# Resumen de Java para programadores de C y C++

acastan@xtec.net

El siguiente documento no es una guía de aprendizaje, sino una referencia rápida de la sintaxis de Java. Si buscas un curso para aprender a programar en Java, te recomiendo consultar el tutorial de Sun en la página web http://java.sun.com/docs/books/tutorial/.



## Tabla de contenidos



## Java

Java es un lenguaje de programación moderno. Características:

- Sintaxis similar a C y C++, pero sin punteros (la gestión de la memoria dinámica es automática).
- Interpretado: un programa escrito en código Java (.java) se compila a código bytecode (.class) y dicho código lo interpreta una máquina virtual de Java.
- Multiplataforma: el programador compila una única vez el programa Java, y el fichero de bytecode que obtiene se ejecuta igual en la máquina virtual de Java de cualquier plataforma.
- Seguro: la máquina virtual Java sobre la que se ejecuta el programa controla que dicho programa no intente ejecutar operaciones no permitidas sobre los recursos del sistema.

Utilidades de la línea de comandos al instalar Java:

```
(compilar)          javac [-classpath camino] [-d carpeta] [-g] nombre.java
(ejecutar)          java [-classpath camino] nombre_clase argumentos
```

```
(ejecutar applet)     appletviewer url
(depurar)             jdb nombre_clase
(crear documentación) javadoc [-classpath camino] [-d carpeta] nombre.java
```

## Comentarios

En Java se mantienen los comentarios de varias líneas de C, los comentarios de una línea de C++, y aparece un nuevo tipo de comentario utilizado para generar automáticamente ficheros de documentación a partir del texto del comentario:

```
// Comentarios de una línea

/* Comentarios de
   varias líneas */

/** Comentarios que generan
    documentación automática */
```

Los comentarios que generan documentación pueden incluir etiquetas HTML y etiquetas @ que marcan un campo de información en particular. Ejemplo:

```
/**
 * Clase <code>Universidad</code>
 * Contiene una lista de alumnos.
 *
 * @author      Alex
 * @version     1.0
 * @see         java.UOC.Alumno
 */
public class Universidad {

  /** Alumnos de la universidad */
  public Vector alumnos;

  /**
   * Método que añade un nuevo alumno a la universidad
   * @param a   Alumno a añadir
   * @return    Número de alumnos de la universidad
   * @throws    UniversidadException Si el alumno a añadir ya existía
   */
  public int añadir(Alumno a) throws UniversidadException {
    ...
  }
}
```

## Declaración de constantes

En java la declaración de una constante és como la declaración de una variable, pero con la palabra `final` delante:

```
final <tipo_datos> <nombre_constante> = <valor>;
```

Ejemplo:

```
final int MAX_ELEM = 20;
```

Desde la versión 1.5.0, Java permite declarar tipos enumerados. A diferencia de C, cada tipo enumerado definido en Java crea una nueva clase, que puede llegar tener campos y métodos. En su forma más simple la sintaxis es:

```
enum <nombre_enumeracion> { <nombre_constante>, ..., <nombre_constante> }
```

Ejemplo:

```
enum estacion { PRIMAVERA, VERANO, OTOÑO, INVIERNO }
estacion a = estacion.VERANO;
```

## *Declaración de variables, y tipos de datos*

En Java la sintaxis de la declaración de variables es similar a C y C++:

**<tipo_datos> <nombre_variable>;**

El nombre de una variable debe estar formado por un primer carácter letra, y el resto puede ser cualquier carácter unicode (incluidas vocales con acentos y letras de otros alfabetos).

Tipos de datos simples (no son clases):

**<tipo_datos_simple> <nombre_variable> [= <valor>];**

| tipo | tamaño | rango |
|---|---|---|
| byte | 8 bits | −128 .. 127 |
| short | 16 bits | −32.768 .. 32.767 |
| int | 32 bits | −2.147.483.648 .. 2.147.483.647 |
| long | 64 bits | −9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807 |
| float | 32 bits | $-3.4x10^{38}$ .. $3.4x10^{38}$ (mínimo $1.4x10^{-45}$) |
| double | 64 bits | $-1.8x10^{308}$ .. $1.8x10^{308}$ (mínimo $4.9x10^{-324}$) |
| boolean | | true o false |
| char | 16 bits | unicode |

En memoria: variable | información | Ejemplo: `int i = 0` i | 0

Tipos de datos compuestos (son clases):

`String, vectores y matrices, colecciones, clases, wrappers/envolventes`

**<tipo_datos_compuesto> <nombre_variable> = new <tipo_datos>(<valor_inicial>);**

En memoria: variable | posición → | información | Ejemplo: `String s = "hola"` s | → | hola

Ejemplo:

```
int i = 0;                        // Tipo simple
int v[] = new int[100];           // Tipo compuesto vector
String s = new String("Hola");    // Tipo compuesto String
Integer n = new Integer(10);      // Tipo compuesto Integer (wrapper del tipo int)
```

Ejemplo de conversión entre tipos numéricos y String:

```
int i = Integer.parseInt("12");           // cadena caracteres a entero
float f = Float.parseFloat("1.15");       // cadena caracteres a real
String s1 = new Float(f).toString();      // real a cadena caracteres
String s2 = new Integer(i).toString();    // entero a cadena caracteres
String s3 = "Números:" + i + " y " + f;   // más fácil: números a cadena caracteres
i = (int) f;                              // convertir real a entero
f = i;                                    // convertir entero a real
```

## *El tipo de datos cadena de caracteres*

La palabra `String` representa el tipo de datos cadena de caracteres:

**String <nombre_variable> [= "<cadena de caracteres>"];**

Sus atributos y métodos más importantes son:

```
string1.equals(string2)                  // Compara dos strings
string1.clone()                          // crea una copia de un string
string1.charAt(posicion)                 // retorna el carácter en una posición
string1.concat(string2)                  // concatena dos strings
string1.indexOF(caracter, posicion)      // devuelve la posición de un carácter
string1.length()                         // devuelve la longitud del string
string1.replace(caracter1, caracter2)    // reemplaza un carácter por otro
string1.substring(posicion1, posicion2)  // extrae una porción del string
string1.toLowerCase()                    // convierte el string a minúsculas
```

```
string1.toUpperCase()                    // convierte el string a mayúsculas
string1.valueOf(numero)                  // convierte un número a string
```

