Introduction to Java

Carlos Kavka
Departamento de Informática
Universidad Nacional de San Luis
San Luis, Argentina
email: ckavka@unsl.edu.ar

Seventh College on Microprocessor-based Real-Time Systems in Physics
The Abdus Salam ICTP, Trieste, Italia
October 28 – November 22, 2002

Contents

Chapter 2

Introduction to Java
by Carlos Kavka

2.1 Introduction

Java is a very powerful language that has generated a lot of interest in the last 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 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 electronic devices. Sun has released four versions of the language, and provides the JDK (Java Development Kit) freely through its web site. This has certainly contributed to the popularity of the language.

These lecture notes assume that you have some familiarity with C. In fact, usually Java can be learned 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, and in particular, we will not be covering applets and interface design.

All examples used in these notes are available for experimentation. In fact, it is a good complement of this lecture notes, the time you can spend executing and modifying the examples in order to fully understand the concepts involved.

2.2 The Java platform

Java programs are compiled to 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 in its own, with its instruction set and memory areas. A Java compiled program can be executed in any system that has a JVM.

The main advantage of this approach is, of course, portability, because the same Java compiled program can be executed in any computer that has a JVM. The price to pay is a slower execution.

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

2.3 A first example

This section presents a small example, the usual Hello World application. The application 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
  }
}
```

The application involves a lot of concepts. We will just be considering the main aspects now. The details will be covered in the next sections.

The application consists of only one class. Its name is `HelloWorld` and it has to be defined in the file `HelloWorld.java`, i. e., 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 it is shown in the example:

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

The 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 can be used to access the standard output stream. The method `println` is called in order to print the string passed as an argument in 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 are introduced by the characters `//` and multi-line comments are defined between the characters `/*` and `*/` like in C.

There exists a third class 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 defined in these comments, and they are interpreted by `javadoc` in order to automatically generate documentation that can be seen with a web browser.

2.4 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 of the class that is going to be defined, with the extension `.java`. The capitalization of the words will be recognized by the compiler. One possibility is the use of the `emacs` editor:

```
# emacs HelloWorld.java
```

Compilation : The Java compiler 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 application 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 generate automatically 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 2.1.

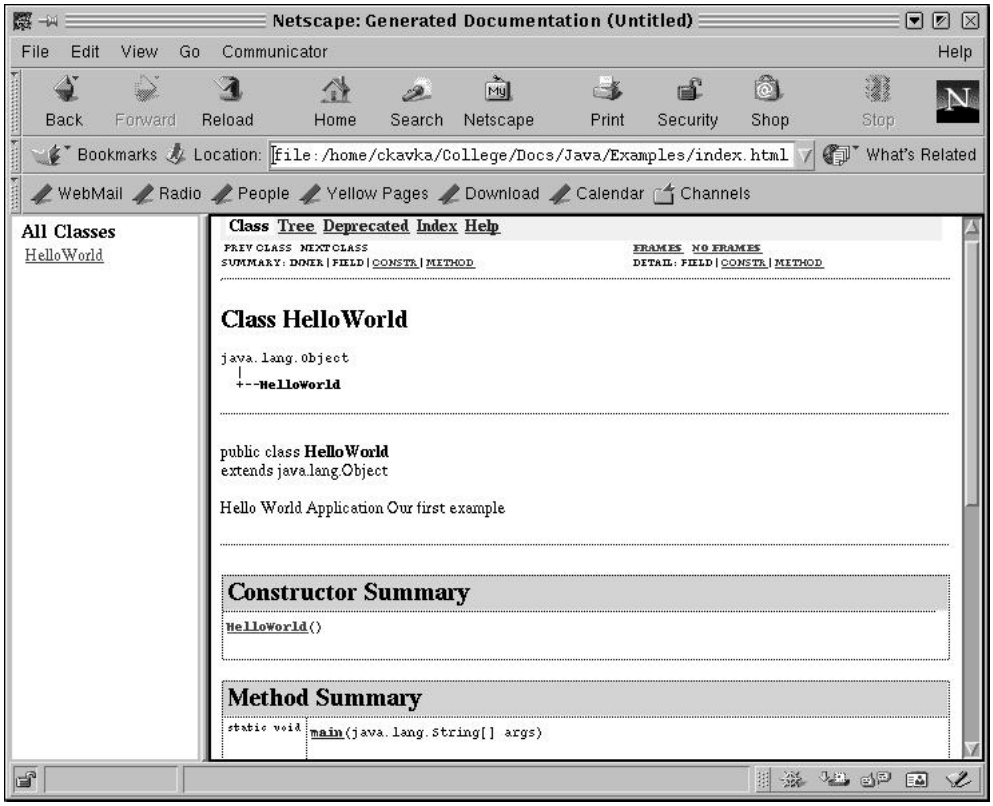

Figure 2.1: Documentation generated by javadoc

2.5 Basic types

Java provides ten primitive types: four types of integers, two types of floating point numbers, characters, booleans, the special type `void` and strings.

The next table presents for some types the minimum and maximum value and the size of the objects of these types:

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	1.4E-45	3.45E38
double	64 bits	4.9E-324	1.7E308
char	16 bits	unicode 0	unicode $2^{16}-1$

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:

```
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 `character` 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.

2.6 Variables

The variables can be declared 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;
double f = 0.33;
char c = 'a';
String s = "abcd";

x = 55;
```

2.7 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
int y = 0x3ef;            // hexadecimal
int z = 0772;             // octal
long m = 240395922L;      // long
```

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 with 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
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
```

2.8 Constants

The declaration for constants is very similar to the declaration of variables. It has to include the word `final` in front. The specification of the initial value is compulsory, as it is 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.

2.9 Expressions

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

2.9.1 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 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.

The shorthand operators composed of the assignment operator and a binary operator are also present. 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
```

2.9.2 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
```

2.9.3 Bit level operators

Java provides a set of operators that can manipulate bits directly. Some operators do boolean algebra on bits: *and* (&), *or* (|) and *not* (~), and others 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 in 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 lower bits side. 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 higher bits side. 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 higher bits side, 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);   // 11111111111111111111111111100111
 }
}
```

The example shows that shorthand operators are also available with the composition of the assignment operator and the binary boolean operators (& and |). The comments specify the binary representation of the values obtained in each sentence. The last two sentences 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);//00011111111111111111111111111110
  System.out.println(x >>>3);//11111111111111111111111111111110
 }
}
```

2.9.4 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
```

2.9.5 String operators

Java provides a complete set of operation 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 one with the characters from both arguments.

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

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);
```

2.9.6 Casting

Java performs a automatic type conversion in the values when there is no risk for data to be lost. 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 a long
    double d = 3.45;
```

```
    float f = (float)d;      // d is a double
  }
}
```

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

2.10 Control structures

Java provides the same set of control structures as in C. The main difference is that the value used in the conditional expressions must be a boolean value, and cannot be an integer. The structures are detailed in the next subsections.

2.10.1 The if control statement

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 sentence or by a compound statement consisting of a set of sentences 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("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
```

2.10.2 Iteration control statements

Java provides the standard `while` and `do-while` iteration control statements. 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 a third type of 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 does the same 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.

2.10.3 break and continue

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

The next example uses `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
3
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 out the execution of the loop when the counter reaches the value 8, causing the control to be transfered to the last sentence of the program.

`break` and `continue` can be used combined with a label, but we will be covering this use in this lecture notes.

2.10.4 Switch control statement

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

```
switch (integral-expression) {
  case integral-value: statement; [break;]
  case integral-value: statement; [break;]
  case integral-value: statement; [break;]

  [default: statement;]
}
```

The square brackets surrounds 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 next statements 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
```

2.11 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 arrays. 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 consists of the use of 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 the first array in this section 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 get the length of the arrays. The next application shows examples on 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 of a
    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;

    // 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]=1.0
d[3]=0.33333334
d[5]=0.2
d[7]=0.14285715
d[9]=0.11111111
```

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];
```

2.12 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 get 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]l);
  }
}
```

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 the class 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 the class `System`. It can be used to terminate the execution of the application.

2.13 Classes

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

```
class Book {

}
```

declares a class called `Book`. New instances of the class can be created with `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 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 class as follows:

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

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();
    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
```

2.13.1 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 initialize all fields of an instance of `Book` with the values passed as arguments:

```
/**
 * Example with books Application (version 2)
 */

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

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

class ExampleBooks2 {

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

    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 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 it was called in the example `ExampleBooks` before, without arguments:

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

This default constructor is only available when no 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 to identify which constructor is called when instances are created:

```
/**
 * Example with books Application (version 3)
 */

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
```

2.13.2 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 type of the return value. It is called by using the dot notation also.

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

```
/**
 * Example with books Application (version 4)
 */

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` corresponds 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)`, that can be used to get the length of a string, and the character in a specified position into the string.

