

    System.out.print(s.charAt(10));
  }
}
```

As the string s has no character in position 10, the execution stops with the following message:

```
Exception in thread "main"
    java.lang.StringIndexOutOfBoundsException:
      String index out of range: 10
        at java.lang.String.charAt(String.java:499)
        at TestExceptions1.main(TestExceptions1.java:11)
```

This error, or exception in *Java* terminology, can be caught and some processing can be done by using the `try` and `catch` statements as shown in the following example:

```
/**
 * Test Exceptions class Application (version 2)
 */
class TestExceptions2 {
  public static void main(String[] args) {

    String s = "Hello";

    try {
      System.out.println(s.charAt(10));
    } catch (Exception e) {
      System.out.println("No such position");
    }
  }
}
```

The output of the execution of the application is:

```
No such position
```

When an exception occurs inside the block defined by `try`, the control is transfered to the block defined by `catch`. This block will process all kinds of exceptions. If we are interested only in processing the exception for index out of bounds for strings, we can do it in the following way:

```
/**
 * Test Exceptions class Application (version 3)
 */

class TestExceptions3 {

  public static void main(String[] args) {

    String s = "Hello";

    try {
      System.out.println(s.charAt(10));
    } catch (StringIndexOutOfBoundsException e) {
      System.out.println("No such position");
    }
  }
}
```

There exist messages that can be sent to an exception object. For example, the next application sends the message `toString` to the exception object `e`:

```
/**
 * Test Exceptions class Application (version 4)
 */

class TestExceptions4 {

  public static void main(String[] args) {

    String s = "Hello";

    try {
      System.out.println(s.charAt(10));
    } catch (StringIndexOutOfBoundsException e) {
      System.out.println("No such position");
      System.out.println(e.toString());
    }
  }
}
```

The output of the execution of the application is:

```
No such position
java.lang.StringIndexOutOfBoundsException:
   String index out of range: 10
```

In a `try` clause, there can be multiple `catch` blocks for all possible exceptions that the clause is intended to handle. It is also possible to specify a special block, by using the keyword `finally`, that will be executed always at the end of the clause, independently if the execution finish normally, or with some exception. It is used normally to release some resource that must be cleaned up.

The next example defines a method that receives a string as argument, which represents an affirmative English sentence, which is expected to end with a dot. It prints to standard output the first character of the sentence and the last character before the dot, using an interesting method of the class `String` that provides `C`-like formatting.

```
Class MultipleCatch {

  public void printInfo(String sentence) {

    try {

      // get first and last char before the dot
      char first = sentence.charAt(0);
      char last = sentence.charAt(sentence.indexOf(".") - 1);
      String out = String.format("First: %c Last: %c",
                           first, last);

      System.out.println(out);
    } catch (StringIndexOutOfBoundsException e1) {
      System.out.println("Wrong sentence, no dot?");
    } catch (NullPointerException e2) {
      System.out.println("Non valid string");
    } finally {
      System.out.println("done!");
    }
  }
}
```

The method has to deal with two exceptions: the first exception can be generated when it is not possible to get the character that is just before the dot, because the dot itself is missing or the sentence consists of the single dot. The second exception can be generated when the string is `null`.

Note that the method `indexOf` returns the index of the first character of the string passed as an argument. If the string is not found, it returns 0.

Consider the following statements:

```
String sentence = "A test sentence.";

MultipleCatch mc = new MultipleCatch();
mc.printInfo(sentence);
```

The output is:

```
First: A Last: e
done!
```

Note that no exception is generated, and the block defined by `finally` is executed.

Consider now the following statements:

```
String sentence = "A test sentence";

MultipleCatch mc = new MultipleCatch();
mc.printInfo(sentence);
```

The output is:

```
Wrong sentence, no dot?
done!
```

The exception `StringIndexOutOfBoundsException` is generated, and the first `catch` block is executed. After that, the block defined by `finally` is also executed.

Consider now the following statements:

```
String sentence = null;

MultipleCatch mc = new MultipleCatch();
mc.printInfo(sentence);
```

The output is:

```
Non valid string
done!
```

The exception `NullPointerException` is generated, and the second `catch` block is executed. After that, the block defined by `finally` is also executed.

There exists a set of predefined exceptions that can be caught. In some cases it is compulsory to catch exceptions. It is also possible to express an interest to not to catch even compulsory exceptions (by using the keyword `throwable`). We will see more examples in sections to follow.

23 Input Output

The input output system in *Java* is rather complex. There are plenty of classes that have to be used in order to read or write data. One advantage is the fact that input output from files, devices, memory or web sites is performed exactly in the same way.

The *Java* input output system is implemented in the package `java.io`. It is based on the idea of streams. An input stream is a data source that can be accessed in order to get data from. An output stream is a data sink, where data can be written to.

The streams are divided into *byte streams* and *character streams*. Byte streams can be used to read or write data in small pieces, like bytes, integers, etc. Character streams can be used to read or write characters.

Java also allows to write and read complete objects (property known as serialization), but we will not be covering it here.

The following subsections introduce the way in which it is possible to work with streams depending on the kind of data we want to work with.

23.1 Byte oriented streams

There exist two classes that can be used for processing byte oriented streams: the class `FileOutputStream` that can be used to write bytes into a stream, and the class `FileInputStream` that can be used to read bytes from a stream.

The next application writes 5 bytes into a file called `file1.data`:

```
/**
 * Write bytes class Application
 */
import java.io.*;

class WriteBytes {

  public static void main(String[] args) {

    int data[] = { 10,20,30,40,255 };

    FileOutputStream f;

    try {
      f = new FileOutputStream("file1.data");

      for(int i = 0;i < data.length;i++)
        f.write(data[i]);

      f.close();
    } catch (IOException e) {
```

```
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

The application defines a reference to a `FileOutputStream`. An instance of this class is created with the keyword `new` by passing a file name as argument. The effect of this operation is to relate the internal object `f` with the file in such a way that, when a write operation is performed on `f`, the data is written into the file.

In this example, all the components of the array are written into the file with the message `write`. The file is closed at the end with the message `close`.

Note that all the operations with the stream are included inside a `try` and `catch` block. This is in fact compulsory, and the compiler will complain if it is not done, due to the fact that an `IOException` can be generated and it must be trapped.