Ejemplo:

```
String s1 = "hola";
String s2 = "adios";
String s3 = s1 + " y " + s2 + " : " + 2004;
```

## Los tipos de datos envolventes

Para cada uno de los tipos simples existe una clase equivalente, con constantes y métodos que nos pueden ser útiles:

| Tipo Simple | | Clase Equivalente |
|---|---|---|
| `byte`, `short`, `int` y `long` | → | `Number`, `Byte`, `Short`, `Integer` y `Long` |
| `float` y `double` | → | `Number`, `Float` y `Double` |
| `boolean` | → | `Boolean` |
| `char` | → | `Character` |

Ejemplo:

```
int i = 4;                      // "i" es una variable entera (tipo simple)
Integer j = new Integer(5);     // "j" es un objeto de la clase Integer (envolvente)
```

## Los tipos de datos vector y matriz

En Java la sintaxis para trabajar con vectores y matrices es similar a C y C++ (se indexa con [ ] y el primer elemento está en la posición 0), pero el número de elementos se declara dinámicamente con `new`, y se puede obtener su longitud con `length`:

Una dimensión:

```
<tipo_datos> <nombre_array>[];
<nombre_array> = new <tipo_datos>[<num_elementos>];
```

Varias dimensiones:

```
<tipo_datos> <nombre_array>[][]...[];
<nombre_array> = new <tipo_datos>[<num_elementos>]...[<num_elementos>];
```

Ejemplo:

```
float v[] = new float[10];           // Una dimensión con 10 elementos
float M[][]= new float[3][4];        // Dos dimensiones con 3x4 elementos
String s[] = {"hola", "adios"};      // Una dimensión con dos elementos inicializados

for (int i = 0; i < v.length; i++)
  v[i] = i;

for (int i = 0; i < M.length; i++)
  for (int j = 0; j < M[i].length; j++)
    M[i][j] = 0;
```

## Las colecciones

La API de Java nos proporciona colecciones donde guardar series de datos: `List`, `ArrayList`, `Vector`, .... Dichas colecciones NO forman parte del lenguage, sino que son clases definidas en el paquete `java.util`. Desde la versión 1.5.0, Java permite además colecciones genéricas:

```
<tipo_collección><<tipo_datos>> <nombre_colección>;
```

```
<nombre_colección> = new <tipo_collección><<tipo_datos>>();
```

Ejemplo:

```
ArrayList<Alumno> miclase = new ArrayList<Alumno>();
miclase.add( new Alumno("Pepe", 5.0) );
miclase.add( new Alumno("Alex", 4.2) );
miclase.add( new Alumno("Pepa", 6.3) );
for (Iterator i = miclase.iterator(); i.hasNext();) {
  System.out.println( i.next() );
}
```

## Código

Java utiliza la misma sintaxis que C y C++:

Una instrucción puede ocupar varias líneas, y acaba en punto y coma.

```
instrucción;
```

Los bloques de instrucciones van entre corchetes:

```
{
  instrucción1;
  instrucción2;
  ...
}
```

## Expresiones y operadores

Java utiliza los mismos operadores que C y C++:

```
(asignación)    =
(aritmética)    ++, --, +, -, *, /, %
(comparación)   ==, ¡=, <, <=, >, >=, !, &&, ||, ?:
(bits)          &, |, ^, ~, <<, >>, >>>
(conversión)    (tipo)
```

## Sentencias

Las sentencias de control del flujo de ejecución del programa en Java son las mismas que en C y C++:

Condicional simple.

```
if (condicion) {
  instrucciones;
}
else {
  instrucciones;
}
```

Ejemplo:

```
if (a != 0) {
  System.out.println("x = " + -a/b);
}
else {
  System.out.println("Error");
}
```

Condicional compuesta:

```
switch (expresión) {
  case <valor>: instrucciones;
                [break;]
  case <valor>: instrucciones;
                [break;]
    ...
  [default: instrucciones;]
}
```

Ejemplo:

```
switch (opcion) {
  case 1: x = x * Math.sqrt(y);
          break;
  case 2:
  case 3: x = x / Math.log(y);
          break;
  default: System.out.println("Error");
}
```

Iterativa de condición inicial:

```
while (condición) {
  instrucciones;
}
```

Ejemplo:

```
i = 0;
while (i < 10) {
  System.out.println(v[i]);
  i++;
}
```

Iterativa de condición final:

```
do {
  instrucciones;
} while (condición)
```

Ejemplo:

```
i = 0;
do {
  System.out.println(v[i]);
  i++;
} while (i<10)
```

Repetitiva:

```
for (inicialización; comparación; incremento) {
  instrucciones;
}
```

Ejemplo:

```
for (i = 0; i < 10; i++) {
  System.out.println(v[i]);
}
```

Repetitiva para colecciones (a partir de la versión 1.5.0 de Java):

```
for (tipo_elemento variable : colección) {
  instrucciones;
}
```

Ejemplo:

```
for (double x : v) {
  System.out.println(x);
}
```

Otras:

```
break;
continue;
label:
return <valor>;
exit;
```

## *Librerías*

Para crear clases dentro de librerías, a inicio de fichero se debe escribir la librería dónde se insertará la clase:

```
package carpeta.subcarpeta....;
```

Para indicar que un programa utiliza código que se encuentra en otro fichero, se usa la palabra `import`:

```
import carpeta.subcarpeta.....clase;
```

Ejemplo:

```
import java.lang.Math;
import java.io.*;
```

Librerías más comunes:

```
java.lang       Clases del lenguaje
java.awt        Clases para definir interfaces gráficas
java.io         Clases para la entrada/salida de datos
java.net        Clases para la comunicación por red
java.applet     Clases para la declaración de applets
java.util       Clases diversas
```