The next example defines an array of books, and initializes it with data from three real 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.
```

2.13.3 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 if the two objects passed as arguments are the same object, 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
```

2.13.4 Static data members

The class is used as a model to create instances. Every instance of a `Book` has four data members (`title`, `author`, `numberOfPages` and `ISBN`), and they can be used to store values in one instance independently of the values stored in other instances.

Static data members (or class variables) are fields 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 data member 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 data member 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:

```
/**
 * Example with books Application (version 8)
 */

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

  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`. 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 data member called `owner` that can be accessed with the methods through all instances of `Books`.

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

2.13.5 Static methods

With the same idea of the static data members, 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());
    System.out.println(Book.description());
  }
}
```

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.

This method can only access static variables. In this case, the only variable that can be accessed from the method `description` is the variable `owner`.

2.13.6 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 data members, 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.

2.13.7 Data members initialization

All data members in an object are guaranteed to have an initial value. There exists a default value for all primitive types, which is defined in the following table:

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

All references to objects gets 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 in 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
```

2.14 The keyword "this"

The keyword `this`, when used inside a method, refers to the receiver object. It has two main uses: it can be used to return a reference to the receiver object from a method and 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 `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 section 2.27.1.

2.15 An example: the complex number class

Let's suppose we want to define a complex number class that allows 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 number 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 printed also.

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 class complex 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 part 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 a result. They do not have to modify the receptor or the complex number passed as argument. They 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 `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 this 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;
}
```

We must be careful if we want to create one complex number as a copy of the other, since the next assignment expression 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 2.13.3). This means that if we increment `e`, then `a` will be incremented also.

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 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 section 2.27.3.

2.16 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" other class (called base class) by using the word `extends` followed by the name of the base class. The definition of the new class specifies the differences with the base class.

Let's suppose we want to extend the definition of the class `Book` we have defined before to be useful to store information on scientific books. We can add two data members 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 `ScientificBook` will have six data members, the four inherited from the base class `Book` and the two new defined data members: `title`, `author`, `numberOfPages`, `ISBN`, `area` and `proceeding`. Note that by default a scientific book is not a proceeding.

2.16.1 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);
    area = a;
  }
```

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 base class `Book`, it is not necessary to do it again here. The constructor of the base 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 add some data members and methods.

2.16.2 Methods

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.

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 data members. However, we can reuse the checking of the four data members from the class Book and just write the comparison for the new data members as follows:

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

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

In this way, this method equals redefine the method with the same name defined in the base class. However, the method equals defined in the base class is called as part of the definition of this method. When super is used to call base 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 data members 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 base class. It can be defined as:

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

New methods can be defined. For example, we can define methods to set the proceeding condition 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 section 2.27.2.

2.16.3 Instanceof and getClass methods

The method `instanceof` returns a boolean value indicating if an object is an instance of a specified class. The method `getClass` returns a string that contains the name of the class the object is instance of.

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 name of the class the receptor objects are instances of. 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 corresponds to methods defined for books.

2.17 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 in 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 `import` statement:

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

It is usual to specify just the name of the package, and not the name of the class, in order to tell the compiler that we are interested in the use of some classes defined in the 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:

```
Thu Nov 08 19:14:18 GMT-02:00 2001
```

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

```
package mypackage;
```

This must be the first non commented statement in our file. The classes defined in this file will 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.

There is a name convention, that we will not be covering here, in order to define packages that can be shared with the Java community.

2.18 Access control

It is possible to control the access to methods and variables from other classes with three so called modifiers: `public`, `private` and `protected`. There exists a default access which is the one we have been using in most examples, that allows full access from all classes that belong to the same package.

Full access means that it is possible to access data members 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 data member:

```
sb1.proceeding = true;
```

Usually we do not want direct access to a data member in order to guarantee encapsulation. In this case we can use the modifier `private`. This modifier guarantees that the data member 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 data member `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 the methods. 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 data members and methods from subclasses and from all classes in the same package.

2.19 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 see now 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 is 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 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 hardware. 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 next is the code of the class `IOBoard` defined as it was discussed above:

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

abstract class IOBoard {
  String name;
  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 data member defined in this class, the name is the value to be stored in the field defined in the base class. Note that the constructor calls the constructor of the base class through `super`.

The methods just print a message identifying themselves, and return, since we are not going to define communication code 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 data member defined in this class, the name is the value to be stored in the field defined in the base class. Note that the constructor calls the constructor of the base 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 base class are ignored.

2.20 Polymorphism

Polymorphism is an important property that our programs should have. We can say that there exists polymorphism when different objects can answer 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 answer 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 2.16.3). 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 just operate with the input output board instances stored in the array just by sending the corresponding messages, without considering the specific type of board. This is possible due to the polymorphism property.

2.21 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.

It is possible to extend this idea with the use of *interfaces*. 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 just represent constant values (being final and static), and the methods can just define a behavior, or exactly 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 provide at least an implementation for all methods defined in the interface.

In our example, if we are not interested in having a name for the boards, and a 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 other 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 we can get a similar effect with abstract classes and 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. Interfacing allows to implement a kind of multiple inheritance. An abstract class should be defined when there is data and/or methods to be defined in the class, and shared by all subclasses.

2.22 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 the following example shows:

```
/**
 * 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 happens 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 just in process the exception for index out of bounds for strings, we can do it in this 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 exists 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
```

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 the interest to not to catch even compulsory exceptions. We will see more examples in sections to follow.

2.23 Input Output

The input output system in Java is rather complicated. Unfortunately, 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 in the same way.

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

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

2.23.1 Byte oriented streams

There exists two classes that can be used for processing byte oriented streams: the class `FileOutputStream` than 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 `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 into 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 with the message `read` one after the other. This message returns the byte read or `-1` when the end of the 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 it is written in this way, it can be read without problems with 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 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.

2.23.2 Buffered byte oriented streams

In order to minimize the communication overhead, it is 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 it was explained in the previous section. The available messages are the same.

The next application shows how to write 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 is created by passing the output stream as 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 = f.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
```

2.23.3 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 following messages 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)

The next application stores into a data buffered byte oriented stream an integer that corresponds to the size of an array of doubles, then 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
```

2.23.4 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:

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

The first message can be used to write characters from a string from the position indicated by the first integer, as many as indicated by the second integer. The second message is similar, but the characters are written from an array of characters. The message `newLine` generates a newline in the output stream independently of the convention used in the current operating system.

The next application writes some characters of two strings into an 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());
    }
  }
}
```

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());
    }
  }
}
```

The output of the execution of the application is:

```
Hello World!
nice!
```

2.23.5 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 variable `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 quoted before, the methods can express their interest in not to catch some specific exceptions. This is done by using the word `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 the `try` and `catch` block was not defined.

2.24 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. The 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");
      }
    }
  }
}
```

The 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 then the thread goes 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, initialized with a different character each one. Then both of them are started as a thread, so the method run 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, so the output produced by them is intermixed.

2.24.1 The Producer and Consumer example

The producer and consumer problem is a standard example that illustrates concurrency, and the problems derived from it. The idea is that there exist two

processes that interact through a common buffer: the producer produces items that are stored into a buffer, and 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 consumer and producer 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();
  }
}
```

A buffer with 20 empty slots is created, 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.

The 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.

The 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, i. e., the one that will be executed as a thread, just removes data from the buffer, and prints it into 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 on 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;
  }
}
```

This implementation seems to be OK. However, it does not work. And it does not work for two reasons:

- Both methods `insert` and `delete` operates concurrently over the same structure. It is necessary to define a critical region, or in other words, both

methods cannot be executed concurrently. If one thread is inserting data, the other must wait till the first one finish, and vice-versa.

- The `insert` method does not check if there is at least one slot free in the buffer, and the `delete` method does not check if there is at least one piece of data available in the buffer.

The next subsections will cover these problems and their solutions.

