Dpto. Física y Arquitectura de Computadores

Area de Arquitectura y Tecnología de Computadores

Apuntes de Java

P. Pablo Garrido Abenza

Introducción

Historia

En el año 1991, Sun Microsystems trabajaba en el desarrollo de software para la comunicación de diversos aparatos electrónicos (TV, vídeos, etc.). Se encontraron con el problema de que cada uno de ellos tenía una arquitectura diferente, por lo que desarrollaron un lenguaje de programación independiente de la arquitectura, denominado Oak.

Por aquella época aun no existía el lenguaje HTML. Con la explosión de Internet en 1994, se dieron cuenta de que Internet era una inmensa red mundial que conectaba entre si ordenadores con diferentes arquitecturas y sistemas operativos, y que su lenguaje de programación era facilmente aplicable a Internet. De hecho, Internet se ha convertido en su principal aplicación. Lo que hicieron fue incorporar su máquina virtual a un navegador llamado WebRunner, que después se denominó HotJava.

Finalmente, en Mayo de 1995, Sun Microsystems presentó oficialmente la primera versión del Lenguaje Java, en la conferencia SunWorld en San Francisco. Ahí mismo se anunció el lanzamiento de Netscape 2.0, y de que incorporaría la máquina virtual de Java. Muy poco después, Microsoft también integró la máquina virtual Java a su navegador Internet Explorer 3.0.

El nombre de Java se lo dio uno de sus desarrolladores, y viene de cómo se nombra el café en cheli americano.

Ventajas

Java es un lenguaje de programación de alto nivel, con varias **ventajas** frente a otros lenguajes:

- Es **independiente de la plataforma** (arquitectura hardware + sistema operativo), siempre y cuando que exista una "Máquina Virtual de Java" implementada para dicha plataforma. Es independiente tanto a nivel de código fuente como de código binario. C++ sólo es independiente a nivel de código fuente, necesitando recompilar el código con el compilador adecuado a la plataforma. Más adelante veremos las diferencias existentes entre C++ y Java.
- Es un lenguaje de programación **orientado a objetos** (POO), muy flexible.
- Su **sintáxis** es muy parecida a la de **C++**, añadiendo ciertas características nuevas (enlace dinámico, nuevos tipos de datos) y eliminando otras (punteros y gestión de memoria), consiguiendo un lenguaje más sencillo. Esto facilita su aprendizaje a los programadores que ya conozcan C/C++.
- Podemos escribir **aplicaciones convencionales** (ejecutables de forma autónoma) y además, **aplicaciones de Internet ó applets** (aplicaciones incluidas en páginas Web, ejecutables desde los navegadores de Internet).

Inconvenientes

- Debido a que se trata de un lenguaje interpretado en la fase de ejecución, la **velocidad de ejecución** de los programas Java es menor que en otros lenguajes, aunque no tanto como los lenguajes totalmente interpretados; Java es mitad compilado y mitad interpretado.
- Si no se conoce C++, aprender Java puede ser largo, pues es un lenguaje complejo, aunque más sencillo que C++, por ejemplo.

Evolución de la programación

Java es un lenguaje de programación orientado a objetos, es decir, utiliza una serie de técnicas para garantizar que los programas realizados sean fiables y fáciles de mantener.

La programación ha evolucionado mucho en los pocos años que tiene de historia (apenas unos cincuenta años).

Tecnología	Características	Lenguajes
Programación lineal (años 50)	• El orden de ejecución se hacía con **saltos** a número de línea concretos (GOTO) – Programación tipo "espaguetti". • Todas las variables eran **globales** (no locales), por lo que cualquier cambio puede afectar a todo el programa. • La **reutilización de código** era muy difícil, por lo que continuamente se tenía que duplicar una y otra vez las mismas líneas de código.	• BASIC • COBOL • FORTRAN
Programación modular (años 60)	• El programa se divide de forma descendente (top-down) en pequeñas tareas desarrolladas de forma independiente, que al final se ensamblan y forman el programa completo: **subrutinas**. Estas subrutinas actúan como una "caja negra", pues reciben unos datos (entrada) y producen unos resultados (salida), sin importar ¿Cómo lo hace?, tan sólo ¿Qué hace?. • Las subrutinas tienen un nombre, lo cual facilita la **reutilización de código**. • **Varios programadores** pueden trabajar de forma simultánea, cada uno con un módulo, y los cambios realizados por uno de ellos en su módulo no afectan al resto.	• MODULA2

Programación estructurada (finales de los 60)	• El programa tiene un diseño **modular** y descendente (top-down). • Se utilizan **estructuras de control** para controlar el orden de ejecución, para evitar los saltos (GOTO): selección, bucles (repetición), secuencia. • Los programas son más fáciles de escribir, entender y mantener. • El programador puede crearse nuevos tipos de datos, los **Tipos Abstractos de Datos (TAD)**, los cuales tienen un conjunto de operaciones implementados con funciones. • Al tener separados el **código** y los **datos**, al trabajar varios programadores se deben comunicar continuamente los cambios realizados en los datos.	• PASCAL • C • Ada
Programación Orientada a Objetos (POO) (Desarrollado desde los 70, pero no difundido hasta los 90).	• Se trata de una mejora a la programación estructurada, en modularidad, reutilización de código y trabajo en equipo. • Se basa en el uso de **objetos**, que son parecidos a los TAD, aunque con nuevas características: propiedades, herencia, encapsulación (ocultación), polimorfismo, etc. • Los objetos contienen los datos, por lo que al dividir el programa en módulos (objetos), éstos contendrán las subrutinas (métodos) y los datos (propiedades o atributos).	• Object Pascal • C++ • Delphi • Java

Comparación de C++ con Java

- Básicamente tienen la misma sintáxis y tipos de datos, aunque Java incorpora otros nuevos.
- C y C++ realizan conversiones automáticas de tipos cuando asignamos valores de un tipo a otra de variable de distinto tipo. Con Java, para mantener la integridad de los datos esto está más restringido. Java realiza conversiones entre tipos de datos enteros.
- En C / C++, los bytes que utiliza cada tipo de datos puede variar de una plataforma a otra. En Java no ocurre esto, siempre ocupan los mismos bytes.
- Con C++ es posible programar sin utilizar la **metodología orientada a objetos**, convirtiéndose en este caso el compilador de C++ en un compilador de C estándar. Con Java siempre se utiliza la programación orientada a objetos.
- En Java no existen los **punteros** ni la gestión manual de memoria. Los objetos se liberan automáticamente cuando ya no se necesitan (la ejecución sale de su ámbito de visibilidad).

JDK

Java se compone de varios **elementos**:

1. Editor de textos o entorno de desarrollo (opcional).
2. Compilador.
3. Máquina virtual de Java.
4. Otras utilidades

Editor de textos o entorno de desarrollo.

Al **escribir programas Java**, podemos utilizar un editor de texto puro (como el Notepad), procesadores de texto (como el WordPad), o entornos de desarrollo Java o IDEs (como Forte de Sun, Visual Café de Symantec, Visual J++ de Microsoft, JBuilder de Borland, pcGRASP, RealJ (antes FreeJava), etc.). Algunos de estos IDE's pueden incorporar su propio compilador, o bien, utilizar el mismo del JDK de Sun, por lo que deberemos tenerlo instalado previamente.

En cualquier caso, el fichero no debe contener información de formato y la extensión del fichero debe ser ***.java*** (Por ejemplo, *HolaMundo.java*). Si utilizamos como editor de texto el Notepad, al grabar nos añadirá al nombre del fichero la extensión .TXT automáticamente. Deberemos cambiarle la extensión posteriormente. En UNIX/Linux podremos utilizar los editores "Pico", "Vi" ó "Emacs".

Compilador

Traduce el programa fuente que hemos escrito en leguaje Java a un código de bytes que no es el código máquina para un ordenador concreto, como hacen todos los compiladores, sino el de una máquina ficticia. Por tanto, para ejecutarlo no lo podremos hacer directamente, sino que se necesita la "Máquina virtual de Java" apropiada a la plataforma de ejecución, la cual no tiene por qué ser la misma que la utilizada durante la compilación.

La extensión del fichero generado por el compilador con el código de bytes es ***.class***.