Otras librerías:

| Package | Description |
|---|---|
| `java.applet` | Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet |

| | |
|---|---|
| | context. |
| java.awt | Contains all of the classes for creating user interfaces and for painting graphics and images. |
| java.beans | Contains classes related to developing beans -- components based on the JavaBeans architecture. |
| java.io | Provides for system input and output through data streams, serialization and the file system. |
| java.lang | Provides classes that are fundamental to the design of the Java programming language. |
| java.math | Provides classes for performing arbitrary-precision integer arithmetic and arbitrary-precision decimal arithmetic. |
| java.net | Provides the classes for implementing networking applications. |
| java.nio | Defines buffers, which are containers for data, and provides an overview of the other NIO packages. |
| java.rmi | Provides the RMI package. |
| java.security | Provides the classes and interfaces for the security framework. |
| java.sql | Provides the API for accessing and processing data stored in a data source (usually a relational database). |
| java.text | Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages. |
| java.util | Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array, zip&gzip file compression). |
| javax.accessibility | Defines a contract between user-interface components and an assistive technology that provides access to those components. |
| javax.crypto | Provides the classes and interfaces for cryptographic operations. |
| javax.imageio | The main package of the Java Image I/O API. |
| javax.management | Provides the core classes for the Java Management Extensions. |
| javax.naming | Provides the classes and interfaces for accessing naming services. |
| javax.net | Provides classes for networking applications. |
| javax.print | Provides the principal classes and interfaces for the Java TM Print Service API. |
| javax.rmi | Contains user APIs for RMI-IIOP. |
| javax.security.auth | This package provides a framework for authentication and authorization. |
| javax.sound.midi | Provides interfaces and classes for I/O, sequencing, and synthesis of MIDI (Musical Instrument Digital Interface) data. |
| javax.sound.sampled | Provides interfaces and classes for capture, processing, and playback of sampled audio data. |
| javax.sql | Provides the API for server side data source access and processing. |
| javax.swing | Provides a set of "lightweight" components that, to the maximum degree possible, work the same on all platforms. |
| javax.xml | Defines core XML constants and functionality from the XML specifications. |
| org.ietf.jgss | This package presents a framework that allows application developers to make use of security services like authentication, data integrity and data confidentiality from a variety of underlying security mechanisms like Kerberos, using a unified API. |
| org.omg.CORBA | Provides the mapping of the OMG CORBA APIs to the Java TM programming language, including the class ORB , which is implemented so that a programmer can use it as a fully-functional Object Request Broker (ORB). |
| org.w3c.dom | Provides the interfaces for the Document Object Model (DOM) which is a component API of the Java API for XML Processing. |
| org.xml.sax | This package provides the core SAX APIs. |

## *Entrada y salida por pantalla*

Por teclado tan sólo se pueden leer líneas de texto, que después deben ser convertidas al tipo correspondiente:

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String s = in.readLine();
int i = Integer.parseInt(in.readLine());
```

Pero a partir de la versión 1.5.0 de Java se puede leer de teclado más fácil con la clase `java.util.Scanner`:

```
Scanner in = new Scanner(System.in);
String s = in.next();
int i = in.nextInt();
in.close();
```

Para imprimir por cónsola concatenamos cadenas de caràcteres y variables en un único String:

```
System.out.println("Hola " + s + i);
```

Pero a partir de la versión 1.5.0 de Java se puede escribir formateado al estilo de C (`%n` es el salto de línea):