2.24.2 synchronized methods

In Java it is possible to define synchronized methods. These methods are not allowed to be executed concurrently on 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:

```
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;
}
```

2.24.3 wait and notify

Subclasses of `Thread` can send the 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 in the buffer has to put itself to sleep when there is no empty slots in a buffer. The thread that is going to remove a value from an empty buffer has to put itself to sleep.

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

The correct code for the implementation of both methods follows:

```
  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();
  }

  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;
  }
```

Note that it is necessary to catch an exception when `wait` is used.
The listing of the `Buffer` class is provided in section 2.27.4.

2.25 JAR files

When we were compiling the example `ProducerConsumer`, 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 application it is necessary to copy the 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 in pairs (*key*,*value*). 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 next file names 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. Its content is:

```
Manifest-Version: 1.0
Main-Class: ProducerConsumer
Created-By: 1.4.0 (Sun Microsystems Inc.)
```

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

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

It is possible to extract an update also the contents of a JAR file. Please refer to the documentation for examples.

2.26 Java Native Interface

The Java Native Interface (JNI) allows to call functions written in other languages from Java code. This is useful when a "100%" Java solution is not enough to solve the problem, may be due to some of the following situations:

- There is a big amount of existing fully tested code implemented in other language, and it makes no sense to rewrite it again in Java.
- The application requires access to system features or devices, which currently are not supported by Java.
- The execution speed is essential.

Even if the JNI can help in these cases, it must be considered that its use has two main disadvantages: portability is lost and security is reduced. Since the native code is compiled for a specific platform, now the application will run just on this platform. It is still possible to provide different versions of the native methods for different platforms, but the benefits of portability are significatively reduced. Also code written in other languages usually does not have the same security level of Java code.

JNI is not an easy topic and tasks like working with Java objects from other languages are quite complicated. The next subsections will introduce JNI by using C as the language for native code. Since JNI is complex, the topics will be presented with examples. However, most of the examples will just illustrate the use of JNI features, without being the *best* examples, in the sense, that they will represent tasks that difficultly would be selected to be implemented as native methods. It was preferred to lose on semantics in order to improve the simplicity of the examples.

2.26.1 The definition of native methods

To develop a Java application that uses native methods we have to follow the next steps:

1. Define the prototype of the function in the Java class with the keyword `native`.
2. Compile the java application.
3. Generate a C header file by using the utility program `javah`.
4. Define the C function by following the JNI guidelines.
5. Compile the C function (and its header file) to a shared library.

The next example shows the definition of a simple method (static with no arguments and no return value) in the class `Book` in order to illustrate the process. The method `printDescription` is defined as a native method (step 1) as follows:

```
class Book {
  .......
  public native static void printDescription();
}
```

Note that the definition of the method is similar to the definition of a standard method, with just two differences: there is no body, and the keyword `native` is used. This keyword tells the compiler that the method is defined externally and in other language.

The class `Book` has to be compiled as usual (step 2):

```
# javac Book.java
```

The utility program `javah` has to be used (step 3) to generate the header file as follows:

```
# javah Book
```

The execution of `javah` produces a header file with the same name of the class and the extension `.h`. In this case it is called `Book.h`. In our example, the contents of this file are:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Book */

#ifndef _Included_Book
```

```
#define _Included_Book
#ifdef __cplusplus
extern "C" {
#endif
/* Inaccessible static: owner */
/*
 * Class:     Book
 * Method:    printDescription
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Book_printDescription
  (JNIEnv *, jclass);
```

This file does not have to be edited, since it is generated by `javah` and it will be regenerated again every time this utility program is executed. The two lines at the end contains the prototype of the C function that we want to define:

```
JNIEXPORT void JNICALL Java_Book_printDescription
  (JNIEnv *, jclass);
```

The name of the function is `Java_Book_printDescription`. It must start with the string "Java", followed by the name of the class and the name of the method, separated by underscores. The type of the return value is `void` and the function takes two arguments, even if the original Java method does not have arguments at all. The first argument is a pointer to a table of function pointers that will be used to access Java functionality from C. The second argument identifies the class to which the message was sent. We will not be using these arguments in this first example. The macros `JNIEXPORT` and `JNICALL` define implementation details for shared libraries, and we can safely ignore the way in which they are implemented.

The C function can be defined (step 4) in the file `Book.c` by using the prototype that was generated by `javah` in the header file as follows:

```
#include <Book.h>
#include <stdio.h>

JNIEXPORT void JNICALL Java_Book_printDescription
(JNIEnv *env,jclass cl) {
    printf("I am a book!\n");
}
```

Note that this is a standard C file, and it is possible to use all C functionality. In this first example, the function just prints a string in the standard output.

The function has to be compiled as a shared library (step 5). This process is certainly dependent on the C compiler and the paths where the JNI header files

have to be searched for. A usual command that can be used to get the shared library from the file `Book.c` with the GNU C compiler is:

```
# gcc -shared -o libBook.so Book.c
```

The flag `-shared` indicates that `Book.c` has to be compiled into a shared library with name `libBook.so`, as indicated by the option `-o` [1].

It is also necessary to instruct the class `Book` to load the shared library that contains the definition of the native methods. This is usually done with a static initialization block in the class `Book`, as follows:

```
class Book {
  .......
  static {
     System.loadLibrary("Book");
  }
}
```

The method `loadLibrary` instructs the class Book to load the shared library. The way in which the name of the library is specified is system dependent.

The following is a test application that we can use to check the native method we have defined:

```
class TestBook1 {
  public static void main(String[] args) {
    Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b1.printDescription();
  }
}
```

The output of the execution of the application is [2]:

```
I am a book!
```

Not too much, but we manage to implement a method in the class `Book` with a C function.

[1] If the compiler cannot find JNI specific header files, it is possible that you have to add to the include search paths the directories `include` and `include/linux` from the Java SDK installation, or specify them in the command line with the option `-I`.

[2] If the program stops with an error message that indicates that the shared library cannot be found, verify that the shell variable `LD_LIBRARY_PATH` contains the directory in which the shared library is located.

2.26.2 Numeric parameters and return values

Other point that makes native functions complicated is the fact that the standard types can be different in Java and in C. All standard types in Java are guaranteed to have a specific size, which is not the case with C. For example, an `int` in C has what is called the *natural* size, which can be of course platform dependent. For this reason, the following types are defined in JNI and has to be used for passing data between Java and C programs:

Java	C	size
boolean	jboolean	1
byte	jbyte	1
char	jchar	2
short	jshort	2
int	jint	4
long	jlong	8
float	jfloat	4
double	jdouble	8

The values for the `jboolean` type are `JNI_TRUE` and `JNI_FALSE`, and are equivalent to `1` and `0` respectively.

The following example shows the definition, in the class `Book`, of the native method `computeWeight` that can be used to compute the weight of a book:

```
public native static float computeWeight(int numPages,
        float paperWeight);
```

It has two arguments: an integer that corresponds to the number of pages of the book, and a float that corresponds to the weight of the paper. It returns a floating point number that corresponds to the weight of the book. Note that the method is static and it does not access the fields of a particular instance of a book.

The C function can be defined as follows:

```
JNIEXPORT jfloat JNICALL Java_Book_computeWeight
(JNIEnv *env, jclass cl, jint numPages, jfloat paperWeight) {
  jfloat pageWeight;

  // assume an A4 page
  pageWeight = (0.23 * 0.297) * paperWeight;

  // compute the weight
  return (numPages / 2.0) * pageWeight;
}
```

The two new parameters are defined after the environment and the class parameters. They have to be defined with types `jint` and `jfloat` respectively. The return type of the functions is `jfloat`. The body of the function just computes the weight by assuming a standard size of paper and returns the result. Note that `jfloat` and `jint` values can be combined with `float` and `int` values. It is necessary just to take care on their respective size in order not to lose precision.

The following application presents an example of the use of this method:

```
class TestBook2 {
  public static void main(String[] args) {
    float weight = Book.computeWeight(1129,80);
      System.out.println("Weight: " + weight);
  }
}
```

The output of the execution of the application is:

```
Weight: 3084.8796
```

2.26.3 Using strings

Things are also difficult when strings have to be shared between Java methods and C functions, since implementation of strings is quite different in both languages: in C a string is a null terminated sequence of 1-byte characters and in Java a string is a sequence of 2-bytes UNICODE characters.

The following example shows the definition, in the class `Book`, of the native method `printTitle` that receives a string as an argument. The method just prints the string in standard output.