Si utilizamos el Java Developer's Kit (JKD) de Sun Microsystems, invocaremos al compilador desde la línea de órdenes del DOS de la siguiente forma:

C:> javac HolaMundo.java

Si no se producen errores, se nos generará el fichero: *HolaMundo.class*

Máquina Virtual de Java

Una máquina virtual es un programa que simula una máquina ficticia, que nos proporciona independencia de la plataforma a la hora de ejecutar los programas escritos en Java.
Se trata de un **intérprete**, que lee los códigos de bytes del fichero compilado, traduce el código máquina de la máquina ficticia al código máquina de la plataforma concreta y ejecuta las instrucciones. Si se interpretase directamente el código fuente en lugar el fichero compilado, la velocidad de ejecución sería muy baja. Al utilizar el código de bytes se consigue casi igualar la velocidad de ejecución de otros lenguajes compilados. Actualmente se está trabajando en compiladores JIT (Just In Time o compiladores al instante).

El programa compilado es siempre el mismo en todas las plataformas, lo que cambia es la máquina virtual, que es la que tiene en cuenta las características concretas de las diferentes arquitecturas hardware y sistemas operativos.

Para ejecutar un programa invocaremos desde la línea de órdenes del DOS a la máquina virtual, especificando el fichero de código de bytes a ejecutar, pero sin la extensión *.class*:

C:> java HolaMundo

NOTA: Debemos escribir el nombre del fichero tal cual, respetando las mayúsculas y minúsculas. Si no lo hacemos, Java nos mostrará un error diciendo que no encuentra la clase. Si en el ejemplo anterior hubiésemos puesto:

C:> java holamundo

nos aparecería el siguiente mensaje:

Exception in thread "main" java.lang.NoClassDefFoundError: holamundo (wrong name: HolaMundo).

Si el programa es en modo texto, la salida de dicho programa será la propia ventana de consola de MS-DOS desde la que hemos ejecutado el programa. Además, no se retorna el control hasta que el programa termine.

Existe otra posibilidad, pero sólo para aplicaciones en modo gráfico, ejecutando:

C:> javaw HolaMundo

La diferencia será que se nos devuelve el control inmediatamente (prompt DOS) y se nos abrirá una ventana, la cual será la salida del programa.

Este intérprete tiene muchas **opciones**. La más interesante es el "profiler" u optimizador. Simplemente poniendo "-prof" detrás de "java", nos generará java.prof, donde se muestra la cantidad de veces que se ha invocado cada uno de los métodos y los milisegundos que se ha empleado en ello:

java –prof HolaMundo

Búsqueda de clases: CLASSPATH

Java puede encontrar clases en el directorio actual desde donde intentamos ejecutar una aplicación. En el caso de que las clases especificadas estén en otro directorio debemos establecer la variable de ambiente CLASSPATH. En dicha variable indicamos al compilador todas las rutas donde debe buscar paquetes de clases, separadas por ';'. También podremos especificar nombres de ficheros con la extensión ".class", ".jar" y ".zip".

La forma de **establecer la variable de ambiente** CLASSPATH depende del sistema operativo en el que vayamos a trabajar:

- DOS y Windows 9x:
  En el fichero Autoexec.bat, debemos tener una línea:
  SET CLASSPATH = <Lista de directorios y archivos>
  Después de hacer algún cambio deberemos reiniciar el sistema.
- Windows NT:
  Desde el Panel de Control => Propiedades del sistema => Solapa "Entorno". Si la variable CLASSPATH no existe la creamos; en otro caso, la seleccionamos y modificamos su contenido.
- UNIX/Linux:
  unset CLASSPATH
  setenv CLASSPATH path1:path2... (Sep. por ':').

Para **consultar el valor** de esta variable de ambiente:

- DOS, Windows 9x, Windows NT:
  Podremos ejecutar el comandos SET del DOS, y nos aparecerá un listado con todas las variables de ambiente establecidas y su valor actual.
- UNIX/Linux:
  Ejecutar SET ó echo $CLASSPATH

Por **ejemplo**, suponiendo que estamos en Windows 98 y nuestras clases las agrupamos en un paquete llamado 'libreria', el directorio raíz deberá ser c:\java\jdk1.3\libreria, y la variable CLASSPATH deberá tener:

SET **CLASSPATH** = . ; C:\JDK1.3 ; C:\JDK1.3\librería

Una alternativa a esta variable de ambiente sería el uso del parámetro "**-classpath"** a la hora de invocar al compilador (javac), al intérprete de java (java) y a otras utilidades del JDK de Sun. En dicho parámetro, especificaríamos una serie de rutas separadas por ';'. Este método tiene la ventaja de que para cada proyecto particular podremos tener unas librerías concretas; con la variable de ambiente CLASSPATH, cualquier cambio afectaría todos los proyectos existentes.

La primera máquina virtual la desarrolló Sun Microsystems, creador y propietario del lenguaje Java y el entorno de desarrollo JDK (Java Develover's Kit). Posteriormente, otras empresas han creado otras máquinas virtuales Java, como Microsoft, pero siguiendo las especificaciones establecidas por Sun, para mantener la compatibilidad.

Utilidades del JDK

Utilidad	Descripción
javac	Compilador del lenguaje Java
java / javaw	Lanzador de aplicaciones (JVM)
javadoc	Generador de documentación
appletviewer	Ejecución y depuración de applets sin necesidad de navegador
jar	Gestión de archivos ".jar" (empaquetamiento de varios archivos)
jdb	Depurador de Java
javah	Generador de ficheros de cabecera
javap	Desensamblador de clases
extcheck	Para detectar conflictos en ficheros ".jar".

javadoc:

Utilizando los comentarios especiales /** */, nos generará los archivos "AllNames.html", "tree.html", "packages.html" y otro fichero "html" con el mismo nombre que nuestra aplicación.

javah:

Para que java se pueda aplicar con las condiciones propias de la plataforma en la que se ejecuta, necesitará ejecutar código nativo escrito en C o en otros lenguajes. Por ejemplo, para las aplicaciones incrustadas o embebidas en las que queramos por ejemplo encender LEDS, relés o sensores necesitaremos acceder al código nativo. Para esto, Java puede trabajar con las llamadas a rutinas nativas.

Para ayudarnos a la hora de escribir el código en C, JDK incluye javah, una utilidad que, dado un archivo class, genera los archivos de cabecera que necesitarán los programas escritos en C.

jdb:

Esta utilidad nos ayuda a depurar los programas, localizando los errores. Podemos detener la ejecución del programa en puntos concretos, para comprobar el valor de ciertas variables, etc.

javap:

Esta utilidad nos permite examinar el código de bytes de una clase compilada. Nos genera un informe con las funciones y variables disponibles. Esta utilidad se ejecuta especificando un fichero class como argumento. Tenemos la posibilidad de obtener la funcionalidad del código (aunque a un nivel muy bajo), especificando la opción –c.

Runtime

Lo que necesita un usuario para ejecutar un programa compilado es la máquina virtual. Si nosotros distribuimos un programa compilado y nos queremos asegurar de que el usuario dispone de todo lo necesario para ejecutarlo, podemos distribuirlo junto con el entorno de ejecución de Java (JRE) ó "runtime".

Este entorno ocupa mucho menos que el entorno JDK, puesto que no incluye las herramientas de desarrollo y otras utilidades.

En Windows, el instalador JRE instala automáticamente las utilidades java y javaw en la ruta de acceso del sistema operativo.

Elementos del lenguaje

Como en todos los lenguajes de programación, un programa es un fichero de texto que contiene una serie de instrucciones, con una sintáxis y una semántica. La primera define los elementos léxicos básicos, y la segunda, las relaciones entre ellos.

Identificadores

Un identificador es un nombre que se utiliza para identificar una única entidad en un programa: un tipo, una clase, un método, una constante, una variable, etc.

La sintáxis que debe cumplir es la siguiente (no debe empezar por dígito):

$$\{letra|\_|\$\}\ [\{letra|digito|\_|\$\}]^n$$

Los identificadores pueden tener cualquier número de caracteres. Como Java utiliza el juego de caracteres Unicode que incluye caracteres de muchos idiomas (incluyendo la "ñ" y la "ç") podemos utilizarlos todos para los identificadores. Java es sensible a las mayúsculas.

Palabras Clave

Son un conjunto de identificadores reservados por el lenguaje, es decir, no podremos utilizarlos como identificadores en los programas; solamente se podrán utilizar para el uso para el que fueron pensados. Por ejemplo, nombres de tipos (int, double), sentencias de control (if, while), etc.

Comentarios

En Java tenemos 3 posibilidades para insertar comentarios en el código fuente, con el objeto de aportar alguna aclaración de una parte del código que el programador considere oportuna para facilitar la comprensión por otras personas. Las dos primeras son similares a C++.

- /* Comentario tradicional al estilo de C en una o más líneas */
- // Comentario hasta el final de la línea.
- /** Comentario para Documentación con la utilidad **javadoc** */
  Se consideran como los comentarios /* .. */ por el compilador, y además, aparecerán en las páginas Web generadas por **javadoc**.