```
System.out.printf("Hola %s %5d%n", s, i);
```

## *Entrada y salida por ficheros*

Existen multitud de clases para trabajar con entrada y salida de datos. Unas clases son para acceder directamente a los dispositivos (teclado, pantalla, ficheros, ...) y otras se enlazan a ellas para filtrar, acumular o comprimir los datos.

| Byte Input Stream | Description |
|---|---|
| `BufferedInputStream` | Reads a buffer of bytes from an `InputStream`, and then returns bytes from the buffer, making small reads more efficient. |
| `ByteArrayInputStream` | Reads bytes sequentially from an array. |
| `CheckedInputStream` | This `java.util.zip` class computes a checksum of the bytes it reads from an `InputStream`. |
| `DataInputStream` | Reads binary representations of Java primitive types from an `InputStream`. |
| `FileInputStream` | Reads bytes sequentially from a file. |
| `FilterInputStream` | The superclass of byte input stream filter classes. |
| `GZIPInputStream` | This `java.util.zip` class uncompresses GZIP-compressed bytes it reads from an `InputStream`. |
| `InflaterInputStream` | The superclass of `GZIPInputStream` and `ZipInputStream`. |
| `InputStream` | The superclass of all byte input streams. |
| `LineNumberInputStream` | This class is deprecated as of Java 1.1; use `LineNumberReader` instead. |
| `ObjectInputStream` | Reads binary representations of Java objects and primitive values from a byte stream. This class is used for the deserialization of objects. |
| `PipedInputStream` | Reads bytes written to the `PipedOutputStream` to which it is connected. Used in multithreaded programs. |
| `PushbackInputStream` | Adds a fixed-size pushback buffer to an input stream, so that bytes can be unread. Useful with some parsers. |
| `SequenceInputStream` | Reads bytes sequentially from two or more input streams, as if they were a single stream. |
| `StringBufferInputStream` | This class is deprecated as of Java 1.1; use `StringReader` instead. |
| `ZipInputStream` | This `java.util.zip` class uncompresses entries in a ZIP file. |
| Character (unicode) Input Stream | Description |
| `BufferedReader` | Reads a buffer of characters from a `Reader`, and then returns characters from the buffer, making small reads more efficient. |
| `CharArrayReader` | Reads characters sequentially from an array. |
| `FileReader` | Reads characters sequentially from a file. An `InputStreamReader` subclass that reads from an automatically-created `FileInputStream`. |
| `FilterReader` | The superclass of character input stream filter classes. |
| `InputStreamReader` | Reads characters from a byte input stream. Converts bytes to characters using the encoding of the default locale, or a specified encoding. |
| `LineNumberReader` | Reads lines of text and keeps track of how many have been read. |
| `PipedReader` | Reads characters written to the `PipedWriter` to which it is connected. Used in multithreaded programs. |
| `PushbackReader` | Adds a fixed-size pushback buffer to a `Reader`, so that characters can be unread. Useful with some parsers. |
| `Reader` | The superclass of all character input streams. |
| `StringReader` | Reads characters sequentially from a string. |
| Byte Output Stream | Description |
| `BufferedOutputStream` | Buffers byte output for efficiency; writes to an `OutputStream` only when the buffer fills up. |
| `ByteArrayOutputStream` | Writes bytes sequentially into an array. |
| `CheckedOutputStream` | This `java.util.zip` class computes a checksum of the bytes it writes to an `OutputStream`. |
| `DataOutputStream` | Writes binary representations of Java primitive types to an `OutputStream`. |
| `DeflaterOutputStream` | The superclass of `GZIPOutputStream` and `ZipOutputStream`. |
| `FileOutputStream` | Writes bytes sequentially to a file. |
| `FilterOutputStream` | The superclass of all byte output stream filters. |
| `GZIPOutputStream` | This `java.util.zip` class outputs a GZIP-compressed version of the bytes written to it. |
| `ObjectOutputStream` | Writes binary representations of Java objects and primitive values to an `OutputStream`. Used for the serialization of objects. |
| `OutputStream` | The superclass of all byte output streams. |
| `PipedOutputStream` | Writes bytes to the `PipedInputStream` to which it is connected. Used in multithreaded programs. |
| `PrintStream` | Writes a textual representation of Java objects and primitive values. Deprecated except for use by the standard output stream `System.out` as of Java 1.1. In other contexts, use `PrintWriter` instead. |
| `ZipOutputStream` | This `java.util.zip` class compresses entries in a ZIP file. |
| Character (unicode) Output Stream | Description |
| `BufferedWriter` | Buffers output for efficiency; writes characters to a `Writer` only when the buffer fills up. |
| `CharArrayWriter` | Writes characters sequentially into an array. |
| `FileWriter` | Writes characters sequentially to a file. A subclass of `OutputStreamWriter` that automatically creates a `FileOutputStream`. |
| `FilterWriter` | The superclass of all character output stream filters. |
| `OutputStreamWriter` | Writes characters to a byte output stream. Converts characters to bytes using the encoding of the default locale, or a specified encoding. |
| `PipedWriter` | Writes characters to the `PipedReader` to which it is connected. Used in multithreaded programs. |
| `PrintWriter` | Writes textual representations of Java objects and primitive values to a `Writer`. |
| `StringWriter` | Writes characters sequentially into an internally-created `StringBuffer`. |
| `Writer` | The superclass of all character output streams. |

Trabajar con ficheros binarios de datos “crudos” (imágenes, ...):

```
int c;
FileInputStream  fitxer1 = new FileInputStream("entrada");
FileOutputStream fitxer2 = new FileOutputStream("salida");
while ((c = fitxer1.read()) != -1) {
  fitxer2.write(c);
}
fitxer1.close();
fitxer2.close();
```

Trabajar con ficheros binarios de tipos simples (int, float, ...) y acceso directo:

```
// Sin acceso directo también podemos con DataInputStream y DataOutputStream
RandomAccessFile fitxer1 = new RandomAccessFile("radis", "r");
RandomAccessFile fitxer2 = new RandomAccessFile("perimetres", "w");
try {
  while (true) {
    fitxer2.writeFloat( 2 * 3.1416 * fitxer1.readFloat() );
}
catch (EOFException e) {
  fitxer1.close();
  fitxer2.close();
}
```

Trabajar con ficheros de texto:

```
String salari;
FileReader fitxer1 = new FileReader("salarios.txt");
FileWriter fitxer2 = new FileWriter("salarios.new");
BufferedReader in  = new BufferedReader(fitxer1);
BufferedWriter out = new BufferedWriter(fitxer2);
while ((salari = in.readLine()) != null) {
  salari = (new Integer(Integer.parseInt(salari)*10).toString();
  out.write(salari);
fitxer1.close();
fitxer2.close();
```

Guardar y recuperar objetos (serialización):

```
class Alumne implements Serializable {
  private String nom;
  private String id;
  private int edad;
  private transient double nota;
  ...
}

class Grup implements Serializable {
  private Alumne [] alumnes = new Alumne[20];
  ...
}

Grup a1 = new Grup();
a1.afegir(new Alumne("Pepe", "43510987F", 25, 8.5));
a1.afegir(new Alumne("Pepa", "65471234H", 18, 9.3));
...
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("grupA1"));
out.writeObject(a1);
out.close();

Grup a2 = new Grup();
ObjectInputStream in = new ObjectInputStream(new FileInputStream("grupA1"));
a2 = (Grup) in.readObject();
in.close();
```

Obtener información de ficheros y directorios:

Los objetos de la clase `File` tienen multitud de métodos para extraer cualquier tipo de información: `canRead`, `canWrite`, `exists`, `isHidden`, `isDirectory`, `isFile`, `createNewFile`, `createTempFile`, `delete`, `renameTo`, `mkdir`, `mkdirs`, `setReadOnly`, `getName`, `getPath`, `toURL`, `length`, `lastModified`, `list`, `listFiles`, ....