```
public native static void printTitle(String title);
```

The C function can be defined as follows:

```
JNIEXPORT void JNICALL Java_Book_printTitle
(JNIEnv *env, jclass cl, jstring title) {

  char *str = (*env)->GetStringUTFChars(env,title,NULL);
  printf(str);
  (*env)->ReleaseStringUTFChars(env,title, str);
  printf("\n");
}
```

The argument `title` is defined of type `jstring`, which is the Java string type defined by JNI to be used in native methods. The function converts the Java string `title` to a normal C string and stores it in the variable `str`, prints it by

calling the standard `printf` function and releases the memory allocated to the string.

The function used to convert the string is `GetStringUTFChars` and it receives as arguments the environment and the Java string [1]. The function used to release the memory is `ReleaseStringUTFChars` and it receives as arguments the environment, the Java string and the C string. The use of this function is compulsory to avoid the generation of garbage (memory that is considered used but is not accessible from the program).

The environment pointer `env` points to a table of pointers to functions. It is used as a *hook* to access Java functions from the native methods. All Java functions will be accessed in this way.

The following application presents an example of the use of this method:

```
class TestBook3 {
  public static void main(String[] args) {
    Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    Book.printTitle(b1.title);
  }
}
```

The output of the execution of the application is:

```
Thinking in Java
```

The following example shows the definition, in the class `Book`, of the native method `getDescription` that receives no arguments and returns a string:

```
public native static String getDescription();
```

The C function can be defined as follows:

```
JNIEXPORT jstring JNICALL Java_Book_getDescription
(JNIEnv *env, jclass cl) {
  jstring jstr;
  char desc[] = "I am a book!";

  jstr = (*env)->NewStringUTF(env,desc);
  return jstr;
}
```

It has no extra arguments and returns a value of type `jstring`. The function `NewStringUTF` converts the C string `desc` into a Java string, which is the return value of the function.

The following application presents an example of the use of this method:

[1] The third argument will be always `NULL` in our examples. Refer to the documentation for more details if necessary.

```
class TestBook4 {
  public static void main(String[] args) {
    Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    String str = b1.getDescription();
    System.out.println(str);
  }
}
```

The output of the execution of the application is:

```
I am a book!
```

2.26.4 Using non static methods and non static fields

All native methods implemented till now were static. In this section we will see how to implement a non static method that access fields from an object. We will implement a method that can be used to increment by a certain amount, the number of pages in a book. In pure Java, it can be implemented as follows:

```
public void incrementNumberOfPages(int amount) {
  numberOfPages += amount;
}
```

The following application presents an example of the use of this method:

```
class TestBook5 {
  public static void main(String[] args) {
  Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
  b1.incrementNumberOfPages(36);
  System.out.println(b1.numberOfPages);
  }
}
```

The output of the execution of the application is:

```
1165
```

The method could be rewritten as a native method with the following definition:

```
public native void incrementNumberOfPages(int amount);
```

The C function can be defined as follows:

```
JNIEXPORT void JNICALL Java_Book_incrementNumberOfPages
(JNIEnv *env, jobject obj_this, jint amount) {

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the field id
  jfieldID id_numberOfPages = (*env)->GetFieldID(env,
        class_Book,"numberOfPages","I");

  // get the field value
  jint numberOfPages = (*env)->GetIntField(env,obj_this
        id_numberOfPages);

  // increment the value
  numberOfPages += amount;

  // set the field value
  (*env)->SetIntField(env,obj_this,id_numberOfPages,
        numberOfPages);
}
```

The function now contains three parameters: the first one is the usual environment and the third one is the integer parameter that corresponds to the amount value. The second parameter in a non static method is a reference to the object that is the receptor of the message, or in other words, a reference to `this`. It means that static methods get a reference to the class, and non static methods a reference to `this`.

In order to access the fields of an object it is necessary to get first a reference to the class, and then a field identifier. The reference of the class in this example is obtained by executing:

```
jclass class_Book = (*env)->GetObjectClass(env,obj_this);
```

The function `GetObjectClass` returns a reference to the class of the object passed as an argument. In this example, the object passed as an argument is referenced by `this`. After the execution of this statement, the variable `class_Book` of type `jclass` will contain a reference to the class `Book`.

The field identifier is obtained by executing:

```
jfieldID id_numberOfPages = (*env)->GetFieldID(env,
        class_Book,"numberOfPages","I");
```

The function `GetFieldID` returns a field identifier of the object field described by the arguments. The first argument is the environment, the second one is the

class of the object, the third one a string that corresponds to the name, and the fourth one a string that identifies the type of the field. In this case, the type is "I" that corresponds to an integer. After the execution of this statement, the variable `id_numberOfPages` of type `jfieldID` will contain a field identifier for `Book.numberOfPages`.

The field value is obtained by executing:

```
jint numberOfPages = (*env)->GetIntField(env,obj_this
      id_numberOfPages);
```

The function `GetIntField` returns the value of the integer field described by its arguments. The first one is the environment, the second one a reference to the object (`this` in this case) and the third one is the field identifier. The returned value is of type `jint`.

After incrementing the number of pages, the new value can be set by executing:

```
(*env)->SetIntField(env,obj_this,id_numberOfPages,
      numberOfPages);
```

The function `SetIntField` sets the integer field identified by its arguments with a specific value. The first argument is the environment, the second one a reference to the object (`this` in this case), the third one the field identifier and the fourth one the new value (of type `jint`).

As it was seen before, a so called *signature* has to be used in order to specify the type of a field. The encoding is described in the following table:

Signature	Type name
B	byte
C	char
D	double
F	float
I	int
J	long
L*classname*;	a class *classname*
S	short
V	void
Z	boolean

In this example, the functions `GetIntField` and `SetIntField` were used respectively to get an integer value and to set an integer value. Of course, other functions must be used for fields with different types. A set of functions `Get--Field` and `Set--Field` are provided by JNI, where `--` can be `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double` or `Object`.

Just to have other example, the next lines shows the definition, in the class `Book`, of the native method `getWeight` that receives an integer argument that corresponds to the weight of the paper and returns the weight of the book:

```
public native float getWeight(float paperWeight);
```

The C function can be defined as follows:

```
JNIEXPORT jfloat JNICALL Java_Book_getWeight
(JNIEnv *env, jobject obj_this, jfloat paperWeight) {

    // get the class
    jclass class_Book = (*env)->GetObjectClass(env,obj_this);

    // get the field id
    jfieldID id_numberOfPages = (*env)->GetFieldID(env,
        class_Book,"numberOfPages","I");

    // get the field value
    jint numberOfPages = (*env)->GetIntField(env,obj_this,
        id_numberOfPages);

    // assume an A4 page
    jfloat pageWeight = (0.23 * 0.297) * paperWeight;

    // compute the weight
    return (numberOfPages / 2.0) * pageWeight;
}
```

The following application presents an example of the use of this method:

```
class TestBook6 {
  public static void main(String[] args) {
    Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    float weight = b1.getWeight(80.0f);
    System.out.println("Weight: " + weight);
  }
}
```

The output of the execution of the application is:

```
Weight: 3084.8796
```

2.26.5 Accessing static fields

JNI provides a set of functions to access and modify static fields in a similar way as with non static fields. The next example shows the definition, in the class `Book`, of the native method `updateOwner` that receives a string as an argument and returns a string. The method updates the value of the static field `owner` with the new string passed as an argument and returns the old value.