Tipos de datos

En Java, los tipos de datos de dividen en:

- **Tipos primitivos**: *byte, char, int, short, long, float, double* y *boolean*.
  Todos los tipos pueden tener valores negativos y positivos, excepto el tipo *boolean*, que sólo puede tomar los valores: *true* y *false*.
  Un *byte* es un valor 0 <= b <= 255; en cambio, un *char* es un carácter unicode, que consta de dos bytes.
- **Tipos referenciados**: arrays, clases e interfaces.

Hay dos tipos de **conversiones** entre tipos primitivos:

Automáticas: Se deben cumplir dos condiciones:

1. Que sean tipos compatibles.
   Compatibles: Enteros y reales.
   No compatibles: char y boolean.
2. Que el tipo destino tenga un mayor rango que el tipo origen (para no haya pérdida de información: truncar).

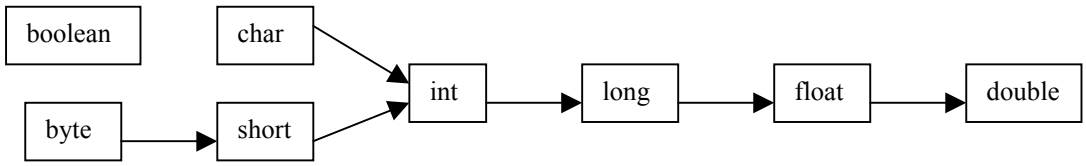
Explícitas (casting): <Variable> = (<tipo>) <Expresión>
Es un operador de "mutación", para forzar la conversión.

Para no tener errores de compilación, debido a conversiones de tipo no permitidas, debemos cumplir las "**reglas de promoción**" de Java (para las conversiones automáticas). Estas reglas especifican cómo los tipos pueden convertirse a otros tipos sin perder datos. Las reglas de promoción se aplican a expresiones que contienen valores de dos o más tipos de datos. Tales expresiones se conocen como expresiones de tipos mixtos. El tipo de cada valor en una expresión de tipo mixto se promueve al tipo "mas alto" de la expresión (en realidad, se crea una versión temporal de cada valor y se utiliza la expresión; los valores originales no se alteran).

Tabla de **promociones permitidas** de tipos primitivos (conversiones automáticas):

Tipos	Promociones permitidas
double	Ninguna (no hay tipos primitivos más grandes que double)
float	double
long	float o double
int	long, float o double
char	int, long, float o double
short	int, long, float o double
byte	short, int, long, float o double
boolean	Ninguna (en Java no se consideran números)

Gráficamente:



Constantes

Son identificadores tales que, una vez que se hayan declarado con un determinado valor, éste no se podrá cambiar durante la ejecución del programa.

La **sintáxis** para definir constantes es:

final [static] <Tipo> <Identificador> = <Valor>

Podemos definir constantes dentro de una clase o dentro de un método. Si lo hacemos dentro de la clase conviene especificar el calificador *static*, para evitar que cada uno de los objetos de esta clase guarden una copia de la constante. Dentro de un método no podremos especificar *static*, y el ámbito de la constante sólo será el método donde está definida.

La utilidad de las constantes es evitar el uso de "números mágicos" entre las líneas de nuestros programas.

final static double PI = 3.14159265358979323846264338327 95;

Variables

Antes de utilizar cualquier variable en un programa debemos declararla. Java, al igual que C/C++, es un lenguaje fuertemente tipado, es decir, debemos especificar siempre el tipo de una variable en su declaración.

Al igual que las constantes, se pueden declarar dentro de una clase o dentro de un método, pero además, dentro de cualquier bloque, es decir, dentro de cualquier sección de código comprendida entre llaves { .. }, aunque esto último no es aconsejable, desde el punto de vista de una buena programación.

De forma opcional, podremos asignarles un valor inicial, lo cual es siempre una buena costumbre de programación para evitar valores "basura".

Las variables miembro de una clase, son inicializadas automáticamente con unos valores por defecto:

Tipo	Valor de inicio por defecto
Caracteres	'\u0000'
Cadenas	"" (Cadena vacía)
Enteros	0
Reales	0.0
Booleanos	false
Referencias a objetos	null

Las variables locales (automáticas) de un método, no son inicializadas automáticamente por el compilador. Estas deben inicializarse antes de su uso, pues en otro caso, el compilador nos informará de un error de sintáxis. Estas variables ahorran memoria, pues tienen una duración automática, es decir, se crean cuando se ingresa en el bloque o método en el que se declaran y se destruyen cuando se sale de dicho bloque.

Una vez declaradas, se usan del mismo modo que las constantes, con la diferencia que las variables pueden cambiar de valor durante la ejecución.

La sintáxis para declarar una variable es la siguiente:

<Tipo> <Identificador> [= <Valor Inicial>];

El **alcance o visibilidad** de un identificador (variable u objeto) es la porción de programa en la que se puede hacer referencia a dicho identificador. Podría tener alcance de clase o alcance de bloque (incluyendo a un método).

Cualquier bloque puede tener variables locales definidas al principio. Su alcance será el bloque en el que están definidas. Cuando se anidan bloques, y un identificador de un bloque exterior tiene el mismo nombre de identificador que un identificador de un bloque interior, el compilador genera un error de compilación, para indicar que esa variable ya está definida. Si una variable local o un parámetro de un método tiene el mismo nombre que un atributo de la clase, el atributo queda "oculto" hasta que el bloque termine de ejecutarse. De todas formas, podremos acceder a los atributos, por medio de "this" (lo veremos cuando hablemos de clases).

Expresiones literales

Una **expresión** es una serie de variables, operadores y llamadas a métodos que se evalúa y devuelve un valor sencillo de un tipo concreto. Se utilizan para calcular y asignar valores a las variables y para controlar el flujo de un programa.

- El trabajo de una expresión se divide en dos partes: realizar los cálculos indicados por los elementos de la expresión y devolver algún valor.
- El tipo devuelto por una expresión, depende de los elementos utilizados en esta.
- En una llamada a un método, el tipo es el del valor de retorno del método.
- El orden de evaluación de una expresión puede variar el resultado.

Pueden formar parte de una expresión los siguientes literales:

- Valor **nulo**: null.

- Literal **entero** (int y long), en base decimal, octal y hexadecimal.
  Ejemplos: -25 (int), 37200L (long), 057 (octal),
  0x0A5F (hexadecimal corto), 0x0A5FL (hexadecimal largo)

- Literales **reales** (float y double)
  Ejemplos: 25.9F (float), 25.9[D] (double).

- Literales de tipo **lógico** (booleano).
  Solo son los dos valores: true y false.

- Literales de tipo **carácter** (char y cadena)
  Ejemplos: 'a', '\n' (Nueva línea), "Hola"
  Para las expresiones podemos utilizar otros **caracteres especiales**, como los de la tabla Unicode y otros caracteres no imprimibles. Del mismo modo debemos utilizar secuencias de escape para poder imprimir las comillas simples, las comillas dobles y la barra invertida. En todos los casos, van precedidos de la barra invertida "\".

Carácter a imprimir	Secuencia de escape
Retorno de carro (CR)	\r
Avance de línea (LF)	\n
Avance de página (FF)	\f
Tabulador (Tab)	\t
Retroceso (BS)	\b
Comillas simples '	\'
Comillas dobles "	\"
Barra invertida \	\\
Caracteres ASCII (Unicode, Octal)	\udddd, \ddd

Operadores

Los operadores son símbolos que nos permiten formar expresiones complejas, a partir de expresiones literales, variables y constantes. Algunos tienen más prioridad que otros a la hora de evaluar la expresión; esta prioridad podremos cambiarla con el uso de paréntesis.

De mayor a menor prioridad, los operadores son:

( ) [ ] .
++ -- - ~ !
New
* / %
+ -
<< >> >>>
< <= >= > instanceof
== !=
&
^
|
&&
||
? :
= *= /= %= += -= <<= >>= >>>= &= |= ^=

Consejo: Utilizar paréntesis para modificar la prioridad o cuando no estemos seguros del resultado de una expresión.

Sentencias de control de flujo

Las sentencias de control de flujo nos permiten controlar el orden de ejecución de las sentencias de los programas en función de ciertas condiciones y realizar bucles. Un bucle es un conjunto de sentencias que se repite un número determinado de veces o hasta se cumpla una determinada condición. En Java son similares a las de C/C++.

- *if* ... [*elseif* ...] [*else* ...]

