Seminario 1

Introduction to Java

PROGRAMMING 3

David Rizo, Pedro J. Ponce de León
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante

Contents

David Rizo, Pedro J. Ponce de León

Main Java features

- Object oriented language: (almost) everything is an object
- Source files: extension *.java*. Each file contains a class. The name of the file must match the name of the class.
- It is compiled to *bytecode*. For each source file, a *bytecode* file with extension *.class* is generated.
- Libraries: *.jar* files containing *. class* files
- *Java Development Kit* (JDK): includes the Java compiler (`javac`), the standard Java libraries and other utilities to develop Java code.
- Main Integrated Development Environments (IDE): *Eclipse**, *Netbeans* and *Intellij IDEA*

(*)We will use Eclipse

Main Java features

- *Bytecode* is an intermediate language that is interpreted and executed by the multiplatform *Java Virtual Machine*.
- *JRE, Java Runtime Environment*: includes the Java Virtual Machine (`java`), standard libraries and other utilities for *bytecode* execution.
- JDK includes the JRE. On a machine where we develop Java applications we would install the JDK.
- On a machine where we want to run Java applications, we only need to install the JRE.

David Rizo, Pedro J. Ponce de León

Basic syntax

The naming rules are basically the same as those used for C++. In Java 'jargon', the fields of a class are called '*attributes*' and the functions of a class are called '*methods*'.

```
// This file must be saved as Clase.java
// In general, each class is located in single file
public class Clase {
    /* All attributes must specify its visibility */
    private int campo1;
    private float campo2; // attributes are initialised to 0

    /* The constructor returns nothing */
    public Clase() { campo1 = 0; }

    /* All methods are defined inline */
    public int getCampo1() {
        return campo1;
    }
```

Constants, static [1]

Constants

Constants are defined using the reserved word `final`

```
public final int KN=10;
```

Methods and static fields

They are defined using the reserved word `static`

```
private static int contador=1;
public static final int KNN=10;
public static void incrementaContador () {
    contador++;
}
```

Main program

.

main

The entry point to an application is the *main* method, a static method:

```
// this is a normal class
public class ClaseConMain {
    // which also has the main method
    // so it can be invoked from the virtual machine
    public static void main(String[] args) {
        // the array 'args' contains the arguments that
        // are passed  to the program from the command line.
        // without including (as in C++) the name of
        // the executable
    }
}
```

Console output

Output

To print through the standard output

```
System.out.print("Cadena"); // without end line
System.out.println(10+3); // with end line
```

To print through the error output

```
System.err.println("An␣error␣has␣occurred...");
```

Compilation and command line execution

Compilation

Compilation in the command line (terminal):

```
> javac ClaseConMain.java
```

It generates the *bytecode* file `ClaseConMain.class`, in the current directory (default behaviour).

Execution

With the command `java` we invoke the virtual machine, indicating the name of the class that contains the *main() method*

```
> java ClaseConMain
```

The file *ClaseConMain.class* must be in the current directory (default behaviour; later on you will see how to change this).

Compiling and executing

ACTIVITY

Now you are ready to compile and execute your first Java program. In the folder `codigo` accompanying this document you will find the class *ClaseConMain* ready to be compiled and executed.

- Open the file *ClaseConMain.java* with a text editor or code editor and take a look at it to get an idea of what the program does.
- Then compile it and run it from the terminal in the folder *codigo*.

Scalar data types [2]

Java is a strongly-typed language. Every expression in the language has an associated type. Almost all types in Java are objects, except the basic scalar data types:

Scalar data types (not objects)

byte, short, int, long, float, double, char, boolean

Java literals are specified as follows:

```
int x = -14;
float a = 100.3f;
double b = 100.3; // or 1.03e2
char c = 'a';
boolean d = true; // or false
```

Operators

We have the same operators as in C++

```
a+b*3;
a++;
if (a==1) b=2;
a = (float)b;
 ...
```

Scalar data types [2]

ACTIVITY

In some slides, like this one and the previous one, you will see that the title has a number between brackets ([2]). This number will help you check the code presented on that slide by running a program that you will find in the accompanying folder `codigo`. The source code file is `Clase.java`. Compile it and execute it passing as argument the number indicated between brackets:

```
> java Clase 2
```

it will show you the result of the code that appears on the slide. You can also look up the source code and search for the string `"// [N]"`, replacing 'N' with the corresponding number. It will take you to a method that contains the code of the current slide.