The next example reads data from a file called `file1.data`:

```
/**
 * Read bytes class Application
 */
import java.io.*;

class ReadBytes {

  public static void main(String[] args) {

    FileInputStream f;

    try {
      f = new FileInputStream("file1.data");

      int data;
      while((data = f.read()) != -1)
        System.out.println(data);

      f.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

In this example, an instance of `FileInputStream` is created relating to the file `file1.data`. The bytes are read one after another with the message `read`. This message returns the byte read, or `-1` when the end of file is reached. The

stream is closed with the message `close`. Note also that in this example the `IOException` should be trapped.

The output of the execution of the application is:

```
10
20
30
40
255
```

There exists a message `write` that can be used to store a complete array of bytes into a file. The next example is similar to the class `WriteBytes`, but it writes all the components of a `byte` array at once:

```
/**
 * Write bytes class Application
 */
import java.io.*;

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

    byte data[] = { 10,20,30,40,50 };
    FileOutputStream f;

    try {
      f = new FileOutputStream("file1.data");

      f.write(data,0,data.length);

      f.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

The message `write` receives as arguments the array of bytes, the index of the first component, and the number of components to be written. Even if the array is written in this way, it can be read without problems using the previous example `ReadBytes`. An equivalent message exists for reading an array of bytes at once.

Something that is important to note is the fact that the message `write` expects an integer as argument, and the method `read` returns an integer, instead of a byte. This is due to the fact that a normal byte can take values from -128 to 127, and the bytes written to or read from a file must be in the range 0 to 255. However, the methods that reads or writes a complete array work with bytes.

23.2 Buffered byte oriented streams

In order to minimise the communication overheads it is the usual practice to use buffers. Byte oriented buffered streams can be defined and used with the classes `BufferedOutputStream` and `BufferedInputStream`. It is still necessary to create the streams as was explained in the previous section. The available messages are the same.

The next application shows how to write to a file using buffered streams:

```
/**
 * Write buffered bytes class Application
 */
import java.io.*;

class WriteBufferedBytes {

  public static void main(String[] args) {

    int data[] = { 10,20,30,40,255 };
    FileOutputStream f;
    BufferedOutputStream bf;

    try {
      f = new FileOutputStream("file1.data");
      bf = new BufferedOutputStream(f);

      for(int i = 0;i < data.length;i++)
        bf.write(data[i]);

      bf.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

A buffered output stream object `bf` is created by passing the file output stream object `f` as the argument. In this way, we are expressing our interest in buffering the output stream.

The same applies to reading:

```
/**
 * Read buffered bytes class Application
 */
import java.io.*;

class ReadBufferedBytes {
```

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

    FileInputStream f;
    BufferedInputStream bf;

    try {
      f = new FileInputStream("file1.data");
      bf = new BufferedInputStream(f);

      int data;
      while((data = bf.read()) != -1)
        System.out.println(data);

      bf.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

The output of the execution of the application is:

```
10
20
30
40
255
```

23.3 Data buffered byte oriented streams

A data buffered byte oriented stream can be used to work with data in small pieces corresponding to the primitive types. The messages shown in Table 3 can be used to read and write data.

read message	write message
readBoolean()	writeBoolean(boolean)
readByte ()	writeByte(byte)
readShort()	writeShort(short)
readInt()	writeInt(int)
readLong()	writeLong(long)
readFloat()	writeFloat(float)
readDouble()	writeDouble(double)

Table 3: Methods for buffered byte oriented streams

The next application stores into a data buffered byte oriented stream an integer that corresponds to the size of an array of doubles, the components of the array of doubles, and finally a boolean value:

```
/**
 * Write data class Application
 */
import java.io.*;

class WriteData {

  public static void main(String[] args) {
    double data[] = { 10.3,20.65,8.45,-4.12 };

    FileOutputStream f;
    BufferedOutputStream bf;
    DataOutputStream ds;

    try {
      f = new FileOutputStream("file1.data");
      bf = new BufferedOutputStream(f);
      ds = new DataOutputStream(bf);

      ds.writeInt(data.length);
      for(int i = 0;i < data.length;i++)
        ds.writeDouble(data[i]);

      ds.writeBoolean(true);

      ds.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

Note that a data buffered byte oriented stream is created in three steps: first the file output stream is created, then it is buffered, and finally the data stream is created.

The next application reads data from a data stream:

```
/**
 * Read data class Application
 */
```

```
import java.io.*;

class ReadData {

  public static void main(String[] args) {

    FileInputStream f;
    BufferedInputStream bf;
    DataInputStream ds;

    try {
      f = new FileInputStream("file1.data");
      bf = new BufferedInputStream(f);
      ds = new DataInputStream(bf);

      int length = ds.readInt();
      for(int i = 0;i < length;i++)
        System.out.println(ds.readDouble());

      System.out.println(ds.readBoolean());

      ds.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

The output of the execution of the application is:

```
10.3
20.65
8.45
-4.12
true
```

23.4 Character oriented streams

The character oriented streams can be used to read and write characters. In order to create an output text stream it is necessary to create an instance of a `FileWriter` and then an instance of a `BufferedWriter`. There exists three methods that can be used to write data into this kind of streams. They are shown in Table 4.

The first message can be used to write characters from a string, starting from the position indicated by the first integer and as many as indicated by the second integer. The second message is similar, but the characters are written from an

message
write(String,int,int)
write(char[],int,int)
newLine()

Table 4: Methods for character oriented streams

array of characters. The message `newLine` generates a newline in the output stream independent of the convention used in the current operating system.

The next application writes some selected characters from two strings into a character oriented output stream:

```
/**
 * Write text class Application
 */
import java.io.*;

class WriteText {
  public static void main(String[] args) {
    FileWriter f;
    BufferedWriter bf;

    try {
      f = new FileWriter("file1.text");
      bf = new BufferedWriter(f);

      String s = "Hello World!";
      bf.write(s,0,s.length());
      bf.newLine();
      bf.write("Java is nice!!!",8,5);
      bf.newLine();

      bf.close(); }
     catch (IOException e) {
     System.out.println("Error with files:"+e.toString());
    }
  }
}
```

The content of the file after the execution of the application is:

```
Hello World!
nice!
```

In order to read from a text oriented stream it is necessary to create an instance of a file reader, and then an instance of a buffered reader. The message

`readLine` can be used to read complete lines from the text file. It returns an instance of a `String` containing the line, or the `null` reference at end of file.

The next application reads lines from a buffered text oriented stream:

```
/**
 * Read text class Application
 */
import java.io.*;

class ReadText {

  public static void main(String[] args) {

    FileReader f;
    BufferedReader bf;

    try {
      f = new FileReader("file1.text");
      bf = new BufferedReader(f);

      String s;
      while ((s = bf.readLine()) != null)
        System.out.println(s);

      bf.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

23.5 Standard input

Sometimes the applications need to read from the standard input. The standard input of an application can be referenced in *Java* with the object `System.in`. In order to read from it, it is necessary to define an `InputStreamReader` and a `BufferedReader`, as the next example shows:

```
/**
 * Standard input class Application
 */
import java.io.*;

class StandardInput {

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

```
    InputStreamReader isr;
    BufferedReader br;

    try {
      isr = new InputStreamReader(System.in);
      br = new BufferedReader(isr);

      String line;
      while ((line = br.readLine()).length() != 0)
        System.out.println(line);
    } catch(IOException e) {
        System.out.println("Error in standard input");
    }
  }
}
```

The method `readLine` returns a line from the standard input as a string. Note that the method `length` is called on this returned string in order to check if the standard input was closed.

The application just copies its standard input into its standard output.

As it was stated before, methods can express their interest in not to catch some specific exceptions. This is done by using the keyword `throws` in the method specification. In this case, it is not necessary to define the `try` and `catch` block.

The next application is the same as the previous one, but the method `main` throws the exception `IOException`:

```
/**
 * Standard input class Application (throws IOException)
 */
import java.io.*;

class StandardInputWithThrows {

  public static void main(String[] args) throws IOException {
    InputStreamReader isr;
    BufferedReader br;

    isr = new InputStreamReader(System.in);
    br = new BufferedReader(isr);

    String line;
    while ((line = br.readLine()).length() != 0)
      System.out.println(line);
  }
}
```

Note that in this example application, the `try` and `catch` block was not defined.

From Java 5.0 (JDK 1.5.0) the handling of standard input devices is simplified, with the use of the `Scanner` class. Using the `Scanner` class, an object can be created to read input from `System.in` in a very simple way. What follows is an example of an application that prints the summation of a set of integers read from standard input.

```
/**
 * Sum integers read from System.in with the Scanner class
 */
import java.io.*;
import java.util.*;

class ReadWithScanner {

  public static void main(String[] args) throws IOException {

      Scanner sc = new Scanner(System.in);
      int sum = 0;
      while (sc.hasNextInt()) {
          int anInt = sc.nextInt();
           sum += anInt;
      }
      System.out.println(sum);
  }
}
```

The scanner object reads and splits the inputs in tokens using a delimiter pattern, which by default is a whitespace. The resulting tokens may then be converted into values of different types using the various `next` methods.

The `Scanner` class can also be used to read from a file by creating an instance as follow:

```
Scanner from_file = new Scanner(new File("file.data"));
```

Please refer to the Java 5.0 API documentation for more details.

24 Threads

In *Java* it is possible to run concurrently different tasks called threads. Each thread can be seen as an independently running task, with some CPU time assigned to it. These threads can communicate between themselves and their access to shared data can be synchronized.

In order to define a thread, it is necessary to create a subclass of the class `Thread`. The class `Thread` has an abstract method called `run`, that has to be

defined in the subclass. This method has to contain the code that will be running as an independent thread.

The next example defines a class called `CharThread` that is a subclass of `Thread`, and defines the method `run`:

```
/**
 * Char thread class Application
 */


class CharThread extends Thread {
  char c;

  CharThread(char aChar) {
    c = aChar;
  }

  public void run() {
    while (true) {
      System.out.println(c);
      try {
        sleep(100);
      } catch (InterruptedException e) {
        System.out.println("Interrupted");
      }
    }
  }
}
```

This class defines a character data member that is initialized with a character value when an instance of the class is created with the constructor.

The method `run` contains an infinite loop where the character is printed, and puts the thread to sleep for 100 milliseconds. Note that an exception can be generated when the thread is sleeping, so code has to be defined in order to catch it.

The next application creates two instances of this class, each initialized with a different character. Then both of them are started as a thread, so the run methods of both instances will be executed concurrently:

```
/**
 * test threads class Application
 */

class TestThreads {

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

```
    CharThread t1 = new CharThread('a');
    CharThread t2 = new CharThread('b');

    t1.start();
    t2.start();
  }
}
```

Note that the two instances of `CharThread` are created by calling the constructor. Both threads are started by sending the message `start`.

The output of the execution of the application is:

```
a
b
a
b
a
b
a
b
a
b
a
b
...
```

Both threads get the CPU for some time quantum, so the output produced by them is intermixed.

24.1 The Producer and Consumer example

The producer and consumer problem is a standard example that illustrates concurrency and the problems associated with it. The idea is that there exist two processes that interact through a common buffer into which the producer stores the items produced and from which the consumer consumes these items.

A synchronization problem arise since both processes have to interact with the same buffer. The buffer is then a critical resource. The other problem arises from the fact that the producer cannot put items into a full buffer, and the consumer cannot consume items from an empty buffer.

The next application shows a possible implementation of the main class for the producer and consumer problem:

```
/**
 * Producer Consumer class Application
 */
```

```
class ProducerConsumer {

  public static void main(String[] args) {

    Buffer buffer = new Buffer(20);

    Producer prod = new Producer(buffer);
    Consumer cons = new Consumer(buffer);

    prod.start();
    cons.start();
  }
}
```

This class creates a buffer with 20 empty slots, and then an instance of the producer thread and an instance of the consumer thread. The common buffer is passed as an argument to the constructor, so both threads will share the buffer. After that, both threads are started.