```
public native String updateOwner(String newOwner);
```

The C function can be defined as follows:

```
JNIEXPORT jstring JNICALL Java_Book_updateOwner
(JNIEnv *env, jobject obj_this, jstring newOwner) {

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the field id
  jfieldID id_owner = (*env)->GetStaticFieldID(env,class_Book,
        "owner","Ljava/lang/String;");

  // get the field value
  jstring owner = (*env)->GetStaticObjectField(env,obj_this,
        id_owner);

  // print old owner
  char *str = (*env)->GetStringUTFChars(env,owner,NULL);
  printf("Old owner: %s\n",str);
  (*env)->ReleaseStringUTFChars(env,owner, str);

  // set the new owner
  (*env)->SetStaticObjectField(env,obj_this,id_owner,newOwner);

  // return it
  return newOwner;
}
```

The function receives the string as the argument `newOwner` of type `jstring` and returns a Java string. In a similar way, in order to access the static fields of an object it is necessary to get first a reference to the class, and then a field identifier. The reference of the class is obtained by executing:

```
jclass class_Book = (*env)->GetObjectClass(env,obj_this);
```

The field identifier is obtained by executing:

```
jfieldID id_owner = (*env)->GetStaticFieldID(env,class_Book,
        "owner","Ljava/lang/String;");
```

The function `GetStaticFieldID` returns a field identifier of the static object field described by the arguments. The first argument is the environment, the second one is the class of the object, the third one a string that corresponds to the name, and the fourth one a string that identifies the type of the field. In this case, the type is "Ljava/lang/String;", indicating that the field contains an instance of the class `Java.lang.String`. Note that the name of the class is specified by a concatenation of packages and class names, separated by slashes.

The field value is obtained by executing:

```
jstring owner = (*env)->GetStaticObjectField(env,obj_this,
        id_owner);
```

The function `GetStaticObjectField` returns a reference to the object stored in the static field described by its arguments. The first one is the environment, the second one a reference to the object (`this` in this case) and the third one is the field identifier. The returned value is of type `jobject`, which can be safely assigned to a `jstring` variable.

In order to print the Java string it is necessary to convert it to a normal C string. This is done in the function as follows:

```
char *str = (*env)->GetStringUTFChars(env,owner,NULL);
printf("Old owner: %s\n",str);
(*env)->ReleaseStringUTFChars(env,owner, str);
```

The new value is set by using the `SetStaticObjectField` function as follows:

```
(*env)->SetStaticObjectField(env,obj_this,id_owner,newOwner);
```

The following application presents an example of the use of this method:

```
class TestBook7 {
  public static void main(String[] args) {
    Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b1.setOwner("Carlos Kavka");
    String newOwner = b1.updateOwner("Rinus Verkerk");
    System.out.println("The new owner is: " + newOwner);
    System.out.println("Just to be sure: " + b1.getOwner());
  }
}
```

The output of the execution of the application is:

```
Old owner: Carlos Kavka
The new owner is: Rinus Verkerk
Just to be sure: Rinus Verkerk
```

The JNI functions `GetStaticObjectField` and `SetStaticObjectField` were used respectively to get an object value and to set an Object value. Of course, other functions must be used for fields with different types. JNI provides a set of functions `GetStatic--Field` and `SetStatic--Field`, where `--` can be `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double` or `Object`.

2.26.6 Calling non static Java methods from C

This section illustrates the procedure to call non static Java methods from a native function in C. The next example shows the definition, in the class `Book`, of the native method `printInitials` that receives no arguments and returns nothing. The method just prints the initials of the authors' name by calling the method `getInitials`.

```
public native void printInitials();
```

The C function can be defined as follows:

```
JNIEXPORT void JNICALL Java_Book_printInitials
(JNIEnv *env, jobject obj_this) {

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the method id
  jmethodID id_getInitials = (*env)->GetMethodID(env,
        class_Book,"getInitials","()Ljava/lang/String;");

  // call the method
  jstring in = (*env)->CallObjectMethod(env,obj_this,
        id_getInitials);

  // print initials
  char *str = (*env)->GetStringUTFChars(env,in,NULL);
  printf("Initials: %s\n",str);
  (*env)->ReleaseStringUTFChars(env,in, str);
}
```

After getting the class as usual, it is necessary to get a method identifier for the method to be called. This is done as follows:

```
jmethodID id_getInitials = (*env)->GetMethodID(env,
        class_Book,"getInitials","()Ljava/lang/String;");
```

The JNI function `GetMethodID` returns a method identifier for the non static method described by the arguments. The first argument is the environment, the second one is the class of the object, the third one a string that corresponds to the name, and the fourth one a string that identifies the signature or prototype of the method. In this case, the type is "()Ljava/lang/String;", indicating that the method has no arguments and the return value is a `Java.lang.String`.

The method `getInitials` is called by executing:

```
jstring in = (*env)->CallObjectMethod(env,obj_this,
        id_getInitials);
```

The function `CallObjectMethod` sends the message specified by the method identifier to a specific object. The method is specified by the third argument and the object by the second argument. This function calls the method, and then returns a reference to an object, which in this case, will be a reference to the Java string returned by the method `getInitials`.

The string is then printed as usual:

```
  char *str = (*env)->GetStringUTFChars(env,in,NULL);
  printf("Initials: %s\n",str);
  (*env)->ReleaseStringUTFChars(env,in, str);
```

The following application presents an example of the use of this method:

```
class TestBook8 {
  public static void main(String[] args) {
    Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b1.printInitials();
  }
}
```

The output of the execution of the application is:

```
B.E.
```

In this example, the function `CallObjectMethod` was used to call a method that returns an instance of an Object. Of course, other functions must be used for methods that returns different types. A set of functions `Call--Method` are

provided by JNI, where `--` can be `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double` or `Object`.

The signature or prototype of a method is defined by specifying between parenthesis the signature for the arguments, and after the parenthesis the signature of the return value [1]. Note that no separator is used. The next table shows some examples:

Method	Signature
int fun(int x,int y);	(II)I
boolean fun2(int x,char c,double f);	(ICD)Z
double fun3(String s,int i,float f);	(Ljava/lang/String;IF)D
String fun4(int i,int j,Book b);	(IILBook;)Ljava/lang/String;

Just to have other example, the next lines shows the definition, in the class `Book`, of the native method `lighter` that receives a reference to a book as an argument and returns a boolean value that indicates if the receptor book is lighter than the book passed as an argument:

```
public native boolean lighter(Book other);
```

The C function can be defined as follows:

```
JNIEXPORT jboolean JNICALL Java_Book_lighter
(JNIEnv *env, jobject obj_this, jobject other) {

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the method id
  jmethodID id_getWeight = (*env)->GetMethodID(env,
        class_Book,"getWeight","(F)F");

    // call the method twice
    jfloat w1 = (*env)->CallFloatMethod(env,obj_this,
        id_getWeight,80.0);
    jfloat w2 = (*env)->CallFloatMethod(env,other,
        id_getWeight,80.0);

    if (w1 < w2)
      return JNI_TRUE;
    else
```

[1] The utility command `javap` with option `-s` can be used to get the signature of all methods from a specific class.

```
      return JNI_FALSE;
}
```

The native function calls the method `getWeight` on both instances: the receptor object (referenced by `this`) and the object passed as argument (referenced by `other`). Note the use of the function `CallFloatMethod` since the return value of the method is a float.

The following application presents an example of the use of this method:

```
class TestBook9 {
 public static void main(String[] args) {
  Book b1=new Book("Thinking in Java","Bruce Eckel",1129);
  Book b2=new Book("Java in a nutshell","David Flanagan",353);
  if (b1.lighter(b2))
    System.out.println("Lighter");
  else
    System.out.println("Not lighter");
 }
}
```

The output of the execution of the application is:

```
Not lighter
```

2.26.7 Calling static Java methods from C

Of course static Java methods can be called from Java. The next example shows the definition, in the class `Book`, of the native method `printDescription2` that receives no arguments and returns nothing. The method just prints the description of the book by calling the static method `getDescription`.

```
public native static void printDescription2();
```

The C function can be defined as follows:

```
JNIEXPORT void JNICALL Java_Book_printDescription2
(JNIEnv *env, jclass cl) {

  // get the method id
  jmethodID id_desc = (*env)->GetStaticMethodID(env,cl,
        "printDescription","()V");

  // call it
  (*env)->CallStaticVoidMethod(env,cl,id_desc);

}
```

It is necessary to get a method identifier for the method to be called. This is done as follows:

```
jmethodID id_desc = (*env)->GetStaticMethodID(env,cl,
        "printDescription","()V");
```

The function `GetStaticMethodID` returns a method identifier for the static method described by the arguments. The first argument is the environment, the second one is the class of the object, the third one a string that corresponds to the name, and the fourth one a string that identifies the signature or prototype of the method. In this case, the type is "()V", indicating that the method has no arguments and the return type is void.

The method `printDescription` is called by executing:

```
(*env)->CallStaticVoidMethod(env,cl,id_desc);
```

The JNI function `CallStaticVoidMethod` sends the message specified by the method identifier to a specific object. The method is specified by the third argument and the class by the second argument. Note that in our example, the method `printDescription` is also native.

The following application presents an example of the use of this method:

```
class TestBook10 {
  public static void main(String[] args) {
    Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b1.printDescription2();
  }
}
```

The output of the execution of the application is:

```
I am a book!
```

In this example, the function `CallStaticVoidMethod` was used to call a method that returns nothing (void). Of course, other functions must be used for methods that returns different types. A set of functions `CallStatic--Method` are provided by JNI, where `--` can be `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double` or `Object`.

2.26.8 Calling Java constructors from C

A native method can create Java objects by calling a constructor. The next example shows the definition, in the class `Book`, of the native method `createNewBook` that receives the title, the author and the number of pages and returns a new Book. The method just calls one of the constructors of the class `Book`.

