

Programming languages

The programming language we will use in this subject is Java. This language arose in 1991 from a project carried out by Sun Microsystems which was aimed to design a language for programming electrical household appliances. It is not until the end of 1995 that it appears as a programming language for computers. Curiously, this was the first time that the specification of a programming language was discussed on the Internet. It has the advantage of being strongly introduced in the market and that most of you know it. Moreover, it has very complete and developed tools and libraries of classes.

Previously, Eiffel was used in this subject. Dr. Bertran Meyer, a well-known pioneer in OO programming, created it in 1985. It is commonly known that it is the most well designed and complete language. It has not been strongly introduced in the market yet, although there is some hope that this will change since it is part of the set of languages that Microsoft has incorporated into its platform ".net". Currently, the tools and libraries which are available for Eiffel are not as complete and developed as those one can find in other languages and, sometimes it is too difficult to work with it.

We have not time enough to do a complete course of Java, but, as well as the recommended bibliography for the subject, you may find good references and even learning material on the Internet. You should be careful since there are subtle differences among the different versions of the language.

Java is executed on a specific machine (or interpreter) called the *Java Virtual Machine* (JVM) from a neutral (or intermediate) code. JVM translates this intermediate code into the specific one of the machine where it is executed. The compiler converts the source files with extension ".java" into the intermediate code by creating one file per class of extension ".class", therefore an executable is a set of files with the extension ".class" (bytecodes), usually one per class (I say usually because when there are internal classes this condition will not be met). You must be careful about properly controlling all the versions of the files ".class", since it is quite common to combine inappropriate versions. You only have to state to the compiler (javac.exe in DOS) the class that contains the main program and it will be in charge of compiling the classes it needs. The compiler performs incremental compilation with the classes of public access (it just compares the day and the hour of the file ".class" with the corresponding ".java"). We have tools at our disposal that allow for storing all the files whose extension is ".class" in only one file which can be directly executed by the JVM, which makes both the installation and the manipulation easier in general.

We will comment on the most significant features of the language. To start with, there is not a clear difference between the attributes and the methods that comprise a class. Java syntax is inspired by C's. Thus:

```

if( condition ) {
    statements;
}
else {
    statements;
}

```

```

switch( expression_1 ) {
    case expression_2:
        statements;
        break;
    ...
    default:
        statements;
        break;
}

```

```

while( condition ) {
    statements;
}

```

```

do {
    statements;
} while( condition );

```

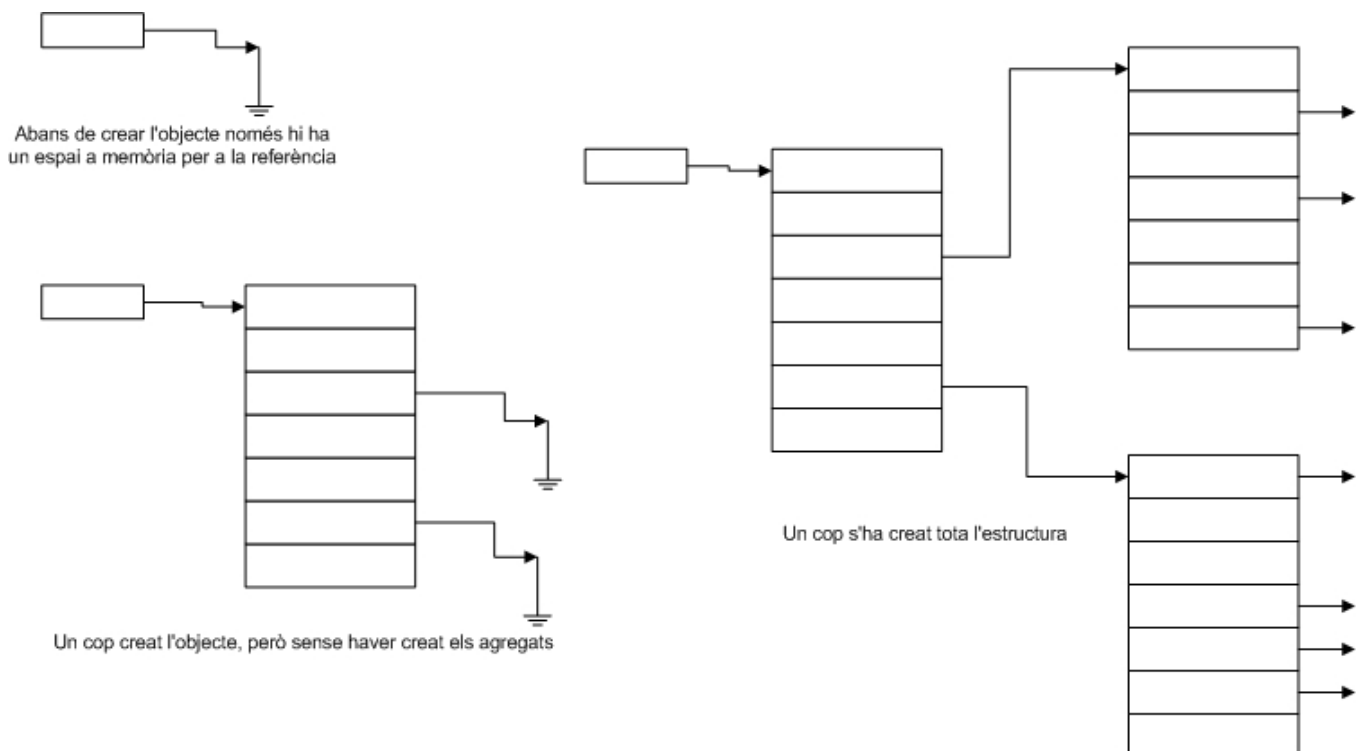
```

for( start ; condition ; increase ) {
    statements;
}

