Introduction

This book is designed for students in computer science and electronics with a basic knowledge of C language. Given the wide scope of applications of Java, our task is limited to the presentation of this language in the context of operating system and the Internet

The Java language is similar to C. The two languages share a large number of keywords; in C Java borrows terminology and syntax. The main difference lies in the fact that Java is an object oriented language. It is based on the concept of classes and objects that are instances of classes.

Classes

A Java program consists entirely of one or more classes and interfaces. One of these classes must define a *main ()* method, from which the Java program execution is initialized

```
public static void main(String argv[]) { }
```

The Java interpreter executes Java programs as long as the *main()* method is not completed. The arguments for a Java program are provided *by String argv [].*

```
public class mon_echo {
public static void main(String argv[]) {
  for(int i=0; i<argv.length; i++)
    System.out.println("argument " + i + ":" +arg[i]);
  System.exit(0);
  }
}
```

The method *exit()* can return a value back to the operating system. (see *exit ()* in C). One fundamental difference between the C language and Java is the lack of global variables or global functions. In Java all methods and variables are declared within classes.

Packages

Each Java class is stored in a separate file that must bear the same name as the main class. For example, a class ReadBytes must be saved in a file ReadBytes.class. Each class is part of a package. Packages are stored in files. In Java, files and subfolders are separated by one point. The files (classes) to be used in class are included with the import statement *import Name_of_package.Class;*

where

```
import Nom_du_package.*;      // etoile ( *)   signifie qu'on veut intégrér toutes les classes du package
```

Example:

```
import java.io.*;
public class ReadBytes {
```

```
public static void main(String argv[]) {
  ... }
```

Rules of accessibility

A package is available if its files and directories are accessible. All classes and interfaces in a package are accessible to other classes and interfaces in the same package. Classes declared as public are accessible from other packages. Classes without this statement are not accessible from another package.

Variables or methods are accessible to other classes in the same package. Variables or methods declared as *private* are accessible only within the class. Variables and methods declared as *public* can be accessed from other packages if their classes are also declared as *public*

Within methods, Java allows the use of local variables. These variables behave as local variables in C language and are inaccessible from outside the methods (functions).

TABLE 1. Règles d'accessibilité

keyword	class	package	class derivated	public
package	**yes**	**yes**		
private	**yes**			
protected	**yes**	**yes**	**yes**	
public	**yes**	**yes**	**yes**	**yes**

No pointer

A wealth of language like C are its pointers. C can handle finely the references to variables and functions. The pointer can also be difficult to control. Java does not have this type of operation and prohibits the use of pointers.

Inheritance

The general definition of a class is as follows:

```
[public] [abstract|final] class Nom_de_classe [extends Classe_de_base] [implements Interface] {
(variables)
(methods)
}
```

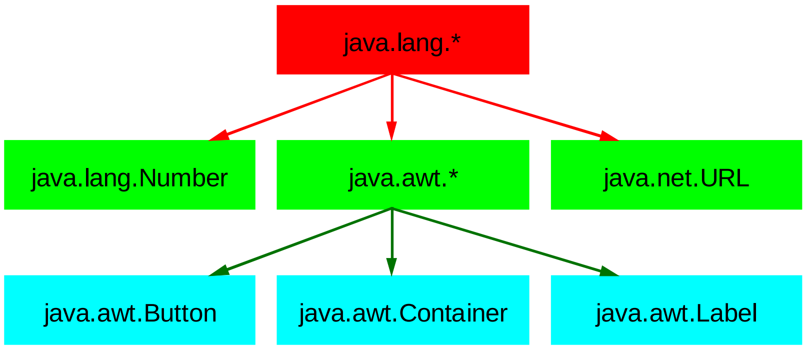
A class declared as abstract is not yet completed; all the methods of such a class are to be redefined before the usage. The classes, the methods and the variables marked by the keyword *final* are defined as constants, they can not be redefined.

The term *extends* indicates that the current class inherits all the characteristics of the base class, also called superclass (*super*).

```
public class Nom_de_classe extends Classe_de_base {
}
```

All methods and class variables are transmitted from the base class. The superclass of all classes is *java.lang.Objet*. All other classes are derived from this class with different paths of inheritance.