```
public native static Book createNewBook(String tit,String aut,
        int num);
```

The C function can be defined as follows:

```
JNIEXPORT jobject JNICALL Java_Book_createNewBook
(JNIEnv *env, jclass cl, jstring tit, jstring aut, jint num) {

  // get the method id
  jmethodID id_constructor = (*env)->GetMethodID(env,cl,
        "<init>","(Ljava/lang/String;Ljava/lang/String;I)V");

  // build the object
  jobject obj_new = (*env)->NewObject(env,cl,id_constructor,
        tit,aut,num);

  // return it
  return obj_new;
}
```

The function `GetMethodID` is used to get the method identifier for the constructor:

```
jmethodID id_constructor = (*env)->GetMethodID(env,cl,
        "<init>","(Ljava/lang/String;Ljava/lang/String;I)V");
```

The name of the method is specified by the string "`<init>`", which is the internal name used in the Java Virtual Machine for the constructors. The string "(Ljava/lang/String;Ljava/lang/String;I)V" is the signature of the first constructor in the class `Book`.

The constructor is called as follows:

```
jobject obj_new = (*env)->NewObject(env,cl,id_constructor,
        tit,aut,num);
```

The function `NewObject` returns a new object created by calling the constructor specified by the third argument of the class, which in turn is specified by the second argument. The parameters for the constructor are passed starting from the fourth position.

The following application presents an example of the use of this method:

```
class TestBook11 {
  public static void main(String[] args) {
    Book b1 = Book.createNewBook("Java in a nutshell",
```

```
        "David Flanagan",353);
    System.out.println("Book: " + b1.title + " ," + b1.author
        + " ," + b1.numberOfPages);
  }
}
```

The output of the execution of the application is:

```
Book: Java in a nutshell ,David Flanagan ,353
```

2.26.9 Using arrays

It is possible to access Java arrays from native methods in C. JNI defines a set of types to be used in C that correspond to Java arrays. They are shown in the following table:

Java arrays	C arrays
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
int[]	jintArray
short[]	jshortArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
Object[]	jobjectArray

The next example shows the definition, in the class `Book`, of the native method `biggest` that receives an array of books as an argument and returns a reference to the biggest book. The method selects the biggest book by considering the number of pages of the books in the array.

```
public native static Book biggest(Book[] books);
```

The C function can be defined as follows:

```
JNIEXPORT jobject JNICALL Java_Book_biggest
  (JNIEnv *env, jclass cl, jobjectArray books) {

  // get the field id
  jfieldID id_numberOfPages = (*env)->GetFieldID(env,cl,
        "numberOfPages","I");

  // get the array length
```

```
  jsize length = (*env)->GetArrayLength(env,books);

  // traverse the array searching for the biggest book
  int i;
  int biggest = 0;
  int numPages = 0;

  for(i = 0;i < length;i++) {

    // get ith book
    jobject book = (*env)->GetObjectArrayElement(env,books,i);

    // get number of pages
    jint numberOfPages = (*env)->GetIntField(env,book,
        id_numberOfPages);

    // compare
    if (numberOfPages > numPages) {
      biggest = i;
      numPages = numberOfPages;
    }
  }

  // return it
  return (*env)->GetObjectArrayElement(env,books,biggest);
}
```

The parameter `books`, that receives the array of books is defined with type `jObjectArray`. The native function obtains the length of this array by executing:

```
jsize length = (*env)->GetArrayLength(env,books);
```

The function `GetArrayLength` returns a value of type `jsize` that corresponds to the length of the array. The parameters are the environment and the Java array.

The i-th object in the array is obtained as follows:

```
jobject book = (*env)->GetObjectArrayElement(env,books,i);
```

The function `GetObjectArrayElement` receives as arguments the environment, the array of objects and the position, and returns a reference to the object stored in this position in the array.

The number of pages of the book in the array can be get as usual with the function `GetIntField`:

```
jint numberOfPages = (*env)->GetIntField(env,book,
        id_numberOfPages);
```

The function returns a reference to the biggest book by using the function `GetObjectArrayElement`:

```
return (*env)->GetObjectArrayElement(env,books,biggest);
```

The following application presents an example of the use of this method:

```
class TestBook12 {
 public static void main(String[] args) {
  Book b1=new Book("Java in a nutshell","David Flanagan",353);
  Book b2=new Book("Thinking in Java","Bruce Eckel",1129);
  Book b3=new Book("Neural Networks, A Comprehensive Foundation",
       "Simon Haykin",696);

  Book[] books = {b1,b2,b3};

  Book biggest = Book.biggest(books);

  System.out.println("The biggest book is: " + biggest.title);
 }
}
```

The output of the execution of the application is:

```
The biggest book is: Thinking in Java
```

The elements in the array can also be modified by using the JNI function `SetObjectArrayElement`. It has one extra parameter that is the reference to be assigned into the specific array position.

These methods can be used to access the elements of a Java array of Objects. JNI provides a set of functions that can be used to convert other types of Java arrays into equivalent C arrays. In this way, the arrays can be processed by using the efficient C instructions. The next example defines the method `smaller`, that receives an array of integers as an argument and returns an integer value that indicates how many numbers in the array are smaller than the number of pages in the book. Note that it converts the Java array into a C array before processing:

```
public native int smaller(int[] numbers);
```

The C function can be defined as follows:

```
JNIEXPORT jint JNICALL Java_Book_smaller
  (JNIEnv *env, jobject obj_this, jintArray numbers) {

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the field id
  jfieldID id_numberOfPages = (*env)->GetFieldID(env,
        class_Book,"numberOfPages","I");

  // get the field value
  jint numberOfPages = (*env)->GetIntField(env,obj_this,
        id_numberOfPages);

  // get the size of the array
  jsize length = (*env)->GetArrayLength(env,numbers);

  // get the standard C array
  int *a = (*env)->GetIntArrayElements(env,numbers,NULL);

  // compute
  int i,count = 0;
  for(i = 0;i < length;i++) {
    if (a[i] < numberOfPages)
      count++;
  }

  // release the array
  (*env)->ReleaseIntArrayElements(env,numbers,a,0);

  // return the value
  return count;
}
```

The function `GetIntArrayElements` converts a Java array of `jint` values into a C array of `int` values:

```
int *a = (*env)->GetIntArrayElements(env,numbers,NULL);
```

The parameters of the function are the environment and the Java array [1]. After the assignment, the variable `a` can be used as a normal array. The memory area assigned dynamically to the array is released by calling the function

[1] The third argument will be always `NULL` in our examples. Refer to the documentation for more details if necessary.

`ReleaseIntArrayElements` with the environment, the original Java array and the C array as parameters [1]:

```
(*env)->ReleaseIntArrayElements(env,numbers,a,0);
```

The following application presents an example of the use of this method:

```
class TestBook13 {
  public static void main(String[] args) {
   Book b1=new Book("Java in a nutshell","David Flanagan",353);

   int[] numbers = {550,645,138,85,1022};

   int num = b1.smaller(numbers);

   System.out.println("Result: " + num);
  }
}
```

The output of the execution of the application is:

```
Result: 2
```

In order to process elements from arrays with different types, JNI provides a set of functions `Get--ArrayElements`, where `--` can be `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double` or `Object`.

The next example shows that the elements of an array can be modified from a native method. The example defines the method `getNumPages`, that receives an array of books and an array of integers as arguments and returns nothing. The method stores the number of pages of each book in the corresponding position of the array of integers:

```
public native static void getNumPages(Book[] books,
        int[] numbers);
```

The C function can be defined as follows:

```
JNIEXPORT void JNICALL Java_Book_getNumPages
  (JNIEnv *env, jclass cl, jobjectArray books,
  jintArray numbers) {

  // get the field id
```

[1] The fourth argument will be always `0` in our examples. Refer to the documentation for more details if necessary.