A corresponding producer class can be defined as follows:

```
/**
 * Producer class Application
 */
class Producer extends Thread {
  Buffer buffer;

  public Producer(Buffer b) {
    buffer = b;
  }

  public void run() {
    double value = 0.0;

    while (true) {
      buffer.insert(value);
      value += 0.1;
    }
  }
}
```

The producer class is a subclass of `Thread`, so it must have a method `run` that can be executed as an independent thread. The class defines one data member that will contain a reference to the buffer passed as argument to the constructor. The method run inserts a double value into the buffer inside an infinite loop. So the producer is producing double values that are inserted into the common buffer.

Similarly a consumer class can be defined as follows:

```
/**
 * Consumer class Application
 */

class Consumer extends Thread {
  Buffer buffer;

  public Consumer(Buffer b) {
    buffer = b;
  }

  public void run() {

    while(true) {
      System.out.println(buffer.delete());
    }
  }
}
```

The consumer class is also a subclass of `Thread`. The class defines a data member that will contain a reference to the common buffer. The `run` method, ie., the one that will be executed as a thread, just removes data from the buffer, and prints it onto standard output.

The Buffer is defined as a circular buffer implemented with one array and two pointers, one for the head position and the other for the tail position. Data is inserted in the tail position, and data is read from the head position. There is one data member used to store the number of elements currently available in the buffer. The Buffer application is shown below:

```
/**
 * Buffer class Application
 */

class Buffer {

  double buffer[];

  int head = 0;
  int tail = 0;
  int size = 0;
  int numElements = 0;

  public Buffer(int s) {
    buffer = new double[s];
```

```
    size = s;
    numElements = 0;
  }

  public void insert(double element) {

    buffer[tail] = element;
    tail = (tail + 1) % size;
    numElements++;
  }

  public double delete() {

    double value = buffer[head];
    head = (head + 1) % size;
    numElements--;
    return value;
  }
}
```

Although this implementation seems to be correct, it does not work for two reasons which are discussed below.

- Both methods `insert` and `delete` operate concurrently over the same structure. It is necessary to define a critical region, or in other words, to prevent both methods accessing the buffer concurrently. If one thread is inserting data the other must wait till the first one has finished, and vice-versa.
- The `insert` method does not check if there is at least one free slot in the buffer, and the `delete` method does not check if there is at least one data value available in the buffer.

The following subsections will further discuss these problems and their solutions.

24.2 synchronized methods

In *Java* it is possible to define synchronized methods. These methods are not allowed to be executed concurrently in the same instance. Each instance has a lock that is used to synchronize the access.

The solution to the first problem is to define the methods as follows, using the keyword `syncronized`:

```
  public synchronized void insert(double element) {

    buffer[tail] = element;
```

```
    tail = (tail + 1) % size;
    numElements++;
  }

  public synchronized double delete() {

    double value = buffer[head];
    head = (head + 1) % size;
    numElements--;
    return value;
  }
```

Synchronized methods allow to implement the concept of *mutual exclusion*, where no more than one process can get access to shared data at the same time. Note that it is possible to have many consumers and many producers, all of them dealing with the same buffer. It is also possible to create more than one buffer, with each instance having its own lock.

24.3 wait and notify

Subclasses of `Thread` can send messages `wait` and `notify`. The messages can be sent only from synchronized methods. The message `wait` puts the calling thread to sleep, releasing the lock. The message `notify` awakens a waiting thread on the corresponding lock.

In our example, the thread that is going to insert a value into the buffer has to put itself to sleep when there is no empty slots in the buffer. The thread that is going to remove a value from an empty buffer has to put itself to sleep to be awakened when a value is available in the buffer.

The thread that has just inserted data into an empty buffer has to notify the delete thread so it can be awakened. The thread that has just removed data from a full buffer has to notify the insert thread so it can be awakened.

The correct code for the implementation of both methods is as follows:

```
  public synchronized void insert(double element) {
    if (numElements == size) {
      try {
        wait();  // go to sleep
       } catch(InterruptedException e) {
       System.out.println("Interrupted");
      }
    }
    buffer[tail] = element;
    tail = (tail + 1) % size;
    numElements++;
    notify();  // wake any thread waiting
  }
```

```
  public synchronized double delete() {

    if (numElements == 0) {
      try {
        wait();  // go to sleep
      } catch(InterruptedException e) {
        System.out.println("Interrupted");
      }
    }
    double value = buffer[head];
    head = (head + 1) % size;
    numElements--;
    notify();  // wake any thread waiting
    return value;
  }
```

Note that it is necessary to catch an exception when `wait` is used. The listing of the `Buffer` class is provided in Appendix D.

25 JAR files

When we were compiling the `ProducerConsumer` example four `class` files were generated as the following command shows:

```
# ls *.class
Buffer.class
Consumer.class
ProducerConsumer.class
Producer.class
```

In order to distribute the executable of this application it is necessary to copy these four files.

Java provides a mechanism to pack and compress files into one file in order to make the process of distribution of applications easier. This compressed file is called a JAR (*Java* ARchive) file.

A JAR file can be created and manipulated by the command `jar`. In order to create a JAR file, it is necessary to define a *manifest* file. The manifest file contains information on the files included in the JAR file. The command `jar` creates a default manifest file in the directory `META-INF` with name `MANIFEST.MF`, just below the current directory.

It is possible to add specific lines to this manifest file by passing, as an argument to `jar`, the name of a text file that contains these lines. Information is specified as (*key*,*value*) pairs. In the Producer Consumer example, the only necessary pair that has to be specified is the name of the class that contains the

`main` function. It can be done in a text file (called mylines.txt in our example) with the following content:

```
# cat mylines.txt
Main-Class: ProducerConsumer
```

The creation of a JAR file for this application can be done as follows:

```
# jar cmf mylines.txt ProducerConsumer.jar
  ProducerConsumer.class Producer.class Consumer.class
  Buffer.class
```

The option `c` specifies creation of a JAR file, `m` that a text file with lines to be added to the manifest file will be supplied in the command line, and `f` that the name of the JAR file will be also supplied in the command line. `mylines.txt` contains the lines to be added to the manifest file, `ProducerConsumer.jar` is the expected output file, and the file names that follow are the names of the files to be added to the JAR file.