- ((<Condición>) ? <Valor si true> : <Valor si False>)

- *switch* (<Expresion>) {
  case <Valor 1>: ... *break*;
  case <Valor 2>: ... *break*;
  default: ;
  }

- *while* (<Condición>) { ... }

- *do* { ... } *while* (<Condición>)

- *for* (<Exp.Inicio>; <Condición Fin>; <Exp. Actualización>) { ... }

- Sentencias *continue* y *break*:
  En el caso de bucles *while*, *do...while* y *for*, podemos forzar el fin de una iteración o salida del bucle con *continue* y con *break* respectivamente. Cuando existe un conjunto de bucles anidados, estas sentencias hacen referencia siempre al bucle más próximo.
  A diferencia de C/C++, podemos especificar de forma opcional una etiqueta a la que saltamos, pudiendo así realizar saltos multinivel.

```
Etiqueta1: do {
                ...
                for (i=1; ( i<=5) ;  i++) {
                        ...
                        break Etiqueta1
                }
        } while (<Condición>)
```

A la hora de escribir bloques y varias sentencias de control anidadas conviene poner en práctica el sangrado, para hacer los programas más comprensibles. Puede utilizarse un carácter tabulador o 3 espacios adicionales por cada nivel de profundidad.

Estructura de los programas

A la hora de escribir un programa en C/C++, debíamos tener una función llamada ***main***(), que era el punto de entrada al programa. En Java, también ocurre lo mismo, sólo que esta función ***main***() es ahora un método de una clase. Un programa en Java debe tener al menos un fichero con una clase pública, que será la clase de la aplicación. En esta clase es donde debemos definir el método *main*, del siguiente modo:

```
class Aplicacion {
   public static void main(String[] args)
   {
      // Punto de entrada a la aplicación
      // ...
   }
}
```

Cualquier programa Java contendrá otras clases, puesto que es un **lenguaje orientado a objetos**. Estas clases serán privadas, siendo la clase Aplicación la única clase pública. Esto es así porque en un único fichero java, sólo puede haber una clase pública.

Un **ejemplo** de aplicación sería el siguiente:

```
/* ================================================================
   Titulo :    Hola Mundo (Modo texto)
   Fichero:    HolaMundo.java
   ================================================================ */
class HolaMundo {
   public static void main(String[] args)
   {
      System.out.println("Hola mundo");
   }
}
/* ================================================================ */
```

Cuando ponemos "System.out.println" hacemos referencia al método "println" de la propiedad "out" de la clase "System". La propiedad "out" es el flujo de salida por defecto (estándar). Este flujo siempre está abierto y listo para aceptar datos de salida. Este flujo corresponde a la pantalla. Del mismo modo, tenemos la propiedad "in", que es el flujo de entrada por defecto (estándar). Este flujo corresponde al teclado.

Dentro de los métodos es donde especificamos las **instrucciones o sentencias** del programa. Estas sentencias pueden ser:

- **Sentencias simples** (terminadas en ';')
- **Sentencias compuestas o bloque**, que contienen una o más sentencias simples, encerradas todas ellas entre llaves { ... }.

La forma de gestionar los **argumentos pasados al programa desde la línea de órdenes** es distinta en C que en Java.

Todas las aplicaciones **Java** han de definir una función main así:

public static void main (String args[])

En este caso, "args" es una matriz de variables String. Cuando se ejecuta el programa desde la línea de órdenes (consola), esta matriz se rellena con los valores de los argumentos que especificamos a continuación del nombre del programa. Esta matriz se trata de una matriz cualquiera, por lo que podremos acceder a sus parámetros por medio de un índice, y saber el número de elementos mediante la propiedad "length". En este caso, el primer argumento es la cadena siguiente al nombre del programa.

En el **lenguaje "C"** tenemos algo parecido. También tenemos que definir una función main:

int main (int argc, char *argv[])

El primer parámetro "argc" es una variable entera que especifica el número de argumentos. En Java no es necesario, pues se trata de una matriz, y podemos obtener el número de argumentos con la propiedad "length", como ya hemos comentado. Otra diferencia es que en el lenguaje "C", el primero elemento del array (argv[0]) contiene el nombre del programa, por lo que "argc" siempre será ≥ 1, y contendrá un valor más que el número de parámetros real. En Java no, puesto que la clase ya contiene el nombre del ejecutable.

Programación Orientada a Objetos (POO)

Conceptos básicos de la POO

La POO comenzó en 1967 con Simula'67 y en los 70's con SmallTalk, pero no fue hasta los 80's cuando aumentó el interés por ellos, debido a la aparición de interfaces gráficos GUI (Graphical User Interface) de tipo WIMP (Windows, Icons, Mice, Pull-Down/Pointers).

Los primeros Lenguajes Orientados a Objetos (LOO) fueron modificaciones de lenguajes de 3ª generación existentes: C++, Objetive-C, Object-Pascal, etc. La mayoría de los actuales son así. La razón de esto es que los lenguajes diseñados exclusivamente para utilizar la POO eran menos eficientes que los lenguajes de programación estructurada.

Objetivos de la POO:

Simular modelos de la realidad, lo cual es bastante difícil con los lenguajes típicos de 3ª generación (programación estructurada). La POO trata de que la estructura de un programa sea lo más parecida al problema real que se está modelando.

Beneficios de la POO:

- **Reutilización de código**: incrementando la productividad.
- **Extensibilidad** (mantenimiento): podremos modificar el comportamiento de los programas sin tener que 'retocar' todos los módulos, es decir, hay una mayor independencia entre módulos, y cada uno de ellos tendrá un función clara.

Con la POO se facilitan las fases de Análisis y Diseño del problema en cuestión, previos a la fase de programación. Del mismo modo, se simplifica la fase de mantenimiento, con lo que se mejora la productividad.

La POO es un nuevo paradigma en la programación. Hasta ahora, los 3 paradigmas de la programación habían sido:

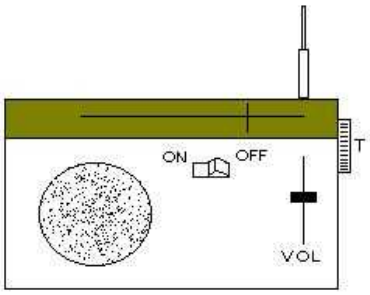
- Procedurales (Imperativos): C
- Funcionales: LISP
- Lógicos: PROLOG

Para el programador que conozca la programación estructurada y no la programación orientada a objetos, la siguiente tabla de equivalencia puede serle útil para familiarizarse con los conceptos básicos de la POO:

Programación Estructurada	Programación Orientada a Objetos
Programa = Datos + Algoritmos	Clase = Datos + Métodos Objeto = Estado + Comportamiento
Tipo definido por el usuario (estructura)	Clase
Variable (de un tipo concreto)	Objeto (de una clase concreta)
Campo de un struct	Propiedad
Subrutina, función o procedimiento	Método
Llamada a una subrutina	Paso de mensaje

Según Grady Booch (creador del UML junto con Jacobson y Rumbaugh):
LOO: Objetos, Clases y Herencia.
LBO: Objetos y Clases (Sin herencia) – Visual Basic 5.0 (o anteriores).

Ejemplo: Radio.



En una radio, tendremos los siguientes componentes:
- Botón de encendido/apagado (On/Off)
- Rueda de volumen
- Indicador de volumen (puede que en la misma rueda)
- Rueda de dial (sintonía)
- Display (para el dial)

Aparte, una radio siempre tendrá un "estado" (nivel de volumen y sintonía)

Mundo real	Clase
Radio	clsRadio
Acción: Encender (On)	Acción: Constructor
Acción: Apagar (Off)	Acción: Destructor
Acción: Subir volumen	Acción: VolumenSubir
Acción: Bajar volumen	Acción: VolumenBajar
Consultar Volumen	Acción: Volumen Propiedad: Volumen
Acción: Dial Izqda.	Acción: MoverIzquierda
Acción: Dial Dcha.	Acción: MoverDerecha
Consultar Dial	Acción: Dial Propiedad: Dial

Clase

Una clase es una **generalización** o plantilla en donde se define los datos (propiedades o atributos) y el comportamiento (métodos) que tendrán todos los objetos que se instancien (definan) a partir de ella. Una vez definida, podremos usarla exáctamente del mismo modo que el resto de clases que incorpora el lenguaje.

Los métodos y atributos que definamos tendrán un alcance global a la clase.

Objeto

Un objeto es una **instancia** (particularización) de una clase, es decir, tendrá todas las propiedades y métodos definidos en la clase, pero con unos valores concretos en la propiedades (estado).