```
  jfieldID id_numberOfPages = (*env)->GetFieldID(env,cl,
        "numberOfPages","I");

  // get the array length
  jsize length = (*env)->GetArrayLength(env,books);

  // get the standard C array for numbers
  int *a = (*env)->GetIntArrayElements(env,numbers,NULL);

  // traverse the array
  int i;

  for(i = 0;i < length;i++) {

    // get ith book
    jobject book = (*env)->GetObjectArrayElement(env,books,i);

    // get number of pages
    jint numberOfPages = (*env)->GetIntField(env,book,
        id_numberOfPages);

    // store it into the numbers array
    a[i] = numberOfPages;
  }
  // release the array
  (*env)->ReleaseIntArrayElements(env,numbers,a,0);
}
```

The native function traverses the array of books and stores the number of pages of each book in the corresponding position of the array `a`. The function `ReleaseIntArrayElements` has to be called at the end, not only to release the memory used by the array, but also to perform the copy of the elements of the C array to the Java array.

The following application presents an example of the use of this method:

```
class TestBook14 {
 public static void main(String[] args) {
  Book b1=new Book("Java in a nutshell","David Flanagan",353);
  Book b2=new Book("Thinking in Java","Bruce Eckel",1129);
  Book b3=new Book("Neural Networks, A Comprehensive Foundation",
        "Simon Haykin",696);

  Book[] books = {b1,b2,b3};
```

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

  Book.getNumPages(books,numbers);

  for(int i = 0;i < 3;i++)
    System.out.println("Book " + i + ": " + numbers[i]);
 }
}
```

The output of the execution of the application is:

```
Book 0: 353
Book 1: 1129
Book 2: 696
```

2.26.10 Exceptions

It is possible to throw exceptions from a C native function. They are passed to Java methods for processing. The next example defines the method `writeFile`, that receives no arguments and returns nothing, but can throw an `IOException`. The native method just writes the title of the book in a text file, generating the exception if something goes wrong.

```
public native void writeFile() throws IOException;
```

The C function can be defined as follows:

```
JNIEXPORT void JNICALL Java_Book_writeFile
  (JNIEnv *env, jobject obj_this) {

  void throw_IO_exception(JNIEnv *);

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the field id
  jfieldID id_title = (*env)->GetFieldID(env,class_Book,
        "title","Ljava/lang/String;");

  // get the field value
  jstring title = (*env)->GetObjectField(env,obj_this,
        id_title);
  char *str = (*env)->GetStringUTFChars(env,title,NULL);
```

```
  FILE *fp;
  if ((fp = fopen("book.text","w")) == NULL) {
    throw_IO_exception(env);
    (*env)->ReleaseStringUTFChars(env,title,str);
    return;
  }

  if (fprintf(fp,"%s\n",str) != strlen(str)+1) {
    throw_IO_exception(env);
    (*env)->ReleaseStringUTFChars(env,title,str);
    return;
  }

  (*env)->ReleaseStringUTFChars(env,title,str);

  if (fclose(fp)) {
    throw_IO_exception(env);
    return;
  }
}
```

The native function gets the string value from the field title, opens a text file, writes the string into the file and closes it. The function `throw_IO_exception` is called if there is an error. This function is defined as follows:

```
void throw_IO_exception(JNIEnv *env) {

  // get IO Exception class
  jclass class_IO_exc = (*env)->FindClass(env,
        "java/io/IOException");

  // get constructor method
  jmethodID id_IO_exc = (*env)->GetMethodID(env,
        class_IO_exc,"<init>","()V");

  // create object
  jthrowable obj_exc = (*env)->NewObject(env,
        class_IO_exc,id_IO_exc);

  // throw it
  (*env)->Throw(env,obj_exc);
}
```

In order to throw an exception an object of the class `IOException` has to be created. This is done by calling the constructor of this class, in the same way as it was detailed in section 2.26.8. First, the class of the exception has to be obtained. This can be done with the function `FindClass`, to which a string identifying the class has to be passed:

```
jclass class_IO_exc = (*env)->FindClass(env,
        "java/io/IOException");
```

Then, the function has to get a method identifier for the constructor (method `<init>`):

```
jmethodID id_IO_exc = (*env)->GetMethodID(env,
        class_IO_exc,"<init>","()V");
```

The function `NewObject` is used to create the exception object:

```
jthrowable obj_exc = (*env)->NewObject(env,
        class_IO_exc,id_IO_exc);
```

The exception is thrown by using the function `Throw`:

```
(*env)->Throw(env,obj_exc);
```

Note that the execution of this function by itself does not force the native method termination. The native method has to perform a `return` after throwing the exception.

The following application presents an example of the use of this method:

```
class TestBook15 {
 public static void main(String[] args) {
  Book b1=new Book("Java in a nutshell","David Flanagan",353);

  try {
    b1.writeFile();
    System.out.println("File written");
  } catch (IOException e) {
    System.out.println("Input/Output error");
  }
 }
}
```

The output of the execution of the application if there are no errors is:

```
File written
```

The output if there are errors is:

```
Input/Output error
```

It is also possible to create the exception with a single function call:

```
(*env)->ThrowNew(env,(*env)->FindClass(env,
        "java/io/IOException"),"A problem");
```

The last argument is the string that identifies the reason of the exception. It can be obtained by a Java method with the usual `toString` message.

JNI also provides the function `ExceptionOcurred` that can be used to determine if a particular exception has been thrown and the function `ExceptionClear` to clear a pending exception.

2.27 Appendixes

2.27.1 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";
  }
}
```

2.27.2 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;
  }
}
```

2.27.3 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;
  }
}
```

2.27.4 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;
}
```

```
}
```