It is possible to see the contents of the JAR file just created by using the option `t` as follows:

```
# jar tf ProducerConsumer.jar
META-INF/
META-INF/MANIFEST.MF
ProducerConsumer.class
Producer.class
Consumer.class
Buffer.class
```

Note that a manifest file was added with the following contents:

```
Manifest-Version: 1.0
Main-Class: ProducerConsumer
Created-By: 1.5.0 (Sun Microsystems Inc.)
```

The application included in the JAR file can be executed as follows:

```
# java -jar ProducerConsumer.jar
```

It is possible to extract and update the contents of a JAR file. Please refer to *Java* documentation for further details and examples.

26 Ant

Ant is a building tool that provides support to compile, pack, deploy and document *Java* applications. In some sense, its functionality is similar to the `make` command, except that the approach is completely different. The `make` specifications are organized as a set of shell commands whereas the specifications for the `ant` command are defined in terms of XML sentences. A build file defines a project which consists of a set of tasks. The building process is executed by calling out the target of tasks, where each task is run by an object that implements a particular Task interface. The main advantage of this approach is portability, since the build files are independent of the operating system. A not so big disadvantage is that the shell commands cannot be used for specifications.

The `ant` specification is usually written in a file called `build.xml`. This file follows the XML syntax, and it specifies a project (identified with the tag `project`), which consists of a set of targets (identified with the tag `target`), which in turn consists of a set of task elements. Each task element corresponds to a specific task to be performed by `ant`.

26.1 A first example

This section introduces `ant` with a simple example that can be used to compile a *Java* application. Let us assume that the current directory contains the file `HelloWorld.java` developed in section 3. A possible definition of the file `build.xml` is as follows:

```
<?xml version="1.0"?>
<!-- first build file -->

<project name="HelloWorld" default="build" basedir=".">
    <target name="build">
        <javac srcdir="." />
    </target>
</project>
```

The first line specifies that this file is an XML file, the second line is a comment and the rest specifies the actions to be taken for this project. The project is called `HelloWorld`, its default action (target) is called `build`, and all directories will be considered relative to the current directory, which is specified by the value given to the attribute `basedir`. Executing the project means just follow one target with name `build`. This target involves only one task element, which is the compilation of all *Java* source files in the current directory.

In order to compile the file, it is enough to execute the command `ant` as the following example shows:

```
# ant
Buildfile: build.xml
```

```
build:
    [javac] Compiling 1 source file

BUILD SUCCESSFUL

Total time: 3 seconds
```

26.2 Projects, targets, task elements and properties

As it was explained before, the file `build.xml` specifies a project, and consists of a set of targets, with each one containing a set of task elements.

A project specifies three attributes: the name of the project (`name`), the default target (`target`) that will be called when no target is specified in the command line, and the base directory (`basedir`), that provides the base for all relative references to directories in the build file. The file `build.xml` in the previous section shows an example of a project definition.

A target specifies five attributes: the name of the target (`name`), the list of targets on which it depends (`depends`), the name of the property that must be set (`if`), the name of the property that must not be set (`unless`) and a comment (`description`). Except for the name of the target, the other attributes are optional. The targets in a project can depend on other targets in the same project. `ant` checks the dependencies between the different targets to determine the order in which the different targets will be executed.

The properties can be used to hold a value. The value assigned to a property cannot be changed, so they are not variables. As an example, the following XML sentences assign the value `yes` to the variable `conditionOK` and the value `src` to the variable `src-dir`:

```
<property name="conditionOK" value="yes"/>
<property name="src-dir"     location="src"/>
```

The values can be accessed by placing the property name between `"${"` and `"}"`. Section 26.3 provides examples of its use.

26.3 A more complicated example

This section shows a file `build.xml` that can be used as a model for building specification files for `ant`. The project called `Complex` can be used to create the TestComplex application developed in section 15. This application consists of two files: `Complex.java` and `TestComplex.java`, which are stored in a subdirectory with name `src`.

The project defines a set of four targets: `init`, `build`, `dist` and `clean`, that can be used respectively to initialize the directory structure, compile the source files, create the `jar` file and clean directories and object files. The file `build.xml` stored in the top directory is as follows:

```
<?xml version="1.0"?>
<!-- first build file -->

<project name="Complex" default="dist" basedir=".">

  <!-- set global properties -->
  <property name="src-dir"   location="src"/>
  <property name="build-dir" location="build"/>
  <property name="dist-dir"  location="dist"/>

  <target name="init" description="initial task">
    <!-- Create the build directory -->
    <mkdir dir="${build-dir}"/>
  </target>

  <target name="build" depends="init" description="compile task">
    <javac srcdir="${src-dir}" destdir="${build-dir}"/>
  </target>

  <target name="dist" depends="build"
   description="build distribution" >

   <!-- Create the distribution directory -->
   <mkdir dir="${dist-dir}"/>

   <!-- Create the jar file -->
   <jar jarfile="${dist-dir}/complex.jar" basedir="${build-dir}">
     <include name="*.class"/>
     <manifest>
       <attribute name="Main-Class" value="TestComplex"/>
     </manifest>
   </jar>
  </target>

  <target name="clean" description="clean up" >
    <delete dir="${build-dir}"/>
    <delete dir="${dist-dir}"/>
  </target>

</project>
```

The project is called `Complex`. Its default target is `dist` and all directory references are relative to the current directory. A set of properties are defined before any action is taken:

```
<!-- set global properties -->
```

```
<property name="src-dir"   location="src"/>
<property name="build-dir" location="build"/>
<property name="dist-dir"  location="dist"/>
```

The property `src-dir` contains the value `src`, which is the directory in which the source files are stored. The property `build-dir` contains the value `build`, which is the directory in which the class files will be stored, and the property `dist-dir` with the value `dist`, is the directory in which the final `jar` file will be created.

The target `init` just creates the directory specified by the variable `build-dir`, which will be `build` in this case:

```
<target name="init" description="initial task">
  <!-- Create the build directory -->
  <mkdir dir="${build-dir}"/>
</target>
```

The target `build` is defined as follows:

```
<target name="build" depends="init" description="compile task">
  <javac srcdir="${src-dir}" destdir="${build-dir}"/>
</target>
```

Note that it depends on the target `init`, meaning that `init` has to be executed before `build`. Its task is to compile all java source files in the directory `src` and to leave all class files in the directory `build`, as specified respectively by the attributes `srcdir` and `destdir`.