David Rizo, Pedro J. Ponce de León

Objects

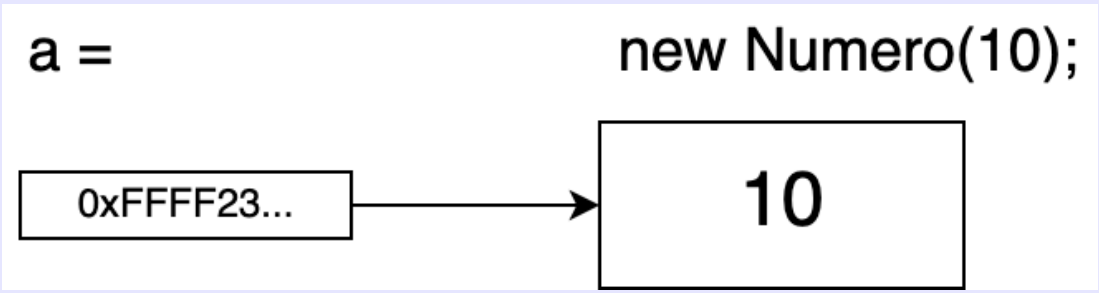
Objects and references to objects

- Objects are always created dynamically with the operator `new`.
- To store the memory address of an object we use **object references** (or *'references'* for short). They are equivalent to pointers in C++.
- References have, by default, a `null` value (in lower case).

In the following code, 'Numero' is a dummy class and 'a' is a reference:

```
Numero a = new Numero(10);
```

It can be graphically represented as:

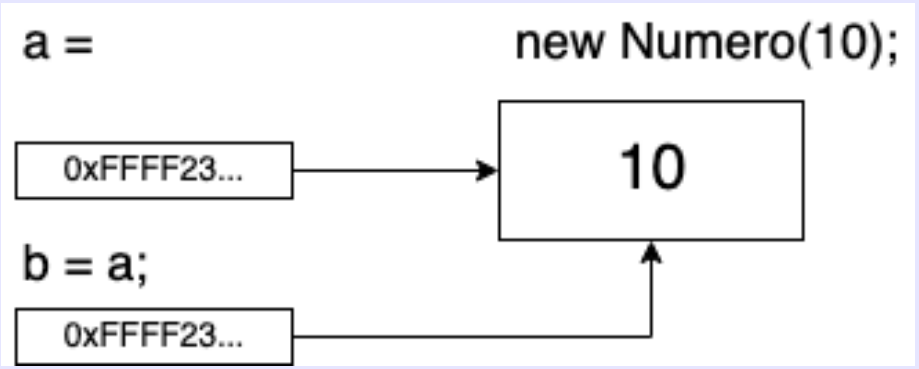


Objects

Reference assignment

```
Numero a = new Numero(10);
Numero b = a;
```

'b' points to the same instance as 'a', i.e. to the same object, the same memory area. We have two references pointing to a single object.