```
File raiz = new File("\\");
String [] dir = raiz.list();
for (i = 0; i < dir.length; i++)
  System.out.println(dir[i]);
```

## *Programa principal y argumentos*

Programa principal en una clase Java:

```
class <Nombre>
{
  public static void main(String[] args)
  {
     instrucciones;
  }
}
```

Ejemplo de programa principal en una clase Java:

```
// Imprime una palabra (primer argumento)
// un número determinado de veces (segundo argumento)
class Mensaje {
  public static void main(String [] args) {
    if (args.lenght == 2) {
      for (int i = 1; i <= Integer.parseInt(args[1]); i++) {
        System.out.println(args[0] + " : " + i);
      }
    }
  }
}
```

Programa principal en un applet Java:

```
class <Nombre> extends Applet
{
  public void init()
  {
     instrucciones;
  }
}
```

Ejemplo de programa principal en un applet Java:

```
import java.awt.*;
import java.applet.*;

public class HolaMundoApplet extends Applet {
  public void init() {
    repaint();
  }

  public void paint(Graphics g) {
    g.drawString("Hola Mundo!", 25, 25);
  }
}
```

Insertar en una página web HTML el applet:

```
<html>
  <body>
    <applet code="HolaMundoApplet" width=300 height=50></applet>
  </body>
<html>
```

## *Clases*

```
class <Nombre> [extends <Nombre_clase_padre>]
{
  // declaración de atributos
  visibilidad [modificadores] tipo atributo1 [= valor];
  visibilidad [modificadores] tipo atributo2 [= valor];

  ...
  // declaración de métodos
  visibilidad [modificadores] tipo metodo1(argumentos) {
    instrucciones;
  }
  visibilidad [modificadores] tipo metodo2(argumentos) {
    instrucciones;
  }

  ...
}
```

Donde:

```
visibilidad = public, protected o private
modificadores = final, static y abstract
argumentos = declaración de variables separadas por comas
```

Ejemplo:

```
class Complejo
{
  private double re, im;

  public Complejo(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public String toString() {
    return(new String(re + "+" + im + "i"));
  }

  public boolean equals(Complejo v) {
    return((re == v.re) && (im == v.im));
  }

  public double modul() {
    return(Math.sqrt(re*re + im*im));
  }

  public void suma(Complejo v) {
    re = re + v.re;
    im = im + v.im;
  }

}
```

Aviso: a partir de la versión 1.5.0 de Java podemos encontrar argumentos variables en número.

```
visibilidad [modificadores] tipo metodo(Object ... args) {
  instrucciones;
}
```

Ejemplo:

```
public void suma(Complejo ... args) {
  for (int i=0; i<args.length; i++) {
    re = re + args[i].re;
    im = im + args[i].im;
  }
}
```

## *Objetos (como si declaráramos variables de tipo compuesto)*

```
<NombreClasse> <NombreObjeto>;                          // declaración
<NombreObjeto> = new <NombreClase>(inicialización);     // instanciación
...
trabajar con NombreObjeto.atributo;
llamar a NombreObjeto.metodo(argumentos);
```

Ejemplo:

```
Complejo z, w;
z = new Complejo(-1.5, 3.0);
w = new Complejo(-1.2, 2.4);
z.suma(w);
System.out.println("Complejo: " + z.toString());
System.out.println("Modulo: " + z.modul());
```

## *Más sobre clases y objetos: sobrecarga, constructores, this, herencia, super, instanceof, conversión, visibilidad, final, abstract, static, equals y clone, polimorfismo*

Sobrecarga:

Denominamos sobrecarga a disponer de dos o más métodos con el mismo nombre dentro de la misma clase. ¿Cuál ejecutar en la llamada al método? Los diferenciaremos por los parámetros de la llamada.

```
class nombre_clase {

  public tipo_retorno nombre_método(parámetros) {
    código;
  }

  public tipo_retorno nombre_método(otros parámetros) {
    otro código;
  }

  ...
}
```

Ejemplo:

```
class Complejo {
  private double re, im;

  public Complejo(double r, double i) {
    re = r;
    im = i;
  }

  public Complejo sumar(Complejo c) {
    return new Complejo(re + c.re, im + c.im);
  }

  public Complejo sumar(double r, double i) {
```

```
    return new Complejo(re + r, im + i);
  }

  public String toString() {
    return re + " + " + im + "i";
  }
}

Complejo c1 = new Complejo(1, 3);
Complejo c2 = new Complejo(-4, 3.5);
c2 = c1.sumar(c2);
c2 = c2.sumar(0.5, -4);
System.out.println(c1 + "\n" + c2 + "\n");
```

Constructor:

Es un método que se ejecuta automáticamente cuando se instancia un objeto de una clase. Dicho método se distingue porque tiene el mismo nombre que la clase.

Una clase puede tener más de un método constructor (sobrecarga). Según los parámetros de la instanciación se llamará a uno u otro.

Por defecto toda clase tiene un constructor sin parámetros y sin código, que desaparece una vez se escribe un método constructor para dicha clase.

Por defecto, antes de ejecutarse todo constructor llama al código del constructor sin parámetros de la clase padre. Podemos cambiar dicho comportamiento (escoger qué constructor de la clase padre se llamará) escribiendo como primera linea de código del constructor: `super(parámetros)`.

```
class nombre_clase {

  public nombre_classe(parámetros) {
    código;
  }

  public nombre_classe(otros parámetros) {
    código;
  }

  ...
}

nombre_clase objeto = new nombre_clase(parámetros_constructor);
```

Ejemplo:

```
class Complejo {
  private double re, im;

  public Complejo(double r, double i) {
    re = r;
    im = i;
  }
}
```

Ejemplo que imprimirá por pantalla AB:

```
class A {
  public A() {
    System.out.print("A");
  }
}

class B extends A {
  public B() {
    System.out.print("B");
  }
}

class Principal {
  public static void main(String[] args) {
    B b = new B();
```

```
  }
}
```

`this`:

Es un término que se utiliza como referencia del objeto que está ejecutando el método. Nos será útil para diferenciar los atributos de la clase de los parámetros, cuando éstos tengan el mismo nombre, y también cuando debamos llamar a un método pasando como referencia el actual objeto que está ejecutando el código.

Ejemplo 1:

```
class Complejo {
  private double re, im;

  public Complejo(double re, double im) {
    this.re = re;   // this.re es el atributo, y re el parámetro
    this.im = im;   // this.im es el atributo, y im el parámetro
  }
}
```

Ejemplo 2:

```
class Actor {
  private Vector pelicules;

  public void inclourePelicula(Pelicula p) {
    pelicules.add(p);
    p.incloureActor(this);
  }
}
```

Herencia:

Decimos que una clase hereda o deriva de otra, cuando automáticamente pasa a disponer de todos los métodos y atributos de esta otra como si fueran propios. La clase que hereda se llama "hija", y la clase de la cual hereda se llama "padre". Por defecto, toda clase hereda de la clase `Object`, a menos que especifiquemos que hereda de otra clase mediante la palabra `extends`:

```
class clase_hija extends clase_padre {
  ...
}
```

La clase hija puede redefinir los métodos heredados de la clase padre, escribiendo de nuevo el código. Desde el código de un método de la clase hija se puede llamar al código de un método de la clase padre mediante la palabra `super`.