The target `dist` is more complex:

```
<target name="dist" depends="build"
  description="build distribution" >

  <!-- Create the distribution directory -->
  <mkdir dir="${dist-dir}"/>

  <!-- Create the jar file -->
  <jar jarfile="${dist-dir}/complex.jar" basedir="${build-dir}">
    <include name="*.class"/>
    <manifest>
      <attribute name="Main-Class" value="TestComplex"/>
    </manifest>
  </jar>
</target>
```

It depends on `build`, meaning that the source files must be compiled before attempting to build the jar file. Its first action is to create the directory `build`. Then it creates a jar file with name `complex.jar` in the `dist` directory,

as specified by the attribute `jarfile`. The base directory for the file references is `build`, as specified by the attribute `basedir`, and the files that have to be included are those with extension .class, as specified by the task element `include` with attribute `name`. The manifest file is specified with the task `manifest`, with the attribute `Main-class` with value `TestComplex` (see section 15 for details on manifest file construction for this example).

The target `clean` just removes the directories created during the process:

```
  <target name="clean" description="clean up" >
    <delete dir="${build-dir}"/>
    <delete dir="${dist-dir}"/>
  </target>
```

An example of its execution follows:

```
# ant
Buildfile: build.xml

init:
    [mkdir] Created dir: ComplexNumbers/build

build:
    [javac] Compiling 2 source files to ComplexNumbers/build

dist:
    [mkdir] Created dir: ComplexNumbers/dist
      [jar] Building jar: ComplexNumbers/dist/complex.jar

BUILD SUCCESSFUL

Total time: 11 seconds
```

It is possible to ask `ant` to execute just a specific target, by specifying it in the command line, as the following example shows:

```
# ant clean
Buildfile: build.xml

clean:
   [delete] Deleting directory ComplexNumbers/build
   [delete] Deleting directory ComplexNumbers/dist

BUILD SUCCESSFUL

Total time: 3 seconds
```

```
# ant build
init:
    [mkdir] Created dir: ComplexNumbers/build

build:
    [javac] Compiling 2 source files to ComplexNumbers/build

BUILD SUCCESSFUL

Total time: 7 seconds
```

Ant is quite more powerful than what the examples have shown. For more information please refer to the Ant manual.

27 Java on the TINI

The TINI (Tiny InterNet Interface) is a platform developed by Dallas Semiconductors that can be programmed in *Java*. Its Java API implements most of the classes in the core packages `java.lang`, `java.io`, `java.net` and `java.util`. A new package called `com.dalsemi` provides support for TINI's unique capabilities.

In order to be able to execute on TINI an application developed in *Java* on another platform, it is necessary to follow a four step process: (1) the source code has to be compiled using a standard *Java* compiler, (2) the resulting class file has to be converted to the special format required by TINI, (3) this is then downloaded to the TINI board and (4) executed there. The following example shows these four steps when creating the `HelloWorld` application developed in section 3.

Step 1: Compiling the source code : The source code is compiled using any *Java* compiler. For example, by using the Sun JDK, the command is:

```
# javac HelloWorld.java
```

Step 2: Converting the .class file : The .class file generated in the previous step is converted to the .tini format required by the TINI board by executing the program `TINIConvertor`:

```
# java -classpath /tini/bin/tini.jar TINIConvertor
       -f HelloWorld.class
       -d /tini/bin/tini.db -o HelloWorld.tini
```

Note that the program `TINIConvertor` is a *Java* application included in the file `tini.jar`. That is why the program is executed by calling the *Java* interpreter specifying the jar file as the class path. The flag `-f` is used to specify the .class file name and the flag `-o` to specify the .tini file name. The file `tini.db` is the TINI API database. This example assumes that the TINI development system is located in the directory `/tini`.

Step 3: Downloading the .tini file : The .tini file is downloaded to TINI by using ftp as follows.

```
# ftp tini
Connected to tini.
220 Welcome to slush. (Version 1.11) Ready for user login.
User (tini:(none)): root
331 root login allowed. Password required.
Password:
230 User root logged in.
ftp> bin
200 Type set to Binary
ftp> put HelloWorld.tini
200 PORT Command successful.
150 BINARY connection open, putting HelloWorld.tini
226 Closing data connection.
ftp: 183 bytes sent in 0.00 Seconds.
ftp> bye
```

Step 4: Executing on the TINI board : The program is executed by calling the *Java* interpreter on TINI in a telnet session:

```
# telnet tini
Connected to tini.
Escape character is '^]'.

Welcome to slush.  (Version 1.11)

tini00a93c login: root
tini00a93c password:
TINI /> java HelloWorld.tini
HelloWorld
```

27.1 Using Ant

Since this process is rather tedious, it is usually automated by using makefiles, ant files or the graphical user interfaces provided by some development environments. In this section, we will explore a solution based on ant, using a package called tiniant. This package provides the `tini` task element that can be used to convert a .class file into a .tini class. The most important attributes are `outputfile` to specify the output file name, `database` to specify the TINI API database, `include` for the source file names and `classpath` for the location of the TINI development kit. It is also necessary to specify the `convert` file set that specifies what to include and what to exclude in the output file. For example, the file `build.xml` for compiling the application `HelloWorld` stored in the directory `src` can be defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="HelloWorld" default="convert" basedir=".">

  <taskdef name="tini" classname="net.geeba.ant.Tini"/>

  <property name="tini.dir" value="/tini/"
  <property name="tini.db" value="${tini.dir}/bin/tini.db"/>
  <property name="tini.classes"
            value="${tini.dir}/bin/tiniclasses.jar"/>
  <property name="tini.jar" value="${tini.dir}/bin/tini.jar"/>

  <target name="init" description="initialize">
    <mkdir dir="build"/>
  </target>

  <target name="build" depends="init" description="compile">
    <javac srcdir="src" destdir="build"
           bootclasspath="${tini.classes}"/>
  </target>

  <target name="convert" depends="build" description="convert">
    <tini outputfile="HelloWorld.tini" database="${tini.db}"
          classpath="${tini.jar}">
      <convert dir="build"/>
    </tini>
  </target>

  <target name="clean" description="clean">
    <delete dir="build"/>
    <delete file="HelloWorld.tini"/>
  </target>
</project>
```

The application can be compiled with the command `ant`:

```
# ant
Buildfile: build.xml

init:
    [mkdir] Created dir: HelloWorldAnt/build

build:
    [javac] Compiling 1 source file to HelloWorldAnt/build

convert:
```

```
     [tini] TINIConvertor (KLA)
     [tini] Version 1.24 for TINI 1.1 (Beta 2 and later ONLY!!!)
     [tini] Built on or around March 20, 2002
     [tini] Copyright (C) 1996 - 2002 Dallas Semiconductor Corp.
     [tini] Loading class HelloWorldAnt/build/HelloWorld.class
            from file HelloWorldAnt/build/HelloWorld.class
     [tini] Getting UNMT...there are 0 user native methods
     [tini]
     [tini] Class HelloWorld, size 125, CNUM 8000, TH Contrib: 19
     [tini]
     [tini] Initial length of the application: 125
     [tini] Output file created in 424 milliseconds.
     [tini] Output file written in 1 milliseconds.
     [tini] Output file size : 472
     [tini] Number of string table entries: 1

BUILD SUCCESSFUL

Total time: 8 seconds
```

The third line in the file `build.xml`:

```
<taskdef name="tini" classname="net.geeba.ant.Tini"/>
```

is used to include the task `tini` in order to make it available for use in the specifications file. The tasks `init` and `clean` are defined in the same way as they were defined in section 26.3. The task `build` compiles the file `HelloWorld.java` to `HelloWorld.class` by including also the specific API class files developed for TINI with the attribute `bootclasspath`:

```
<target name="build" depends="init" description="compile">
  <javac srcdir="src" destdir="build"
         bootclasspath="${tini.classes}"/>
</target>
```

The task `convert` converts `HelloWorld.class` to `HelloWorld.tini` as specified by the attribute `outputfile`, using the database specified by the property `tini.db` and including the class definitions from `tini.jar`:

```
<target name="convert" depends="build" description="convert">
  <tini outputfile="HelloWorld.tini" database="${tini.db}"
        classpath="${tini.jar}">
    <convert dir="build"/>
  </tini>
</target>
```

It is possible to automate the download procedure by using a shell script for the FTP transfer and make ant to call it after the building process is finished. The following shell script called `deploy.sh` transfers the file `HelloWorld.tini` to the TINI board:

```
#!/bin/sh
ftp -n $1 <<!
user $2 $3
bin
put HelloWorld.tini
close
quit
!
```

It initiates an FTP session by using the board name, the user name and the password passed as arguments in the command line. Note that it is possible to add more FTP commands before the close statement. An example of its use from the command line follows:

```
# ./deploy.sh tini root tini
Connected to tini.
220 Welcome to slush. (Version 1.11)
331 Password required for root.
230 User root logged in.
200 Type set to I.
local: HelloWorld.tini remote: HelloWorld.tini
150 Opening BINARY mode data connection for 'HelloWorld.tini'.
226 Transfer complete.
472 bytes sent in 0.00 Seconds.
221 Goodbye.
```

A special ant task called `exec` allows to call a script from the file `build.xml`. The executable file is specified with the attribute `executable`, the base directory with `dir` and the arguments with the FileSet `arg`. The attribute `output` can be used to specify a file where the output of the command execution will be stored. The following task definition can be used to download `HelloWorld.tini` to the TINI board with name `tini`, user account `root` and password `tini`:

```
  <target name="deploy" depends="convert">
    <exec dir="." executable="deploy.sh">
      <arg value="tini"/>
      <arg value="root"/>
      <arg value="tini"/>
    </exec>
  </target>
```

The TINI application can then be downloaded by using ant as follows:

```
# ant deploy
```

A The Book example

```
/**
 * Books Application
 */

class Book {
  String title;
  String author;
  int numberOfPages;
  String ISBN;
  static String owner;

  /** This constructor creates a Book with a specified title,
   *  author, number of pages and unknown ISBN
   */

  Book(String tit,String aut,int num) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = "unknown";
  }

  /** This constructor creates a Book with a specified title,
   *  author, number of pages and ISBN
   */

  Book(String tit,String aut,int num,String isbn) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = isbn;
  }

  /** This method returns a string containing the initials of
   *  the author
   */

  public String getInitials() {
    String initials = "";

    for(int i = 0;i < author.length();i++) {
      char currentChar = author.charAt(i);
      if (currentChar >= 'A' && currentChar <='Z') {
        initials = initials + currentChar + '.';
```

```
      }
    }
    return initials;
  }

  /** This method returns true if both the receptor and the
   *  argument correspond to the same book
   */

  public boolean equals(Book b) {
    return (title.equals(b.title) && author.equals(b.author) &&
            numberOfPages == b.numberOfPages &&
            ISBN.equals(b.ISBN));
  }

  /** This method sets the owner of the book
   */

  public void setOwner(String name) {
    owner = name;
  }

  /** This method gets the owner of the book
   */

  public String getOwner() {
    return owner;
  }

  /** This method returns a description of the book
   */

  public static String description() {
    return "Book instances can store information on books";
  }
}
```

B The Complex number example

```
/**
 * Complex Number class
 */

public class Complex {
  private double real;         // real part
  private double im;           // imaginary part

  /** This constructor creates a complex number from its real
   *  and imaginary part.
   */

  Complex(double r,double i) {
    real = r;
    im = i;
  }

  /** This constructor creates a complex number as a copy
   *  of the complex number passed as argument
   */

  Complex(Complex c) {
      this(c.real,c.im);
  }

  /** This method returns the real part
   */

  public double getReal() {
    return real;
  }

  /** This method returns the imaginary part
   */

  public double getImaginary() {
    return im;
  }

  /** This method returns a new complex number wich is
   *  the result of the addition of the receptor and the
   *  complex number passed as argument
   */
```

```
public Complex add(Complex c) {
  return new Complex(real + c.real,im + c.im);
}

/** This method returns a new complex number wich is
 *  the result of the substraction of the receptor and the
 *  complex number passed as argument
 */

public Complex sub(Complex c) {
  return new Complex(real - c.real,im - c.im);
}

/** This method returns a new complex number wich is
 *  the result of the product of the receptor and the
 *  complex number passed as argument
 */

public Complex mul(Complex c) {
  return new Complex(real * c.real - im * c.im,
                     real * c.im + im * c.real);
}