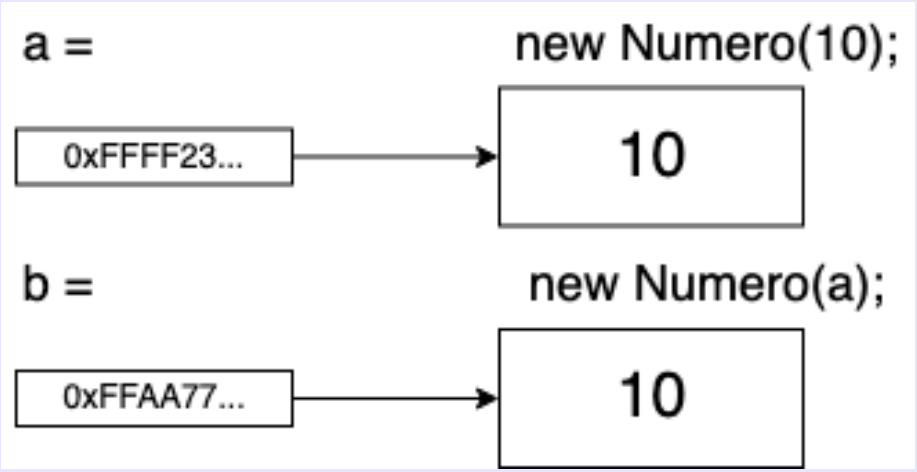


Objects

Copying objects

To duplicate an object it is necessary to create a new object with `new`, possibly invoking the *copy constructor*:

```
Numero a = new Numero(10);
Numero b = new Numero(a);
```



Objects

The class *Object*

The class *Object* represents *all objects* in Java. Any object of any class is also an object of the class Object.

```
Object obj = new Numero(10); // Ok
obj = new Persona(); // Ok
```

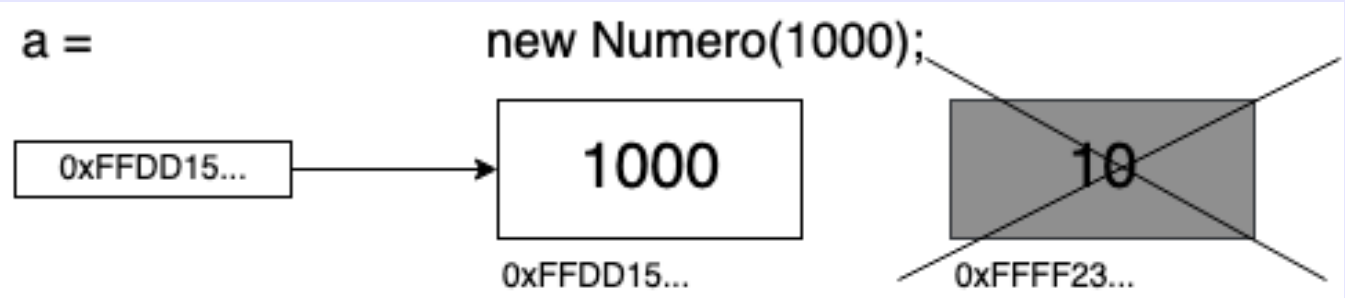
We can use references of type *Object* to point to any type of object.

Objects

Java Garbage Collector

In Java it is not necessary to explicitly free the memory of objects that we no longer need. The *garbage collector* takes care of objects that run out of references pointing to them:

```
Numero a = new Numero(10);
a = new Numero(1000);
```



Objects

operator instanceof

The expression

```
obj instanceof Clase
```

returns true if 'obj' points to an object of class 'Clase', and false otherwise.

Casting (conversion)

It is similar to C++:

```
int x = 10;
float f = (float) x;
```

Given an object of any type, we can also use the cast operator to assign it to a reference of a known type:

```
Object any;
MiClase obj = (MiClase) any;
```

Note: to do the cast without risk, we must be sure that 'any' points to an object of type 'MiClase'.

Objects

Dot operator ('.')