```

Attributes may be primitive or complex. The primitive types are: boolean, char, byte, short, int, long, float and double (in short, one logical, one for characters, 4 integer and 2 real ones).

Complex attributes may either come from the libraries of classes or be part of the development but, generally, the classes will be defined in separated files. Complex attributes are implemented by default in the structure of the class with a reference (a pointer) to the position of the memory where the true complex structured is instantiated.



It is advisable that you bear this characteristic in mind, since it provides the explanation of the real behaviour of the assignment, of the copy of classes, and of the pass of classes as parameters. Moreover, it justifies the complexity of the operations of saving and retrieving classes in files and databases. Generally, object-oriented languages implement objects in memory like this.

The values of the attributes are automatically initialized: the boolean value with *false*, the character with the *null character*, every numeric with *zero* and the references are created *empty* (this makes the programmer instantiate the referenced object always).

In Java (the same as in C) there is the word *void* that indicates the absence of type (essential for writing procedures).

Every class inherits from another one. In some cases this inheritance is explicit, written in the program code. In the rest of cases, it is produced by default when there is not the clause that specifies inheritance. In Java the class from which a class inherits by default is called OBJECT. Therefore it is good to know which are the consequences of this inheritance (attributes and methods), since every class receives them either directly or indirectly.

All constructor methods are called as the class, they differ in the arguments (quantity and type); the clause *new* plus the name of the class and the arguments are used in order to instantiate an object.

```
String literal
literal = new String("this is text")
```

Or simply:

```
String literal = "this is text"
```

Or:

```
String literal = new String("this is text")
```

The **methods** of the class are procedures and functions. In Java, as well as in C, a procedure is declared as a function that returns *void*, that is, they are declared the same and differ from each other in the return type.

```
Return_type <<nom>> (parameters)
```

The returned type (*return_tupe*) is unique, but it may be a structure as complex as needed. The result will be returned with the clause *return* followed by a result of the type *return_type*. Every function must execute a clause *return* prior to finish.

Arguments are always transferred by value and no other thing can be specified, but when a complex class is transferred the value will be the value of a reference, that is, a reference pass is being performed.

In order to declare **constants** we have the clauses *final* and *static*, which combined indicate that an attribute is constant:

```
public static final double PI=3.14159265358979323846;
```

The clause *final* indicates that it cannot be modified and *static*, the existence of an only copy for all the objects of the existing class, even before instantiating the first object of this class. But you should remember that the clause *final* has different meanings when it is applied to a method and to a class.

Visibility: any attribute or method of a class is visible inside the whole class, but limiting this visibility from the other classes of the program is possible. The elements of a class are visible by de-

fault by the classes that belong to the same package as the class. (To define the packages in Java you have the clause `package`, explained below).

To modify the visibility there are the clauses `private`, `protected` and `public`:

- `private`, visible only by the class, not even who inherits it.
- `protected`, visible by the members of the same package and who inherits.
- `public`, non-restricted access.

One class is either `public` or visible only into the package (that's the default value). On the other hand, according for attributes or methods it works as follows:

Visibility (from...)	public	protected	private	default
The class itself	Yes	Yes	Yes	Yes
Another class from the own package	Yes	Yes	Not	Yes
Another class outside the package	Yes	Not	Not	Not
One subclass from the own package	Yes	Yes	Not	Yes
One subclass outside the own package	Yes	Yes	Not	Not

Java has the clause `package` to create **packages** (instance of the topics and subsystems defined in the analysis phase). Java's packages structure has no relation with the structure of inheritances. In order to get a class to belong to a package (called `<<nameOfPackage>>` for example) the file ".java" must begin with the declaration:

```
package <<nameOfPackage>>;
```

The structure of the packages must correspond with the ones of the directories and subdirectories where the source (.java) and object (.class) files of the classes are stored.