2.27.5 The native Book example

```
#include <Book.h>
#include <stdio.h>

/*
 * Class:     Book
 * Method:    printDescription
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Book_printDescription
(JNIEnv *env,jclass cl) {
    printf("I am a book!\n");
}

/*
 * Class:     Book
 * Method:    computeWeight
 * Signature: (IF)F
 */
JNIEXPORT jfloat JNICALL Java_Book_computeWeight
(JNIEnv *env, jclass cl, jint numPages, jfloat paperWeight) {
  jfloat pageWeight;

  // assume an A4 page
  pageWeight = (0.23 * 0.297) * paperWeight;

  // compute the weight
  return (numPages / 2.0) * pageWeight;
}

/*
 * Class:     Book
 * Method:    printTitle
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_Book_printTitle
(JNIEnv *env, jclass cl, jstring title) {

  char *str = (*env)->GetStringUTFChars(env,title,NULL);
  printf(str);
```

```
  (*env)->ReleaseStringUTFChars(env,title, str);
  printf("\n");
}

/*
 * Class:     Book
 * Method:    getDescription
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_Book_getDescription
(JNIEnv *env, jclass cl) {
  jstring jstr;
  char desc[] = "I am a book!";

  jstr = (*env)->NewStringUTF(env,desc);
  return jstr;
}

/*
 * Class:     Book
 * Method:    incrementNumberOfPages
 * Signature: (I)V
 */
JNIEXPORT void JNICALL Java_Book_incrementNumberOfPages
(JNIEnv *env, jobject obj_this, jint amount) {

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the field id
  jfieldID id_numberOfPages = (*env)->GetFieldID(env,
        class_Book,"numberOfPages","I");

  // get the field value
  jint numberOfPages = (*env)->GetIntField(env,obj_this
        id_numberOfPages);

  // increment the value
  numberOfPages += amount;

  // set the field value
  (*env)->SetIntField(env,obj_this,id_numberOfPages,
```

```
        numberOfPages);
}

/*
 * Class:     Book
 * Method:    getWeight
 * Signature: (F)F
 */
JNIEXPORT jfloat JNICALL Java_Book_getWeight
(JNIEnv *env, jobject obj_this, jfloat paperWeight) {

    // get the class
    jclass class_Book = (*env)->GetObjectClass(env,obj_this);

    // get the field id
    jfieldID id_numberOfPages = (*env)->GetFieldID(env,
        class_Book,"numberOfPages","I");

    // get the field value
    jint numberOfPages = (*env)->GetIntField(env,obj_this,
        id_numberOfPages);

    // assume an A4 page
    jfloat pageWeight = (0.23 * 0.297) * paperWeight;

    // compute the weight
    return (numberOfPages / 2.0) * pageWeight;
}

/*
 * Class:     Book
 * Method:    updateOwner
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_Book_updateOwner
(JNIEnv *env, jobject obj_this, jstring newOwner) {

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the field id
  jfieldID id_owner = (*env)->GetStaticFieldID(env,class_Book,
```

```
        "owner","Ljava/lang/String;");

  // get the field value
  jstring owner = (*env)->GetStaticObjectField(env,obj_this,
        id_owner);

  // print old owner
  char *str = (*env)->GetStringUTFChars(env,owner,NULL);
  printf("Old owner: %s\n",str);
  (*env)->ReleaseStringUTFChars(env,owner, str);

  // set the new owner
  (*env)->SetStaticObjectField(env,obj_this,id_owner,newOwner);

  // return it
  return newOwner;
}

/*
 * Class:     Book
 * Method:    printInitials
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Book_printInitials
(JNIEnv *env, jobject obj_this) {

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the method id
  jmethodID id_getInitials = (*env)->GetMethodID(env,
        class_Book,"getInitials","()Ljava/lang/String;");

  // call the method
  jstring in = (*env)->CallObjectMethod(env,obj_this,
        id_getInitials);

  // print initials
  char *str = (*env)->GetStringUTFChars(env,in,NULL);
  printf("Initials: %s\n",str);
  (*env)->ReleaseStringUTFChars(env,in, str);
}
```

```
/*
 * Class:     Book
 * Method:    lighter
 * Signature: (LBook;)Z
 */
JNIEXPORT jboolean JNICALL Java_Book_lighter
(JNIEnv *env, jobject obj_this, jobject other) {

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the method id
  jmethodID id_getWeight = (*env)->GetMethodID(env,
        class_Book,"getWeight","(F)F");

    // call the method twice
    jfloat w1 = (*env)->CallFloatMethod(env,obj_this,
        id_getWeight,80.0);
    jfloat w2 = (*env)->CallFloatMethod(env,other,
        id_getWeight,80.0);

    if (w1 < w2)
      return JNI_TRUE;
    else
      return JNI_FALSE;
}

/*
 * Class:     Book
 * Method:    printDescription2
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Book_printDescription2
(JNIEnv *env, jclass cl) {

  // get the method id
  jmethodID id_desc = (*env)->GetStaticMethodID(env,cl,
        "printDescription","()V");

  // call it
  (*env)->CallStaticVoidMethod(env,cl,id_desc);
```

```
}

/*
 * Class:     Book
 * Method:    createNewBook
 * Signature: (Ljava/lang/String;Ljava/lang/String;I)LBook;
 */
JNIEXPORT jobject JNICALL Java_Book_createNewBook
(JNIEnv *env, jclass cl, jstring tit, jstring aut, jint num) {

  // get the method id
  jmethodID id_constructor = (*env)->GetMethodID(env,cl,
        "<init>","(Ljava/lang/String;Ljava/lang/String;I)V");

  // build the object
  jobject obj_new = (*env)->NewObject(env,cl,id_constructor,
        tit,aut,num);

  // return it
  return obj_new;
}

/*
 * Class:     Book
 * Method:    biggest
 * Signature: ([LBook;)LBook;
 */
JNIEXPORT jobject JNICALL Java_Book_biggest
  (JNIEnv *env, jclass cl, jobjectArray books) {

  // get the field id
  jfieldID id_numberOfPages = (*env)->GetFieldID(env,cl,
        "numberOfPages","I");

  // get the array length
  jsize length = (*env)->GetArrayLength(env,books);

  // traverse the array searching for the biggest book
  int i;
  int biggest = 0;
  int numPages = 0;
```

```
  for(i = 0;i < length;i++) {

    // get ith book
    jobject book = (*env)->GetObjectArrayElement(env,books,i);

    // get number of pages
    jint numberOfPages = (*env)->GetIntField(env,book,
        id_numberOfPages);

    // compare
    if (numberOfPages > numPages) {
      biggest = i;
      numPages = numberOfPages;
    }
  }

  // return it
  return (*env)->GetObjectArrayElement(env,books,biggest);
}

/*
 * Class:     Book
 * Method:    smaller
 * Signature: ([I)I
 */
JNIEXPORT jint JNICALL Java_Book_smaller
  (JNIEnv *env, jobject obj_this, jintArray numbers) {

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the field id
  jfieldID id_numberOfPages = (*env)->GetFieldID(env,
        class_Book,"numberOfPages","I");

  // get the field value
  jint numberOfPages = (*env)->GetIntField(env,obj_this,
        id_numberOfPages);

  // get the size of the array
  jsize length = (*env)->GetArrayLength(env,numbers);
```

```
  // get the standard C array
  int *a = (*env)->GetIntArrayElements(env,numbers,NULL);

  // compute
  int i,count = 0;
  for(i = 0;i < length;i++) {
    if (a[i] < numberOfPages)
      count++;
  }

  // release the array
  (*env)->ReleaseIntArrayElements(env,numbers,a,0);

  // return the value
  return count;
}

/*
 * Class:     Book
 * Method:    getNumPages
 * Signature: ([LBook;[I)V
 */
JNIEXPORT void JNICALL Java_Book_getNumPages
  (JNIEnv *env, jclass cl, jobjectArray books,
  jintArray numbers) {

  // get the field id
  jfieldID id_numberOfPages = (*env)->GetFieldID(env,cl,
        "numberOfPages","I");

  // get the array length
  jsize length = (*env)->GetArrayLength(env,books);

  // get the standard C array for numbers
  int *a = (*env)->GetIntArrayElements(env,numbers,NULL);

  // traverse the array
  int i;

  for(i = 0;i < length;i++) {
```

```
    // get ith book
    jobject book = (*env)->GetObjectArrayElement(env,books,i);

    // get number of pages
    jint numberOfPages = (*env)->GetIntField(env,book,
        id_numberOfPages);

    // store it into the numbers array
    a[i] = numberOfPages;
  }
  // release the array
  (*env)->ReleaseIntArrayElements(env,numbers,a,0);
}

/*
 * Class:     Book
 * Method:    writeFile
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Book_writeFile
  (JNIEnv *env, jobject obj_this) {

  void throw_IO_exception(JNIEnv *);

  // get the class
  jclass class_Book = (*env)->GetObjectClass(env,obj_this);

  // get the field id
  jfieldID id_title = (*env)->GetFieldID(env,class_Book,
        "title","Ljava/lang/String;");

  // get the field value
  jstring title = (*env)->GetObjectField(env,obj_this,
        id_title);
  char *str = (*env)->GetStringUTFChars(env,title,NULL);

  FILE *fp;
  if ((fp = fopen("book.text","w")) == NULL) {
    throw_IO_exception(env);
    (*env)->ReleaseStringUTFChars(env,title,str);
    return;
  }
```

```
  if (fprintf(fp,"%s\n",str) != strlen(str)+1) {
    throw_IO_exception(env);
    (*env)->ReleaseStringUTFChars(env,title,str);
    return;
  }

  (*env)->ReleaseStringUTFChars(env,title,str);

  if (fclose(fp)) {
    throw_IO_exception(env);
    return;
  }
}
```

Bibliography

[1] Cay Horstmann and Gary Cornell (2001). *Core Java, Volume I - Fundamentals*. The Sun Microsystems Press.

[2] Cay Horstmann and Gary Cornell (2000). *Core Java, Volume II - Advanced Features*. The Sun Microsystems Press.

[3] Cay Hortsmann (2000). *Computing concepts with Java 2, essentials*. Second Edition. Wiley.

[4] Steven Holzner (2000). *Java 2*. The Coriolis Group.

[5] Bruce Eckel (2000). *Thinking in Java*. Second Edition. Prentice Hall.

[6] Laura Lemay and Rogers Cadenhead (2000). *Java 2. Guida Completa*. Apogeo s.r.l.

[7] David Flanagan (1997). *Java in a nutshell: A desktop quick reference*. Second Edition. O'Reilly.

[8] Patrick Chan and Rossana Lee (1997). *The Java class libraries: An annotated reference*. Addison Wesley.

[9] *The Java tutorial. A practical guide for programmers*. Sun Microsystems. Available online at java.sun.com.