/** This method returns a new complex number wich is
 *  the result of the product of the receptor and the
 *  complex number passed as argument
 */

public Complex div(Complex c) {
  double r,i;

  if (Math.abs(c.real) >= Math.abs(c.im)) {
    double n = 1.0 / (c.real + c.im * (c.im / c.real));
    r = n * (real + im * (c.im / c.real));
    i = n * (im - real * (c.im / c.real));
  } else {
    double n = 1.0 / (c.im + c.real * (c.real / c.im));
    r = n * (im + real * (c.real / c.im));
    i = n * (- real + im * (c.real / c.im));
  }
  return new Complex(r,i);
}

/** This method returns a new complex number wich is
 *  the result of the scaling the receptor by the
 *  argument
```

```
   */

  public Complex scale(double c) {
    return new Complex(real * c,im * c);
  }

  /** This method computes the norm of the receptor
   */

  public double norm() {
    return Math.sqrt(real * real + im * im);
  }

  /** This method increments the real part by a value
   *  passed as argument. Note that the method modifies
   *  the receptor
   */

  public Complex addReal(double c) {
    real += c;
    return this;
  }

  /** This method returns a string representation of
   *  the receptor
   */

  public String asString() {
    return "" + real + " + i * " + im;
  }
}
```

C The Scientific Book example

```
/**
 * Scientific Book Class
 */

class ScientificBook extends Book {
  String area;
  boolean proceeding = false;

  /** This constructor creates a Scientific Book with a
   *  specified title, author, number of pages, ISBN and
   *  area. Proceeding is set to false
   */

  ScientificBook(String tit,String aut,int num,String isbn,
                 String a) {
    super(tit,aut,num,isbn);
    area = a;
  }

  /** This method returns true if both the receptor and the
   *  argument correspond to the same book
   */

  public boolean equals(ScientificBook b) {
    return super.equals(b) && area.equals(b.area) &&
           proceeding == b.proceeding;
  }

  /** This method returns a description of the book
   */

  public static String description() {
    return "ScientificBook instances can store information" +
           " on scientific books";
  }

  /** This method sets proceeding to true
   */

  public void setProceeding() {
    proceeding = true;
  }

  /** This method sets proceeding to false
```

```
   */

  public boolean isProceeding() {
    return proceeding;
  }
}
```

D The Producer and Consumer example

```
/**
 * Producer Consumer class Application
 */

class ProducerConsumer {

  /** This method creates a common buffer, and starts two
   *  threads: the producer and the consumer
   */

  public static void main(String[] args) {

    // creates the buffer
    Buffer buffer = new Buffer(20);

    Producer prod = new Producer(buffer);
    Consumer cons = new Consumer(buffer);

    // start the threads
    prod.start();
    cons.start();
  }
}

/**
 * Producer class Application
 */

class Producer extends Thread {
  Buffer buffer;

  /** This constructor initialize the data member buffer as
   *  a reference to the common buffer
   */

  public Producer(Buffer b) {
    buffer = b;
  }

  /** This method executes as a thread. It keeps inserting
   *  a value into the buffer
   */

  public void run() {
```

```
    double value = 0.0;

    while (true) {
      buffer.insert(value);
      value += 0.1;
    }
  }

/**
 * Consumer class Application
 */

class Consumer extends Thread {
  Buffer buffer;

  /** This constructor initialize the data member buffer as
   *  a reference to the common buffer
   */

  public Consumer(Buffer b) {
    buffer = b;
  }

  /** This method executes as a thread. It keeps removing
   *  values from the buffer, and printing them
   */

  public void run() {

    while(true) {
    System.out.println(buffer.delete());
    }
  }
}

/**
 * Buffer class Application
 */

class Buffer {

  double buffer[];

  int head = 0;
  int tail = 0;
  int size = 0;
```

```
int numElements = 0;

/** This constructor initialize the data member buffer as
 *  an array of doubles. The size of the array is also
 *  initialized
 */

public Buffer(int s) {
  buffer = new double[s];
  size = s;
  numElements = 0;
}

/** This method inserts an element into the circular
 *  buffer. The thread goes to sleep if there is no empty
 *  slots, and notify waiting threads after inserting an
 *  element
 */

public synchronized void insert(double element) {

  if (numElements == size) {
    try {
      wait();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
  buffer[tail] = element;
  tail = (tail + 1) % size;
  numElements++;
  notify();
}

/** This method removes an element from the circular
 *  buffer. The thread goes to sleep if there is no element
 *  to remove, and notify waiting threads after removing an
 *  element
 */

public synchronized double delete() {

  if (numElements == 0) {
    try {
      wait();
    } catch(InterruptedException e) {
```

```
        System.out.println("Interrupted");
      }
    }
    double value = buffer[head];
    head = (head + 1) % size;
    numElements--;
    notify();
    return value;
  }
}
```

References

[1] Arnold K, Gosling J and Holmes D, ***The Java programming Language***, Prentice Hall, 2005.

[2] Eckel B, ***Thinking in* Java *(4th Edition)*** Prentice Hall, 2006. Third edition available on-line at `http://www.mindview.net/Books/TIJ`

[3] Flanagan D, **Java *in a Nutshell: A desktop quick reference (3rd Edition)***, O'Reilly, 1999.

[4] Holzner S, **Java *2***, The Coriolis Group, 2000.

[5] Hortsmann C, ***Computing concepts with* Java *2, essentials (2nd Edition)***, Wiley, 2000.

[6] Horstmann C and Cornell G, ***Core* Java*, Volume II – Advanced Features***, The Sun Microsystems Press, 2000.

[7] Horstmann C and Cornell G, ***Core* Java*, Volume I – Fundamentals***, The Sun Microsystems Press, 2001.

[8] Lemay L and Cadenhead R, **Java *2. Guida Completa***. Apogeo, 2000.

[9] Sun Microsystems, ***The* Java *tutorial. A practical guide for programmers***. Available online at `http://java.sun.com`

[10] Loomis D, ***The TINI specification and developer's guide***, Addison Wesley Professional, 2001.

Index