Por tanto, antes de utilizar un objeto deberemos crearlo, al igual que deberemos eliminarlo al finalizar, aunque esto último no es necesario en Java, pues se hace de forma automática en el momento en que la ejecución del programa sale del ámbito de ejecución (visibilidad) del objeto.

Propiedad o atributo

Son los datos que contiene y encapsula una clase, es decir, deben estar ocultos al exterior, y tan sólo serán accesibles por medio de algún método. Cuando instanciemos objetos de esta clase, cada uno de ellos podrá tener valores distintos en las propiedades.

Método

Son las diferentes operaciones que se definen en la clase, que se podrán realizar sobre un objeto. Son las que definen su comportamiento. La forma de invocar un método de un objeto es mediante el envío o paso de mensajes. Así mismo, dentro de un método se podrán enviar nuevos mensajes a otros objetos para ejecutar alguna acción o solicitar cierta información.

A cada método debemos también **asignar un nombre** que exprese bien la tarea que realiza. Esto hace más comprensibles los programas sin la necesidad de escribir demasiados comentarios y fomenta la reutilización de código.

Conviene además que realice una sola tarea. Si un método realiza varias tareas, seguramente podrá dividirse en varios métodos más pequeños.

Debemos tener cuidado con el **tamaño de los métodos**, ya que un programa altamente modularizado puede hacer llamadas a muchos métodos, lo cual consume tiempo de ejecución y recursos; por otro lado, un programa monolítico (de una sola pieza) es difícil de programar, depurar y mantener. Además, un método que ocupe más de una o dos páginas será difícil de entender.

Mensaje

El mecanismo de comunicación entre objetos durante la ejecución del programa es el llamado *paso de mensajes*. Cuando un objeto quiere que otro objeto realice alguna acción o quiere solicitarle una información le envía un mensaje. Cuando el segundo objeto recibe el mensaje, éste ejecuta el método que tiene asociado dicho mensaje.

La POO tiene una serie de **propiedades**:

- Encapsulación.
- Polimorfismo.
- Herencia (Simple/Múltiple)
- Concurrencia
- Persistencia

Encapsulación

Es un mecanismo que nos permite ocultar información al exterior de una clase sobre las datos y detalles de implementación de la misma. Se basa en la idea de modularidad o uso de módulos. Básicamente, un módulo es un bloque de software independiente de los otros módulos pero que interacciona con ellos mediante algún mecanismo. La programación modular es anterior a la POO, pero ahora es mucho más clara pues cada módulo será una clase y tendrá una función concreta, pues reflejará el comportamiento de un objeto de la realidad.

Algunos autores se refieren a la encapsulación como un mecanismo de ocultación de información o de **abstracción**. Abstracción es la representación de las características esenciales de algo sin incluir detalles irrelevantes, con el objeto de tener una visión global.

Tendremos distintos niveles de abstracción según la visibilidad que tengamos de un objeto. A un nivel bajo de abstracción, podremos ver todos los detalles de la implementación; a un nivel más alto, tan sólo veremos el interface con el exterior, es decir, las propiedades y métodos públicos (caja negra).

Polimorfismo

Podemos definir **métodos** con el mismo nombre en distintas clases o en la misma, teniendo comportamiento diferente en cada una de ellas (código diferente). Un función sobrecargada es una función que tiene más de una definición.

La sobrecarga de funciones elimina la necesidad de tener que asignar nombres diferentes a las distintas funciones y se consiguen programas más fáciles de entender. Es el compilador el que se encarga de alterar el nombre para distinguirlos internamente.

Mediante el polimorfismo, es posible diseñar e implementar sistemas que son más fácilmente **extensibles**. Podemos tener una superclase genérica y varias subclases. Pues podremos añadir nuevas subclases sin necesidad de modificar ni recompilar la superclase y las otras subclases hermanas.

Por ejemplo, la clase "Impresora", con algún método "ImprimirArchivo". Podemos tener como subclases, las clases "Laser" y "Matricial". Si más adelante necesitamos añadir un nuevo tipo de impresora, p.e. "Tinta", no necesitaremos recompilar ninguna de las otras clases, tan sólo heredaríamos de "Impresora" y añadiríamos la nueva clase "Tinta" a la jerarquía. Al llamar al método "ImprimirArchivo", dependiendo del objeto que lo llame, se realizarán unas acciones u otras. Con las técnicas tradicionales (sin polimorfismo), deberíamos haber hecho unas sentencias de selección (if, switch, etc.) para detectar el tipo de impresora, y en función de ello, llamar a una función concreta para cada caso.

De forma análoga a la “sobrecarga de funciones”, la “sobrecarga de operadores”, permite al programador dar nuevos significados a los operadores que incorpora el lenguaje. Los operadores ya están sobrecargados en la mayoría de los lenguajes.

Por ejemplo, el operador suma + se utiliza en varias situaciones:

- Sumar enteros (1 o 2 instrucciones)
- Sumar reales (50 o 60 instrucciones)
- Concatenar cadenas (BASIC, Pascal, ...)

Prácticamente se pueden sobrecargar todos los operadores definidos por el lenguaje, pero no otros (no podemos crear nuevos operadores).

La sobrecarga de operadores nos permite utilizar objetos como operandos, y nosotros definiremos cómo se realiza la operación, codificándolo en un método.

Por ejemplo, imaginemos que tenemos la clase Matriz. Internamente, esta clase tendrá un array de dos dimensiones (n x m), con los distintos valores en filas y columnas. Para evitar el tener que llamar a un método que sume matrices, podríamos sobrecargar el operador suma (+).

Suma de enteros	Suma de reales (float)	Suma de matrices
int a=2, b=2, c; c = a + b;	float a=2.5, b=1.5, c; c = a + b;	Matriz a, b, c; c = a + b;

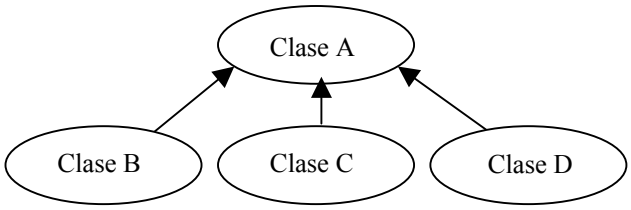
Herencia

Podemos definir una clase a partir de una clase ya existente, heredando sus propiedades y métodos, pudiendo añadir otras nuevas o redefiniendo otras. La clase existente recibe el nombre de **clase base o superclase**, y la nueva, **clase derivada o subclase**. La herencia reduce el número de cosas que tenemos que definir en la nueva clase.

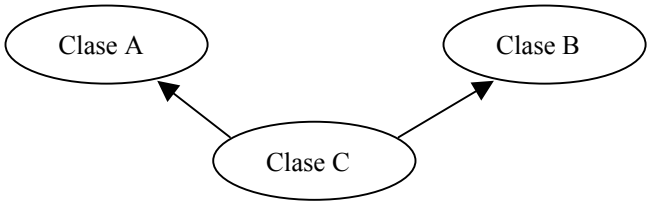
La herencia en programación es similar a la genética. Los hijos reciben atributos de los padres (ADN), y éstos a su vez de los abuelos, etc., pero cada uno de ellos se particulariza con características propias. Las características son comunes tanto al padre como a la madre, son las características de un ser humano: Cabeza, dos ojos, dos brazos, etc., pero el hijo finalmente tendrá un color de ojos o pelo concreto, teniendo en cuenta las características particulares del padre y de la madre, genes dominantes, azar, etc.

Existen dos tipos de herencia:

- **Herencia simple**: es aquella en la que cada clase derivada hereda de una única clase, es decir, cada clase tiene un solo ascendiente. Cada clase base puede sin embargo tener muchos descendientes.