```
tipo_retorno metodo_clase_hija(argumentos) {
  ...
  ... super. metodo_clase_padre(argumentos);
  ...
  }
```

Un objeto de la clase hija es del tipo de la clase hija, pero también del tipo de la clase padre y todos sus antecesores. Podemos preguntar por el tipo de un objeto con el operador booleano `instanceof`, y cambiar su tipo escribiendo previamente su nuevo tipo entre paréntesis:

```
objeto instanceof nombre_clase

(nueva_clase) objeto
```

Una clase sólo puede heredar de otra clase y de varias interfícies. En Java no existe la herencia múltiple.

Ejemplo de sobrecarga, constructor, herencia, `this` y `super`:

```
class Persona {
  private String nombre;
  private int edad;

  public Persona() {
  }

  public Persona(String nombre, int edad) {
```

```
    this.nombre = nombre;
    this.edad = edad;
  }

  public String toString() {
    return "Nombre: " + nombre + "\n" + "Edad: " + edad + "\n";
  }
}

class Alumno extends Persona {
  private double nota;

  public Alumno(String nombre, int edad, double nota) {
    super(nombre, edad);
    this.nota = nota;
  }

  public String toString() {
    return super.toString() + "Nota: " + nota + "\n";
  }
}

Persona p1 = new Persona();
Persona p2 = new Persona("Alex", 22);
Alumno a1 = new Alumno("Pepe", 20, 8.5);
System.out.println(p1 + "\n" + p2 + "\n" + a1);
System.out.println(p2 instanceof Alumno);
System.out.println(a1 instanceof Persona);
System.out.println((Persona) a1);
```

Visibilidad:

La accesibilidad de los métodos y atributos de una clase viene declarado por las palabras `public` (son accesibles por cualquier clase), `private` (sólo son accesibles por la clase que los ha declarado) y `protected` (sólo són accesibles por la clase que los ha declarado y por sus clases hijas o descendientes).

`final`:

Término que se utiliza para declarar una constante (cuando lo encontramos delante de un atributo), un método que no se podrá redefinir (cuando lo encontramos delante de un método), o una clase de la que ya no se podrá heredar (cuando lo encontramos delante de una clase).

`abstract`:

Término que denota un método del cual no se escribirá código. Las clases con métodos abstractos no se pueden instanciar, y sus clases herederas deberán escribir el código de sus métodos abstractos si queremos crear alguna instancia suya.

`static`:

Término que se aplica a los atributos y métodos de una clase que pueden utilizarse sin crear un objeto que instancie dicha clase. El valor de un atributo estático, además, es compartido por todos los objetos de dicha clase.

Ejemplo:

```
class Persona {

  private String nombre;
  public static int num = 0;

  public Persona(String nombre) {
    this.nombre = nombre;
    num++;
  }

  public static int cuantos() {
```

```
    return num;
  }

  public void finalize() throws Throwable {
    num--;
    super.finalize();
  }
}

class C {
  public static void main(String[] args) {
    Persona p1 = new Persona("Maria");
    Persona p2 = new Persona("Alex");
    System.out.println(Persona.cuantos());
  }
}
```

`equals` y ==, `clone` y =:

El nombre de una variable de tipo simple no es más que otra forma de llamar a la dirección de memoria que contiene el valor de la variable (referencia directa). Podemos comparar valores de variables con el operador ==, y asignar un valor a una variable con el operador =.

En cambio, el nombre de un objeto de una clase no contiene los valores de los atributos, sino la posición de memoria donde residen dichos valores de los atributos (referencia indirecta). Utilizaremos el operador == para saber si dos objetos ocupan la misma posición de memoria (son el mismo objeto), mientras que utilizaremos el método `equals` para saber si sus atributos tienen los mismos valores. Utilizaremos el operador = para asignar a un objeto otro objeto que ya existe (serán el mismo objeto), mientras que utilizaremos el método `clone` para crear una copia de un objeto determinado y asignarla a otro.

Ejemplo:

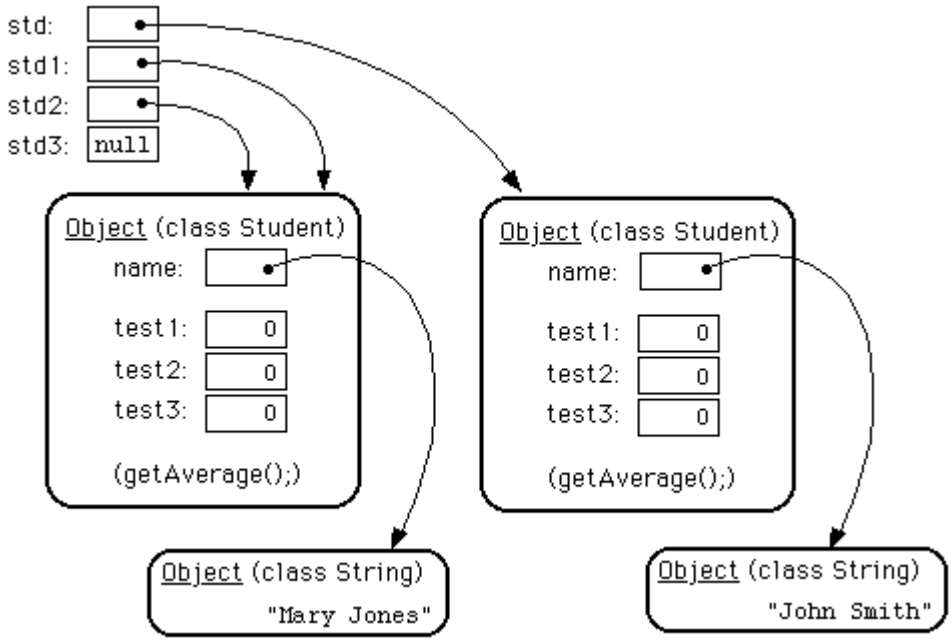
```
class Student {