FIGURE 1._ **Hierarchy of classes**



Java does not support multiple inheritance. The interface mechanism makes multiple inheritance possible. The interfaces are abstract and it can not be instantiated

Instances and constructors

To instantiate a class (create an object), use the constructors. A class can contain several constructors. A constructor method determines how the instance (object) may be created, it bears the same name as the class.

Example:

```
class Personne {
String nom;
int age;
// méthode constructeur Personne
Personne(String s, int a){
nom= n;
age = a;
}
```

An object is created for each instantiation of a class, this instantiation is determined by the constructor of the class to derive.

```
[static][final] Nom_de_classe nom_de_l'objet =   new Nom_de_classe(paramètres);
```

Examples:

```
ComplexNumber cn = new ComplexNumber(2.67,3.14);
Button b = new Button("etiquette");
```

Memory for new objects is allocated dynamically. The creation of an object in Java works as a malloc() call in C language In Java, there is no standard method like free() used to release the space occupied by an object no longer used. The Java interpreter performs the "cleaning" of the memory automatically, this technique called “garbage collection”.

Arrays

A special case is *array* object. In Java, arrays are dynamically created by the instruction *new* and deleted by the garbage collector.

Exemples:

```
byte tampon[] = new byte[512];
Button buttons[] = new Button[10];
byte DeuxDim[] = new byte[16][32];
```

Strings

The Strings are instances of the class java.lang.String. Note that a character array (string) is not terminated by the null character. The content of a String is a constant.

Methods

In a class, the methods define the functions of the object. Methods can return a value (e.g. *int* or *String*) or carry out assignments and internal processing (*void*).

A method must be invoked with the application object:

```
Button sendbut = new Button(“send”);
String newtext = sendbut.getLabel();
```

Button is a class *sendbut* is an object of class *Button*, and *getLabel* () is a method to retrieve the label. In our example *sendbut.getLabel ()* returns the value "send".

Exceptions

An exception is a signal which indicates an exceptional situation such as a runtime error. The term *throw* means the possibility of detecting an exceptional condition. The term *catch* means that an exception is handled by the program itself.

The exceptions are propagated in the lexical structure of the program so that an exception not captured by a block is reported in the upper block. The exception that is not captured spreads to the method *main()* and causes an error message generated by the Java interpreter

An exception is an object that is an instance of a subclass *java.lang.Throwable*. Throwable has two subclasses *java.lang.Error* and *java.lang.Exception*. These errors are due mainly to the problems of memory overflow and they can not be captured

The three terms: *try/catch/ finally* form the mechanism for handling exceptions. The *try* block is completed by zero or more catch blocks. Each *catch* block defines the behavior of the program in case of emergency. The block *finally* completes this mechanism and indicates the default processing. It is always executed

```
try {
//instructions a executer et qui peuvent provoquer une erreur - exception }
catch {
// instructions a executer en cas d'erreur }
finally {
// instructions a executer dans tous les cas }
```

The exceptions applicable to standard methods in use are listed in *java.io.IOException* and in a set of more specific subclasses.

Among the most common exceptions are:

- *MalformedURLException*
- *IndexOutOfBoundsException*
- *IOException*
- *FileNotFoundException*
- *NumberFormatException*

An example:

```
try {
  in = new BufferedReader(new FileReader("toto.txt"));
  }
catch (FileNotFoundException e) {
  System.out.println("fichier " + "toto.txt" + "n'existe pas")
   }
```

Summary

This introductory chapter was devoted to some basic features of the Java language. In the second chapter we will introduce, in more detail, the elements mentioned in the introduction. We will talk about classes and objects while trying to explain their implementation and their behavior during the interpretation.

In the following chapters we will study classes prepared for the development of applications. These applications are based primarily on the management of streams and the lightweight processes called *threads*.

The *threads* are used to communicate via standard I/O, to manage files and pipes and to send the packets over the Internet. The threads can develop more complex applications requiring multiple programs running concurrently.
In the second part of this work we study how to program Internet applications using datagram mode and connected mode. We will also use standard protocols to develop some client-server applications.