To access the attributes or invoke the methods of a class using an object reference, we use the '.' operator:

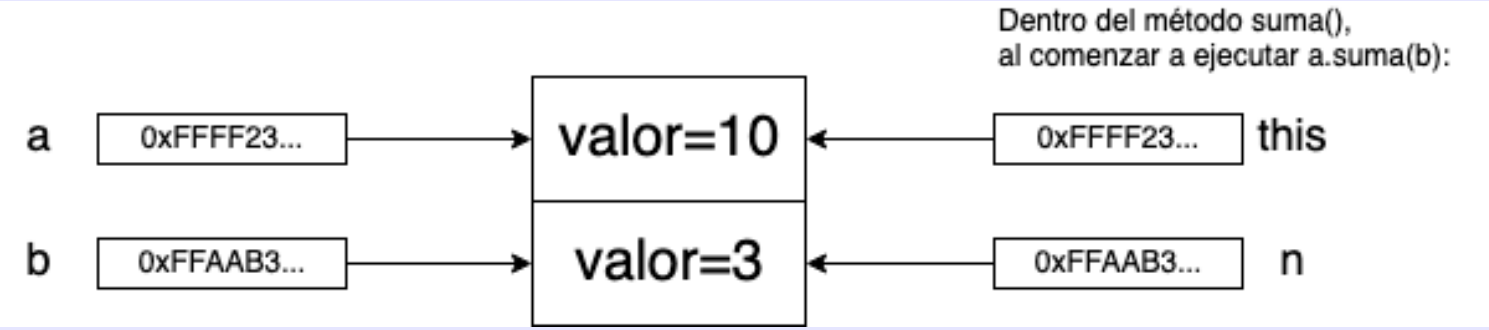
```
Numero a = new Numero(10);
Numero b = new Numero(100);

a.valor = 3;
b.valor = 4;
a.suma(b);
a.incrementa(3);
```

Objects

Reference `this` [3]

As in C++, inside a non-static method we can refer to the object used to invoke the method with the reference `this`.

```
public class Numero {
    private int valor;
    public Numero(int valor) { this.valor = valor; }
    public void suma(Numero n) { this.valor += n.valor; }
}
...
Numero a = new Numero(10);
Numero b = new Numero(3);
a.suma(b);
```



Objects

ACTIVITY

Run

```
> java Clase 3
```

to explore the result of the code in the previous slide (we will not remind you anymore, remember to do it when you see the number between brackets on a slide).

Objects

Comparison

The expression

```
a==b
```

is comparing references, i.e. memory addresses. To compare the content of two objects we must do:

```
a.equals(b)
```

Method 'equals'

If we want to be able to compare objects of a class we have created, we must implement the 'equals' method.

```
public boolean equals(Object obj)
```

The argument of equals is a reference to an object of type 'Object'. This means that an object of any class can be passed to the equals method (although it will usually be an object of the same type as the object we want to compare it to).

Objects

Implementation of 'equals' [4]

To implement the 'equals' method we must take into account that the equals operation must comply with the reflexive, symmetric and transitive properties and make sure that

```
x.equals(null) == false // for any non-null x
```

Furthermore, to compare the attributes of the argument with those of the object `this`, we must convert the argument to a reference of our class. Therefore, any implementation of the equals method must perform these comporbations:

```
public boolean equals(Object obj) {
    if (obj == this) return true; // both references
                                  // point to the same object
    if (obj == null) return false;
    if (!(obj instanceof MiClase)) return false;
    MiClase elotro = (MiClase) obj;
    // Starting here compare the attributes of both
    // objects ('this' and 'elotro') to determine whether
    // they are the same or not.
    // WARNING! if the attributes are references to objects,
    // use 'equals' to compare them.
}
```

Objects

Wrappers

Each scalar type has an equivalent class:

```
Byte, Integer, Float, Double, Char, Boolean
```

that are initialised

```
Integer a = null; // is null by default
a = new Integer(29);
int x = a.intValue(); // x is 29
```

Java simulates compatibility between scalar types and their corresponding 'wrapper' classes by direct assignment between them. This is known as 'boxing' and 'unboxing'.