Java calls **cast** to the mechanism that makes **compatibility between types** possible. It is automatic between primitive numerical classes and in the inheritance from children to parents, which allows to create an object of a father class with the structure of a child due to the fact that an object of the class child has the father's complete structure and, in general, it will be provided with some additional attributes and operations (for example, `PERSON onePerson = new CITIZEN()`, assuming that the class `CITIZEN` directly or indirectly inherits from the class `PERSON`). It also admits the cast from parents to children, but in this case it will be necessary to make it explicitly, as well as to define the remaining part.

The syntax of the language itself prevents from multiple **inheritance**, that is to say, it prevents from inheriting from more than one class. The clause for the inheritance is `extends`, it is provided with a syntax that only allows for simple inheritance.

One may consider that multiple inheritance is either partially or in a very limited way covered by the Java *interfaces* through the clause `implements`, but in fact an interface is not a class because it only accepts defining or accepting deferred methods that will have to be implemented by the inheritor and it has no attributes, that is, it is like a class without attributes where all the operations are deferred. The combination of interfaces and cast had allowed the previous versions of Java to implement something similar to genericity.

Just as the clause `extends` only accepts one name of class, the clause `implements` accepts a list of interfaces. The syntax is:

```
class <<name>> extends <<father>> implements <<listInterfaces>> {
```

A class can be declared as deferred (`abstract`), whenever it has any deferred element (declared also as `abstract`), but we will not be allowed to instantiate real objects of this type. They are declared

```
[public] abstract class <<name>>...
```

Up to the version 1.5, Java has not fully provided support to **genericity**, that is, quite recently.

Actually, it is inspired by the C++ *templates*, which are used for implementing genericity in that language. An example in Java:

```
List<Integer> myIntList = new LinkedList<Integer>();  
myIntList.add(new Integer(0));  
Integer x = myIntList.iterator().next();
```

Where `List` is a *generic interface* that takes a parameter type, in this case, `Integer`. Specifying the parameter type will be necessary every time an object of the class `List` is created. It will have been defined:

```
public interface List<E> {
```

The class `LinkedList` will implement `List` or will inherit from anyone who did it in order to make cast between her and `List` possible.

A generic class can accept more than one parameter (such as: `class HashTable <Key, Value>`) and it can be constrained that the parameter class is such that it inherits from a specific class (either with `extends` or `implements`).

```
interface Hashable {...};  
class HashTable <Key implements Hashable, Value> {...};
```

Java accepts a treatment of the errors through the classes derived of *Exception* (`java.lang.Exception`) by using the blocks *try*, *catch* and *finally*. On the other hand, it has not an own mechanism as the one that Eifel has that allows it to incorporate the formal specification (pre and postconditions, invariants, etc.) in the source of the program so that it can be used in order to check the level of correctness of the program in the test phase.

Java incorporates in the language itself a big number of aspects that in any other language are extensions which belong to software companies or to computers' manufacturers (*threads*, remote execution, components, security, access to databases, etc.). For this reason several experts believe that *Java* is the perfect language to learn modern computing, because it incorporates all these concepts in a standard way, much more simple and clear than with the aforementioned extensions of other languages.

Javier García de Jalón, José Ignacio Rodríguez, Iñigo Mingo, Aitor Imaz, Alfonso Brazales, Alberto Larzabal, Jesús Calleja i Jon García. Escuela Superior de Ingenieros Industriales de San Sebastián. Universidad de Navarra.

Java v1.5 compiler

You must have at least the JDK (Java Development Kit), that you can download from <http://java.sun.com>.

It is very convenient that the PATH from DOS includes the directory BIN of the compiler (for example: "C:\Program files\fib\jdk1.5\bin").

The environment variable CLASSPATH indicates to the compiler the directories where it has to search the classes. In the last versions, CLASSPATH does not have to include the current directory nor the ones of the standard libraries.

The statement of compilation is:

```
javac <<nameOfFile>>.java
```

Putting the extension ".java" is necessary, whereas in the execution statement the extension will not be put:

```
java <<nameOfFile>> [arguments]
```

You can create libraries of classes in ".jar" or ".zip" files (without packing the latter ones). All the files ".class" of an executable can also be stored in only one file ".jar" and directly execute it with the JVM. In the directory BIN there is a utility tool to create files ".jar". If you need the compiler to handle the file ".jar" as a directory or library of classes, you will have to add the full name of the file to CLASSPATH.

Only the JVM is necessary for the environment of execution, you can get it from the same place by downloading *J2SE Runtime Environment (JRE)*.