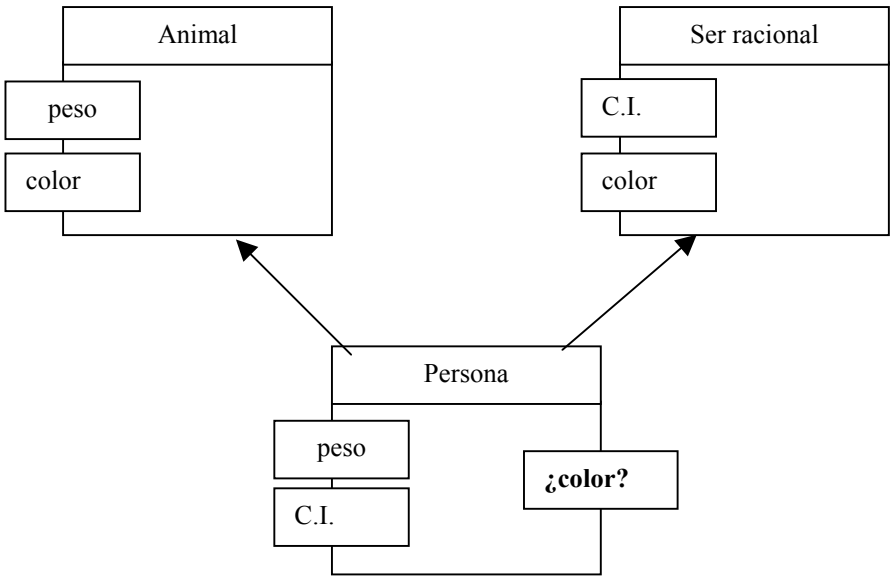
- **Herencia múltiple**: es aquella en la cual una clase derivada tiene más de una clase base.



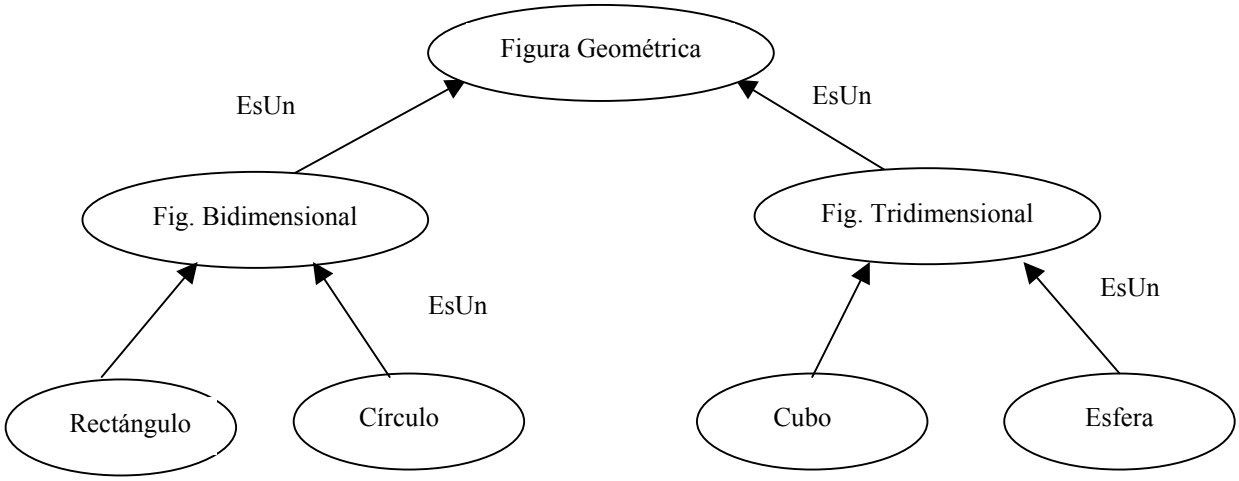
Nos todos los LOO la soportan. Además, no es aconsejable pues podemos tener problemas: ambigüedad y herencias repetidas.

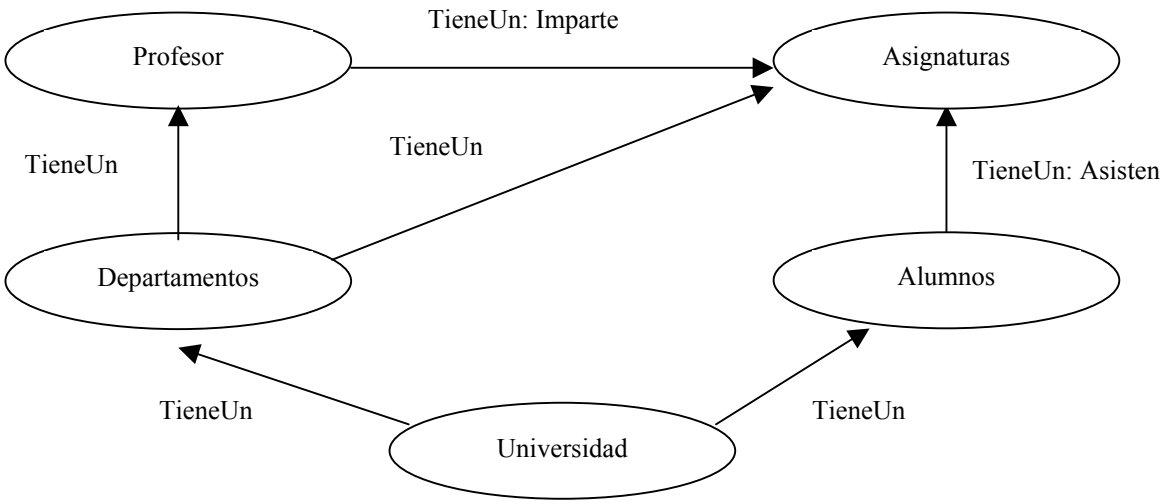
En Java, a diferencia de C++, no existe la herencia múltiple, esto es, una subclase tan sólo tiene una superclase. Esto facilita el aprendizaje pues la estructura que forman es de árbol.

- Ambigüedad, la cual se produce cuando se hereda un miembro de datos o función con el mismo nombre de más de una clase base, y pueden ser contradictorias.



Las relaciones de herencia se deben utilizar para reflejar relaciones entre objetos del mundo real de la mejor forma posible. Los dos tipos de relaciones más comunmente simulados con relaciones de herencia son: "EsUn" (semántica de clasificación) y "TieneUn" (semántica de composición).





El uso de la herencia (tanto simple como múltiple) adolece una serie de problemas:

- Velocidad de ejecución baja
- Incremento del tamaño del programa
- Programas más complejos

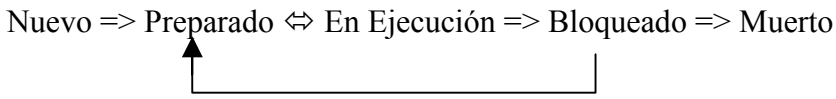
Algunos autores consideran la herencia como: "Un paracaidas, no lo siempre lo necesitas, pero cuando hace falta, te alegras de que esté ahí".

Persistencia

Propiedad de un objeto que le permite "sobrevivir" (perdurar) en la ejecución de un programa, es decir, los datos permanecen durante la ejecución del programa y, al terminar la ejecución y volver a ejecutarlo, continúan exactamente con el mismo valor. La idea es equivalente al uso ficheros o bases de datos, pero de forma transparente para el programador. Normalmente se consigue con alguna clase que implementa la persistencia, o con el uso de las denominadas Bases de Datos Orientadas a Objetos (BDOO).

Concurrencia

Un programa (proceso) concurrente se caracteriza por tener varios hilos de ejecución (thread). Todos los hilos están dentro del espacio de trabajo de un proceso, compartiendo variables globales, ficheros abiertos y otros recursos. El concepto es parecido al de multitarea (varios procesos), salvo que en este caso no lo controla el sistema operativo si el propio programador dentro del programa:



Normalmente se consigue haciendo llamadas al sistema operativo, por lo que no tenemos asegurada la transportabilidad.

Garbage Collection

Se trata de un mecanismo de "limpieza" de la memoria dinámica, memoria utilizada por los objetos. Con este proceso se eliminan de la memoria dinámica los objetos que ya no se utilizan más a lo largo de la ejecución del programa. Existen varias estategias para implementar este proceso:

- Estar constantemente en ejecución por debajo de nuestro programa (hilo).
- Ejecutarse sólo en determinadas circustancias.
- Llamada explícita del recolector de basura.

Los LOO han seguido evolucionando, pero no terminan de cuajar debido a ciertos problemas a la hora de simular un problema del mundo real: persistencia y concurrencia. Con Java, se han solucionado estos dos problemas.

Implementación de la POO con Java

Clases

Hemos dicho que las clases contienen atributos y métodos. La forma de definir una clase es la siguiente:

```
[<Modificadores de clase>] class <NombreClase> [extends <SuperClase>]
[implements <Lista Interfaces>]
{
        // Atributos
        [<Modificador de atributo>]  <Tipo> <NombreAtributo>;

        // Métodos
        // ...
}
```

Los **modificadores de clase** que podemos utilizar son:

Modificador	Significado
public	Clase pública: se puede usar desde fuera del fichero donde está implementada.
private	Clase privada: sólo se podrá utilizar dentro del fichero donde está implementada. Es el valor por defecto si no se especifica nada.
abstract	Clase abstracta: contiene métodos virtuales que tendrán que ser definidos por alguna subclase. No pueden crearse instancias de estas clases, tan sólo pueden ser ampliadas por otras subclases, las cuales deben definir todos los métodos virtuales.
final	Clase final: ninguna clase puede heredar de ella. Además, de forma automática, todos sus métodos se convierten también en **final.**

Con la cláusula **extends** indicamos la clase de la que heredamos (única).
Con la cláusula **implements** indicamos una lista de interfaces que implementaremos. Esto lo veremos más adelante, cuando hablemos de interfaces.