Objects

Boxing [5]

When we do

```
Integer b = 3;
```

internally the following is done

```
Integer b = new Integer(3);
```

Unboxing [5]

and on the contrary, when we write

```
Integer b = new Integer(100);
int x = b;
```

internally the following is done

```
int x = b.intValue();
```

Exceptions

Concept

- An exception is a mechanism designed to handle error situations by altering the normal execution flow of a program.
- Examples of exceptions are access to an invalid memory address, division by zero, or the use of a negative position in an array.
- In its most basic form, when an exception occurs, the method being executed aborts its execution and returns the control to the method that invoked it. This operation is repeated until the main program is reached, which stops the execution of the program.
- Exceptions are objects of a class with a name with the form 'BlablaException'.

David Rizo, Pedro J. Ponce de León

Exceptions [6]

The two exceptions we are most likely to encounter are:

NullPointerException

It is thrown when we are accessing an uninitialised memory address (for which no new has been made). For example:

```
Integer a, b;
if (a.equals(b)) { // this if throws NullPointerException
}
....
```

ArrayIndexOutOfBoundsException

It is thrown when an invalid position in an array is accessed. For example:

```
int [] v = new int[10];
v[20] = 3; // this throws ArrayIndexOutOfBoundsException
```

Strings [7]

String

Java has a class for working with strings

```
String s = new String("Hola");
```

Remember the comparison

```
s == "Hola" // wrong
s.equals("Hola") // ok
```

toString()

All classes usually implement the `toString()` method. Java uses this method to convert an object of any class to a string.

```
Float f = new Float(20);
String s = f.toString();
```

WARNING: 'String' starts with a capital letter. Class names in Java are usually written with an initial capital letter.

David Rizo, Pedro J. Ponce de León

Strings

Concatenation

Strings can be concatenated with the + operator. If we mix other types, they are automatically converted to a string using the `toString()` method.

```
int i=100;
"El valor de i es = " + i;
```

This code internally creates 4 objects, it does

```
String s1 = new String("El valor de i es = ");
String s2 = new Integer(i).toString();
String s3 = s1.concat(s2); // which creates a new object
```

StringBuilder

To avoid creating so many objects we can use `StringBuilder`

```
StringBuilder sb = new StringBuilder();
sb.append("El valor de i es = ");
sb.append(i);
sb.toString(); // it returns a String object
```

Arrays [9]

Arrays are defined as the dynamic arrays of C++

```
Integer [] v; // v is reference pointing to null
```

Arrays are objects. They are initialised as follows:

```
v = new Integer [100];
```

Now the components of `v`, that is `v[0], v[1],` , etc... are `null`, they must be initialised

```
// v.length is the length reserved for the array
for (int i=0; i<v.length; i++) {
    v[i] = new Integer(0);
    // v[i] = 0 (equivalent because of boxing)
}
```

You can create literal arrays by also allocating memory

```
int [] v = new int []{1,2,3,4,5}; // contains 5 integers
```

Arrays can be copied using a loop or with the static method `arraycopy` of class `System`

```
int [] origen = new int []{1,2,3,4,5};
int [] destino = new int[origen.length];
System.arraycopy(origen, 0, destino, 0, origen.length);
```

Methods, call by value

Methods

The member functions of a class are called methods.

Parameters

All parameters are called by value

```
void F(int a, String x, int [] v) {
    a=10; // this change does not affect the original value
    x = "Hola"; // it does not affect the original
    v[2] = 7000; // what has been passed by value is
                 // the reference to v, v[2] is changed
                 //in the original
}
```

Methods, call by value

ACTIVITY [10]