  public String name;
  public double test1, test2, test3;

  public double getAverage() {
    return (test1 + test2 + test3) / 3;
  }
}

...

Student std, std1, std2, std3;
std = new Student();
std1 = new Student();
std2 = std1;
std3 = null;
std.name = "John Smith";
std1.name = "Mary Jones";
```




Polimorfismo:

Se trata de declarar un objeto de una clase, pero instanciarlo como un descendiente de dicha clase (lo contrario no es posible):

```
clase_padre objeto = new clase_descendiente(parámetros_constructor);
```

Ejemplo:

```
class Complex {

  double re, im;

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  // Lo siguiente también podría ser "abstract public void imprimir();"

  public void imprimir() {
    System.out.println(re + " " + im);
  };
}


class Complex1 extends Complex {

  public Complex1(double re, double im) {
    super(re, im);
  }

  public void imprimir() {
    System.out.println(re + "+" + im + "i");
  }
}


class Complex2 extends Complex {

  public Complex2(double re, double im) {
    super(re, im);
  }

  public void imprimir() {
    System.out.println("(" + re + "," + im + ")");
```

```
  }
}


class Principal {

  public static void main(String[] args) {

    Complex v[] = new Complex[2];
    v[0] = new Complex1(5, 4);
    v[1] = new Complex2(1, 3);

    for (int i = 0; i < v.length; i++) {
      v[i].imprimir();
    }
  }
}
```

## *Interfaces*

Clase completamente abstracta. No tiene atributos y ninguno de sus métodos tiene código. (En Java no existe la herencia múltiple, pero una clase puede implementar una o más interfaces, adquiriendo sus tipos).

```
interface <Nombre_interface> [extends < Nombre_interface_padre>] {
  visibilidad [modificadores] tipo metodo1(argumentos);
  visibilidad [modificadores] tipo metodo2(argumentos);
  ...
}

class <Nombre_clase> extends <clase_padre> implements <interface1>, <interface2>, ...
{
  ...
}
```

Ejemplo: queremos crear una clase llamada `Vector`, cuyos elementos pueden ser de cualquier tipo, pero si no sabemos el tipo, ¿Cómo podemos ordenarlos con el método `ordenar` de la clase `Vector`? ¿Cómo podemos imprimirlos con el método `imprimir` de la clase `Vector`? Deben ser comparables entre ellos mediante algún método, y también imprimibles mediante un método.

```
interface Comparable {
  int compareTo(Object o);
}


interface Imprimible {
  String toString();
}


class miVector {

  Object elementos[];
  int num;

  public miVector2(int capacidad) {
    elementos = new Object[capacidad];
    num = 0;
  }

  public void añadir(Object o) {
    if (num < elementos.length)
      elementos[num++] = o;
  }

  public void ordenar() {
    Object aux;
    for (int i = 0; i < num-1; i++)
```

```
      for (int j = i+1; j < num; j++)
          if (((Comparable)elementos[i]).compareTo(elementos[j]) > 0) {
              aux = elementos[i];
          elementos[i] = elementos[j];
          elementos[j] = aux;
          }
  }

  public void imprimir() {
    for (int i = 0; i < num; i++)
      System.out.println((Imprimible)elementos[i]);
  }
}


class Persona {
  public String nom;
}


class Alumne extends Persona implements Comparable, Imprimible {

  public double nota;

  public int compareTo(Object o) {
    return nom.compareTo(((Alumne)o).nom);
  }

  public String toString() {
    return nom + " " + nota;
  }
}


class Principal {

  public static void main(String[] args) {

    miVector v = new miVector(5);

    Alumne a = new Alumne();
    a.nom = "Pepe";
    a.nota = 6.7;
    v.añadir(a);

    Alumne b = new Alumne();
    b.nom = "Pepa";
    b.nota = 7.6;
    v.añadir(b);

    v.ordenar();
    v.imprimir();
  }
}
```

## *Applets*

```
import java.awt.*;
import java.applet.*;

class <Nombre> extends Applet {

  // declaración de atributos
  visibilidad [modificadores] tipo atributo1 [= valor];
  visibilidad [modificadores] tipo atributo2 [= valor];
  ...
  // declaración de métodos
```

```
  visibilidad [modificadores] tipo metodo1(argumentos) {
    instrucciones;
  }

  public void init() {              // método de inicialización (principal)
    instrucciones;
  }

  public void start() {                    // método de activación (-> visible)
    instrucciones;
  }

  public void paint(Graphics g) {          // método de visualización (dibujar)
    instrucciones;
  }

  public void stop() {              // método de desactivación (-> invisible)
    instrucciones;
  }

  public void destroy() {                  // método de destrucción
    instrucciones;
  }

  ...
}
```

Insertar el applet en una página web (HTML):

```
<html>
  <body>
    ...
    <applet code="nombre_applet" atributos>
      <param nombre_parametro = "valor">
      <param nombre_parametro = "valor">
      ...
      código HTML si el navegador no puede ejecutar Java
    </applet>
  </body>
<html>
```

Ejemplo:

```
import java.awt.*;
import java.applet.*;

public class PruebaApplet extends Applet
{
  Image logo;
  AudioClip melodia;
  TextField cuadroTexto;

  public void init()
  {
    logo = getImage(getDocumentBase(), "imagenes/Logotipo.png");
    melodia = getAudioClip(getDocumentBase(), "Melodia.au");
    cuadroTexto = new TextField(20);
    cuadroTexto.setText("Aquí puedes escribir");
    add(cuadroTexto);
  }

  public void start()
  {
    melodia.loop();
  }

  public void paint(Graphics g)
  {
    g.drawImage(logo, 0, 0, this);
    g.drawString(cuadroTexto.getText(), 25, 25);
  }
```

```
  public void stop()
  {
    melodia.stop();
  }

  public void destroy()
  {

  }
}
```

HTML Applet:

```
<html>
  <body>
    <applet code="PruebaApplet" width=300 height=50 align=middle></applet>
  </body>
<html>
```

## *Excepciones*

Para gestionar la excepción:

```
try {
  código donde se pueden producir excepciones
}

catch (TipoExcepcion1 NombreExcepcion) {
  código a ejecutar si se produce una excepción del tipo TipoExcepcion1
}

catch (TipoExcepcion2 NombreExcepcion) {
  código a ejecutar si se produce una excepción del tipo TipoExcepcion1
}

...

finally {
  código a ejecutar tanto si se produce una excepción como si no
}
```

Ejemplo:

```
String salari;
BufferedReader fitxer1 = null;
BufferedWriter fitxer2 = null;
try {
  fitxer1 = new BufferedReader(new FileReader("c:\\salarios.txt"));
  fitxer2 = new BufferedWriter(new FileWriter("c:\\salarios.new"));
  while ((salari = fitxer1.readLine()) != null) {
    salari = (new Integer(Integer.parseInt(salari)*10).toString());
    fitxer2.write(salari+"\n");
  }
}
catch (IOException e) {
  System.err.println(e);
}
catch (NumberFormatException e) {
  System.err.println("No es un número");
}
finally {
  fitxer1.close();
  fitxer2.close();
}
```

Existen multitud de excepciones en Java, agrupadas por familias. Por ejemplo: `ArithmeticException`, `IOException`, `EOFException`, `FileNotFoundException`, `NullPointerException`, `NegativeArraySizeException`, `ArrayIndexOutOfBoundsException`, `SecurityException`, `NumberFormatException`, ...

También existen multitud de errores en Java, que son fallos de la máquina virtual que es mejor que no los gestione la aplicación. Por ejemplo: `OutOfMemoryError`, `InternalError`, `StackOverflowError`, `UnknownError`, `NoClassDefFoundError`, ...

Ejemplo de excepciones agrupadas por familias:

```
java.lang.Object
 └─ java.lang.Throwable
     ├─ java.lang.Exception
     │   ├─ java.lang.RuntimeException
     │   │   ├─ java.lang.IllegalArgumentException
     │   │   │   └─ java.lang.NumberFormatException
     │   │   ├─ java.lang.ArithmeticException
     │   │   └─ java.lang.IndexOutOfBoundsException
     │   │       ├─ java.lang.ArrayIndexOutOfBoundsException
     │   │       └─ java.lang.StringIndexOutOfBoundsException
     │   ├─ java.io.IOException
     │   │   ├─ java.io.EOFException
     │   │   ├─ java.io.FileNotFoundException
     │   │   ├─ java.net.MalformedURLException
     │   │   └─ java.net.SocketException
     │   │       ├─ java.net.PortUnreachableException
     │   │       ├─ java.net.NoRouteToHostException
     │   │       └─ ...
     │   └─ ...
     └─ java.lang.Error
         ├─ java.lang.VirtualMachineError
         │   ├─ java.lang.OutOfMemoryError
         │   ├─ java.lang.UnknownError
         │   └─ ...
         └─ ...
```

Para crear una nueva excepción:

```
public class NombreNuevaExcepcion extends NombreExcepcion {
  atributos y métodos
}
```

Para declarar que un método lanza excepciones:

```
visibilidad [modificadores] tipo metodo(argumentos) throws NombreExcepcion1,
NombreExcepcion2, ... {
  ...
  ... throw new NombreExcepcion1(parámetros);
  ...
  ... throw new NombreExcepcion2(parámetros);
  ...
}
```

Ejemplo:

```
class CoeficientZeroException extends ArithmeticException {
  public CoeficientZeroException(String mensaje) {
    super(mensaje);
  }
};


// clase ecuación de primer grado:  a x + b = 0
class Ecuacion {
```

```
  private double a, b;

  public Ecuacion(double coef1, double coef0) {
    a = coef1;
    b = coef0;
  }

  public double Raiz() throws CoeficientZeroException {
    if (a == 0)
      throw new CoeficientZeroException("La ecuación no es de primer grado");
    else
      return -b/a;
  }
}

class Principal {

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

    try {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      System.out.print("Introduzca el coeficiente del término de grado 1: ");
      double c1 = Integer.parseInt(in.readLine());
      System.out.print("Introduzca el coeficiente del término de grado 0: ");
      double c2 = Integer.parseInt(in.readLine());
      Ecuacion eq = new Ecuacion(c1, c2);
      System.out.println("La solución es : " + eq.Raiz());
    }
    catch (CoeficientZeroException e) {
      System.err.println(e);
    }
    catch (NumberFormatException e) {
      System.err.println("Número incorrecto ... " + e.getMessage());
    }
    catch (Exception e) {
      System.err.println("Excepción desconocida");
      throw e;
    }
  }
}
```

## *Próximamente: hilos de ejecución (programación concurrente)*

No es una característica del lenguaje, sino una serie de clases de la API de Java que permiten programación multihilo. (Ver clase Thread y métodos sincronizados)

## *Próximamente: comunicación a través de Internet (TCP/IP)*

No es una característica del lenguaje, sino una serie de clases de la API de Java que comunicación a través de redes. (Ver clases Socket, ServerSocket, MulticastSocket, DatagramPacket, DatagramSocket y URL)