En todas las clases existen siempre dos propiedades: **this** y **super**:

- **this**: Es una referencia al objeto (instancia de la clase) que está ejecutando en ese momento un método, pues todos los objetos comparten la definición de la clase. Utilizar explícitamente la referencia this en un método estático (static) es un error de sintáxis. Dentro de un constructor, podemos llamar al contructor sin parámetros con this(), que realizaría las labores de inicialización.
- **super**: Es una referencia a la superclase desde la que una clase hereda, pudiendo así, ejecutar métodos de la superclase (incluyendo contructores y miembros privados). Si en un constructor no llamamos al contructor de la superclase con super(), el constructor de subclase llamará implícitamente al constructor por omisión de la superclase (o al constructor sin argumentos). Si ésta no tiene ningún constructor ocurrirá un error de sintáxis. No puede utilizarse llamadas super() en cascada, es decir: super.super.x. Daría error de sintáxis. Deberíamos utilizar una técnica de conversión explícita u obligada: ((ClaseAbuelo) this).Miembro.

Java permite crear clases dentro de clases. Las clases que están dentro de otras se denominan **clases internas**. También permite la definición de **clases anónimas** o sin nombre

Atributos

La sintáxis para definir atributos es:

[<Modificador de atributo>] <Tipo> <NombreAtributo>;

Los **modificadores de atributo** son:

Modificador	Significado
public	Atributo público: se puede usar desde fuera de la clase directamente, esto es, sin utilizar ningún método (no se recomienda).
private	Atributo privado: sólo se podrá utilizar dentro de la clase. Para acceder a ellos desde fuera de la clase se tienen que utilizar métodos públicos.
static	Atributo estático: Cada objeto de una clase tiene su propia copia de todos los atributos. En ciertos casos, puede interesar que todos los objetos de la clase compartan una sola copia de una variable. Un atributo "static" representa información que abarca toda la clase, su alcance es toda la clase. Se puede utilizar junto con "public" o "private".

Si un atributo es público, cualquier método del programa podrá leerla o establecerla a voluntad.

El acceso a datos privados (private) a través de métodos set y get protege a las variables para que no reciban valores no válidos, y además, aísla a los clientes de la clase respecto a la representación de las variables de ejemplar. Así, si la representación de los datos cambia (p.e. un float lo convertimos en un int para reducir consumo de memoria), sólo tendremos que cambiar la implementación del método; los clientes no necesitan cambiar nada, puesto que el interfaz provisto para estos métodos sigue siendo el mismo.

No es necesario que estos métodos se llamen set y get (fijar y obtener) pero es recomendable.

Cuando no se especifica ningún modificador de acceso de un método o variable (atributo) se considera que el método tiene **acceso "amigable"** (también denominado acceso de paquete). Si el programa utiliza varias clases dentro de un mismo paquete (grupo de clases relacionadas entre sí), cada una de estas clases puede acceder a los métodos y atributos de las demás directamente a través de una referencia a un objeto. El acceso amigable permite la interacción de objetos de diferentes clases sin necesidad de métodos get y set para acceder a los datos, eliminándose así algo del gasto extra por llamadas a métodos.

Sin embargo, el acceso amigable desvirtúa el ocultamiento de información y reduce el valor de la POO.

Métodos

```
[<Modificadores de método>] <TipoRetorno> <NombreMetodo> ( [<ListaParámetros>] )
[throws <Lista TiposExcepcion>]
{
        // Variables locales
        ...
        // Instrucciones o sentencias
        ...
        return (<Expresión>);
}
```

Los **modificadores de método** que podemos utilizar son:

Modificador	Significado
public	Método público: es visible desde otras clases, subclases, incluso de otros paquetes.
private	Método privado: sólo es visible desde dentro de la misma clase.
protected	Método protegido: es visible desde dentro de la misma clase, y además, desde todas las subclases que hereden de ella, que estén en el mismo paquete.
static	Método estático: es compartido por todos los objetos de la clase, por lo que en vez de llamarlo con <NombreObjeto>.<Método>, se debe llamar con <NombreClase>.<Método>. Sólo se usa para acceder a los atributos de tipo **static**, no pudiendo acceder a los demás.
abstract	Método abstracto (virtual): deberá ser redefinido por una subclase.
final	Método final: no se podrá redefinir por ninguna subclase. Si tenemos la seguridad de que esto es así, conviene poner el modificador **final**, pues el código se ejecutará más rápido (eliminamos un gasto extra al invocar al método, pues no se hace ninguna llamada, sino que el compilador la sustituye por el código completo de la función: técnica de inclusión de código en línea. En C++, se usaban los "inlines"). Todos los métodos de una clase "final" son implícitamente "final". Estos métodos utilizan, por tanto, ligado estático, no dinámico, que siempre es más rápido.
synchronized	Método sincronizado: se accede a él en exclusión mutua, es decir, si hay un objeto ejecutando este método, y un segundo objeto intenta ejecutarlo también (desde otro thread), tendrá que esperar a que el primer objeto termine la ejecución.
native	Método nativo: permite la inclusión de código en otros lenguajes, como por ejemplo C.

La claúsula **throws** se utiliza para la gestión de errores o manejador de excepciones que veremos más adelante.

En una clase existen **dos métodos especiales**:

- **Constructor**: cuya misión es inicializar un objeto en el momento en que se declara (instancia a partir de su clase). Aunque su uso no es obligatorio, es muy aconsejable, por ejemplo, para inicializar los atributos de la clase. El constructor es un método como los demás, pero que debe de tener el mismo nombre que la clase, y no hay que especificar un tipo de retorno. Lo que si podemos hacer de forma opcional es especificar parámetros; si no lo hacemos, el constructor recibe el nombre de constructor por defecto. Es posible definir más de un constructor, siempre y cuando los parámetros de inicialización sean distintos (número o tipo), y el constructor que se ejecutará dependerá de los parámetros pasados en el momento de la creación del objeto con el operador **new**. De todas formas, es posible llamar explícitamente a un constructor desde cualquier método de su misma clase, con la propiedad **this**, especificando diferentes atributos para seleccionar el constructor adecuado.
- **Destructor**: Es el método que se ejecutará justo antes de que el objeto se destruya. En este caso, el método destructor debe ser único y tener la siguiente estructura:

  protected void **finalize**() { ... }

  Una referencia a un objeto puede ser eliminada porque el flujo de ejecución salga fuera del ámbito donde ella está declarada, o porque explícitamente se le asigne el valor **null**. Cuando no quedan referencias a un objeto, el recolector de basura de Java se invoca automáticamente. Podemos ejecutar explícitamente un destructor, pero esto no hace que sea eliminado por el recolector de basura. Sólo cuando se eliminan todas las referencias que apuntan al mismo, éste se marca como destruible.

En cuanto a la **lista de parámetros**, podemos pasarlos por valor y por referencia:

- **Paso por valor**: el método puede utilizar dicho parámetro, incluso modificar su valor, pero en el método desde el que se llamó al método actual, la variable no cambiará de valor.
- **Paso por referencia**: cualquier modificación que este método haga sobre el valor del parámetro afectará a la variable u objeto fuera de este método.

Lenguaje	Paso de parámetros	Mecanismo
C	Por valor	Por defecto
	Por referencia	'*' delante del parámetro en la función y & en la llamada
Pascal	Por valor	'var' delante del parámetro en la definición de función
	Por referencia	Por defecto
Visual Basic	Por valor	'ByVal' delante del parámetro en la definición de la función
	Por referencia	'ByRef' delante del parámetro en la definición de la función
Java	Por valor	Todos los tipos primitivos
	Por referencia	Objetos y Arrays

Es decir, en Java, los objetos y los arrays siempre se pasan por referencia, no es posible pasarlos por valor. En cuanto a los tipos primitivos, si que es posible pasarlos por referencia, pero no de forma directa. Se trata de realizar un "truco", convertirlo en un array, que como hemos visto si se pueden pasar por referencia. Aunque los objetos también se pasan por referencia, se podría pensar que se podría utilizar las clases "envoltura", pero no funciona. Vamos a ver primero este caso, cuando lo convertimos a objeto.