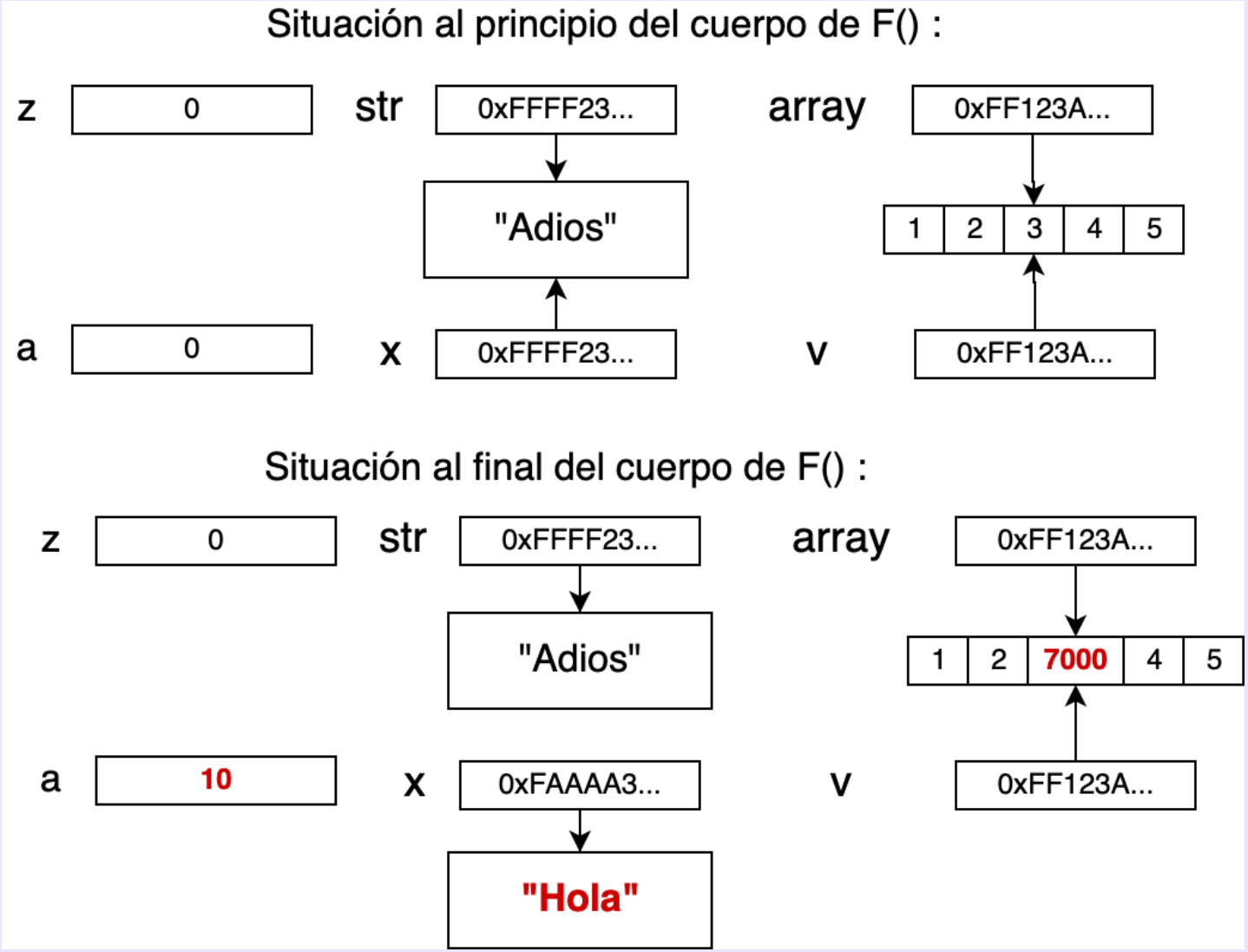
Given the function F() defined in the previous slide and the following code

```
int z = 0;
String str = "Adios";
int [] array = new int []{1,2,3,4,5};
F(z,str,array);
```

- Try to predict the value of 'z', 'str' and 'array[2]' after the call to F().
- Execute the activity with the command `java Clase 10` in the folder `codigo` and check if you got it right.
- See the diagram on the next slide to understand what happened.

Methods, call by value

ACTIVITY [10]



Flow control [11]

In general it does not change with respect to C++. From Java version 1.7 there is the possibility to use strings in a `switch`.

Loops

To go through arrays, for example, we will use:

```
String v[] = new String[] {"Azul", "Verde", "Rojo"};

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

for (String color: v) {
    ystem.out.println(color); // prints one colour per line
}
```

The 'if', 'while' and 'do-while' instructions are used just like in C++.

Packages

package

Classes are physically distributed in directories. They make up what is called `packages`

For a class to be in a package you must:

- Save the file of the class in the package directory
- Declare at the beginning of the file the `package` it belongs to, separating directories (packages) with dots

```
package prog3.ejemplos;
class Ejemplo { }
```

The file `Ejemplo.java` must be saved in the directory `prog3/ejemplos`.

Packages

Modularization

It is not mandatory to use packages, but it is recommended. If we want to use a class from another package we must include it:

```
package prog3.ejemplos;

// Java library class
import java.util.ArrayList;

// Class from another package of ours
import prog3.otrosejemplos.Clase;

// The following includes all classes of package prog3.practicas.
// For traceability, it's better not to use the *
import prog3.practicas.*;
```

The classes belonging to the `java.lang` package are included by default; it is not necessary to include them explicitly

```
// The following is not necessary,
// all java.lang classes are included by default
import java.lang.String;
```

Packages

ACTIVITY

- Locate in the folder `codigo` the package `prog3` and inside it the classes `Ejemplo` and `Clase`.
- Check their source code and in particular the instructions `package` and `import`.
- Inside the `codigo` folder, compile the class *Ejemplo*: `javac prog3/ejemplos/Ejemplo.java`
- Where is the `Ejemplo.class` file saved?
- The compiler has also compiled the file `prog3/otrosejemplos/Clase.java` without you asking to do it. Why?

Java API

API (Application Programming Interface)

Java has an extensive library of classes that can be consulted at `http://download.oracle.com/javase/8/docs/api/overview-summary.html`

- It is structured in packages.
- The package `java.lang` contains the basic classes of the language
- The package `java.util` contains classes to create collections of objects: dynamic vectors (lists), sets, maps, etc.

David Rizo, Pedro J. Ponce de León

Dynamic vectors [12]

As a linear dynamic storage medium we will use the class `ArrayList`.

```
import java.util.ArrayList; // needed
....
ArrayList<Integer> v = new ArrayList<Integer>();
v.add(87); // this internally makes v.add(new Integer(87))
v.add(22); // this makes the vector bigger

ArrayList<String> sv = new ArrayList<String>();
sv.add("PROG3");
sv.add("JAVA");

// get() returns an object of the specified type
Integer a = v.get(0);
String s = sv.get(1); // "JAVA".

v.get(100); // throws an exception (runtime error)
System.out.println(v.size()); // size() returns the size
```

CLASSPATH

ClassNotFoundException

This exception sometimes appears when trying to start a Java program. Before starting the main program, the virtual machine must load all the necessary .class files. If it does not find any, it throws this exception.

Ejemplo

```
mihome$ java Clase
Error: Could not find or load main class Clase
Caused by: java.lang.ClassNotFoundException: Clase
```

But where should those files be? In the *classpath*.

CLASSPATH

classpath

The *classpath* is the list of directories where Java looks for the .class files needed to run an application.
By default they are

- the current directory
- path to the JRE (Java Runtime Environment) libraries, where the Java API .class files are located.

David Rizo, Pedro J. Ponce de León

CLASSPATH

Let's suppose that our main program is compiled in a file called `Clase.class` that resides in `/home/mihome/codigo`.

Case 1

All our classes are in the same directory. We do not use `package`. From that directory,

```
/home/mihome/codigo> java Clase
```

(You have already checked that this works if you have done the activities)

CLASSPATH

Case 2

We run `java` from a different directory than the one containing our `.class`. You have to define the *classpath*:

Option 1

Define the environment variable **CLASSPATH** with the list of directories where the .class are (better to use absolute paths)

```
.../otrodirectorio> export CLASSPATH=/home/mihome/codigo

 .../otrodirectorio> java Clase
```

Option 2

Use the `Java` options **-cp** or **-classpath**:

```
.../otrodirectorio> java -cp /home/mihome/codigo Clase
```

CLASSPATH

Case 3

The .class are distributed in several directories.

```
> export CLASSPATH=/home/mihome/milibjava:/home/mihome/codigo
> java Clase
```

or use **-cp**. WARNING: the option '-cp' cancels the environment variable CLASSPATH. You must use one or the other, but not both at the same time.

Packages and *classpath*

When our classes are organised in packages. Suppose we have the following:

Project structure

modelo/MiClase.java:

```
package modelo;
public class MiClase {...}
```

mains/Main.java:

```
package mains;
public class Main {...}
```

modelo/m2/OtraClase.java:

```
package modelo.m2;
public class OtraClase {...}
```

The *classpath* must contain **the parent directory** of the package structure.

Packages and *classpath*

Suppose that the project directory is `/home/mihome/codigo`. If we want to use **OtraClase** in Main.java:

```
import modelo.m2.OtraClase;
```

When running

```
.../otrodir>java -cp /home/mihome/codigo mains.Main
```

to be able to run the class Main, 'java' will look in the *classpath* directory for a directory with name `mains` and inside it, it will look for the file `Main.class`, the directory `modelo/m2`, and inside this last directory for the file `OtraClase.class`.

David Rizo, Pedro J. Ponce de León

Packages and *classpath*

ACTIVITY

In the folder `codigo` you will find the packages `model` and `mains` indicated in the previous slides. Your goal will be to run the main program from another directory.

- Take a look at the code of the three classes in the packages.
- Switch to another directory.
- Set the *classpath* so that it points to the folder `codigo` (use absolute paths). You can use the CLASSPATH environment variable or the -cp option of the virtual machine.
- Run the main program located in the class *Main* as indicated in the previous slide.

David Rizo, Pedro J. Ponce de León

Documentation I

Javadoc

In Java a format based on embedded annotations in comments is used. They start with `/**` and the annotations types start with `@`:

```
package paquete;
/**
* Example class: we briefly document the task
* of the class
* @author drizo
* @version 1.8.2011
*/
public class Ejemplo {

    /**
     * This is a field for ...
     */
    private int x;

    private int y; // this does not appear in the javadoc
```

Documentation II

```
    /**
     * Constructor: makes this operation...
     * @param ax is the radius of ...
     * @param ab If it's true ...
     * '@param' documents arguments of a method.
     * Its format is '@param <id> <description>'
     * <id> must match the name of one of the arguments
     */
    public Ejemplo(int ax, boolean ab) { ... }

    /**
     * Getter.
     * @return a value always greater than zero...
     */
    public double getX() { return x; }
}
```

Documentation III

Generation

Documentation in HTML is generated as follows:

```
javadoc -d doc paquete otropaquete
```

generates a directory `doc` with the documentation of the classes in packages `paquete` and `otropaquete`.

ACTIVITY

- Generate the documentation of the package `mains` that is in the folder `codigo` that accompanies this document.
- To see the result open the generated file `doc/index.html` in your browser.

David Rizo, Pedro J. Ponce de León

JAR files

jar is a Java utility (similar to **tar**) to pack in a single file with extension **.jar** a directory structure. It is usually used for .class files.

JAR

To pack, from the working directory:

```
/home/mihome/code> jar cvf MisClases.jar mains model
```

Now we can put MisClases.jar wherever we want (e. g. /home/mihome/libs) and use it from anywhere:

```
> java -cp /home/mihome/libs/MisClases.jar mains.Main
```

To see the content of a .jar file:

```
> jar tvf MisClases.jar
```

ANT

ant

ant is a tool to automate the different tasks related to compilation, generation of documentation, jar files, etc. It is similar to 'make'. In *Programming 3* we will use it as part of the script that automates the grading of your assignment submissions. You do not need to know how it works, but if you are curious...

'ant' tutorial

In the following link you have a brief tutorial:
`https://www.tutorialspoint.com/ant/index.htm`