Conversión en un objeto: el mecanismo se llama "encapsulación" o "envoltura". Se trata de encapsular un tipo primitivo en una clase equivalente, para poder gestionarlos como si fueran objetos. Las clases de envoltura de tipo que disponemos son:

Tipo primitivo	Clase que lo encapsula
byte	Byte
short / int / long	Short / Integer / Long
float / double	Float / Double
char	Character
boolean	Boolean

Estas clases heredan de la clase Object (clase raíz), pudiendo por tanto utilizar todos sus métodos.

```
public static void Incrementar (Integer objInt)
{
        int nInt = objInt.intValue();
        nInt++;
        objInt = new Integer(nInt);
}

public void main (String[] args)
{
        int nInt = 20;
        Integer objInt = new Integer(nInt);
        Incrementar (objInt);
        nInt = objInt.intValue();
}
```

¡¡ No funciona !!

Este método no funciona, pues no tenemos un método para asignarle un nuevo valor, al igual que con "intValue" obtenemos su valor.

Conversión en un array:

```
public static void Incrementar (int[] aInt)
{
        aInt [0]++;
}

public void main (String[] args)
{
        int nInt = 20;
        int[] aInt = { 20 };
        Incrementar (aInt);
        nInt = aInt[0];
}
```

¡¡ Si funciona !!

Mensajes

El envío o paso de mensajes es simplemente la llamada a uno de los métodos de una clase, o la consulta de un atributo público (no aconsejable).

La forma de realizar esto depende del tipo de método al que vayamos a enviar el mensaje.

Métodos de tipo "static"	<NombreClase>.<Método>
Todos los demás tipos de método	<NombreObjeto>.<Método>

Por último, decir que se pueden concatenar varias llamadas en una sola línea, sin necesidad de tener que especificar el nombre del objeto.

Polimorfismo

La librería de clases de Java incorpora muchos métodos sobrecargados.

Por ejemplo, la clase **InputStream** (flujos de entrada) implementa 3 formas del método **read**:

- public int read ()
- public int read (byte[] b)
- public int read (byte[] b, int off, int len)

La clase **PrintStream** implementa múltiples formas de los métodos **print**:

- public void print (int i)
- public void print (double d)
- public void print (char[] s)

Nosotros podemos definir también varios métodos con el mismo nombre. El compilador decide cual de ellos utilizar en función de:

- Número de parámetros
- El tipo de los mismos
- El orden

No se admite que dos métodos sólo difieran en el tipo retornado. Si creamos métodos sobrecargados con listas de parámetros idénticas y diferentes tipos devueltos obtendremos un error de sintáxis durante la compilación.

La sobrecarga de funciones **elimina la necesidad de tener que asignar nombres diferentes** a las distintas funciones. En el caso anterior del método **print** podríamos haber dado los siguientes nombres:

- public void printInt (int i)
- public void printDouble (double d)
- public void printStr (char[] s)

Con el polimorfismo (sobrecarga de funciones) **conseguimos**:

- **Programas más fáciles de entender**: Es el compilador el que se encarga de alterar el nombre para distinguirlos internamente.
- **Programas fácilmente extensibles**: Podemos ejecutar sobre un conjunto de objetos distintos, unos métodos, que dependiendo de la clase de esos objetos, se comportará de una manera determinada. Esto hace posible agregar a un sistema nuevos clases, cuyos objetos serán capaces de responder a mensajes

ya existentes sin tener que modificar el sistema base. Esto es posible gracias al ligado dinámico (en tiempo de ejecución) que normalmente es más lento que el ligado estático (en tiempo de compilación). Pero en este caso, el polimorfismo implementado con ligado dinámico de métodos es tan eficiente como el estático, es decir, tan eficiente como si lo hubiésemos implementado con lógica de conmutación (switch, if, etc.)

C++ tiene la palabra reservada "operator", para realizar la sobrecarga de operadores. Java también tiene esta palabra como reservada pero no la usa, es decir, Java no permite (todavía) la **sobrecarga de operadores**.

Persistencia

Con el uso de la técnica de "Seriación de Objetos", que consiste en escribir en un fichero un objeto, para posteriormente realizar el proceso inverso de "Deseriación de Objetos" para leerlos de nuevo. Esto se hace con las clases "ObjectOutputStream" y "ObjectInputStream" del paquete "java.io". Se convierte el estado de un objeto (valor de los atributos), su clase y su prototipo en una secuencia de bytes y viceversa. Otra posibilidad es el uso de Bases de Datos Orientadas a Objetos, frente a las tradicionales Bases de Datos Relacionales (SQL).

Concurrencia

Se consigue con el uso de hilos (threads), que se ejecutan de forma concurrente. Si no creamos nuevos hilos, tan sólo tendremos un hilo principal o primario, creado por el sistema al crear el proceso para un programa. Para crear hilos utilizamos la clase "Thread" del paquete "java.lang", y el modificador de método "synchronized".

Garbage Collection

Java tiene un algoritmo de "Recolección de basura" que se ejecuta en un subproceso (hilo) de forma paralela a nuestra aplicación, limpiando los objetos desreferenciados en segundo plano, continuamente. No nos preocupamos de la reserva de memoria para la creación de objetos, ni de su liberación. Java lo gestiona automáticamente.

El proceso de recolección de basura se activa cuando la JVM detecta que hay objetos que no están siendo referenciados por ninguna variable, para liberar el espacio que éste ocupaba.

Sin embargo, podemos forzar una recolección de basura completa, llamando al método **gc** (garbage collector) de la clase System y Runtime. No se ejecutará de manera inmediata, sino cuando la JVM pueda.

Creación y referencia de objetos

Una vez definida una clase, para **crear un objeto** a partir de la clase (instanciar) utilizaremos el operador new del siguiente modo:

<Objeto> = **new** <NombreClase>([<Lista de parámetros>])

La lista opcional de parámetros será la que determinará el **constructor** que se utilizará.

Cuando creamos un objeto, también creamos una **referencia** a dicho objeto. Si tenemos creado un objeto de una determinada clase, podemos declarar otro objeto de las misma clase, y asignarle la referencia que ya teníamos, sin utilizar el operador **new**. En este caso, tendremos dos referencias al mismo objeto.

Un objeto no se eliminará de la memoria mientras exista alguna referencia a dicho objeto, además de, como ya dijimos, salgamos de su ámbito (visibilidad), por ejemplo, un bloque, un método o una clase.

Comparación de objetos

La forma de comparar si dos objetos es definiendo un método **equals** en la clase, en el que especificaremos cuándo el contenido de dos objetos son iguales.

El operador "==" nos dice cuándo dos referencias son iguales (en el caso de trabajar con objetos, ya que en el caso de trabajar con tipos primitivos si nos compara el contenido o valor de dos variables), es decir, si ambas referencias se refieren al mismo objeto en memoria.

Matrices o Arrays

Una matriz es una estructura homogénea, compuesta por varios elementos, todos del mismo tipo y almacenados consecutivamente en memoria. Podremos acceder a cada elemento directamente por el nombre de la variable matriz, seguida de uno o más subíndices encerrados entre corchetes, dependiendo del número de dimensiones de la matriz.

Matriz m:

m[0]	m[1]	m[2]	m[3]	m[4]	m[5]	m[6]	m[7]	m[8]	m[9]	...

En Java, las matrices son objetos. Para utilizar una matriz debemos por tanto declararla y después crearla: Más tarde, tendremos que asignarle valores a los elementos.

1. Declarar una matriz.

Hay dos posibilidades:

<Tipo>[] nombre;
<Tipo> nombre[];

El tipo puede ser cualquier tipo primitivo o referenciado (objetos). En ningún caso especificamos el tamaño de la matriz.

2. Crear una matriz.

Reservamos la memoria necesaria para almacenar todos los elementos especificados. Este número puede ser una variable, con lo que podremos decidir el tamaño del array en tiempo de ejecución.

Nombre = new <Tipo>[<NumeroElementos>];

También es posible declarar y crear una matriz en la misma línea:

<Tipo>[] nombre = new <Tipo>[<NumeroElementos>];
<Tipo> nombre[] = new <Tipo>[<NumeroElementos>];

3. Inicializar valores a los elementos.

Puesto que una matriz es un objeto, el compilador la inicializa automáticamente; en este caso, lo que se inicializa es el valor de todos y cada uno de los elementos.

Una forma de asignar valores a los elementos es accediendo a los elementos de una matriz, mediante el subíndice:

m[5] = 10;

Tenemos también la posibilidad de inicializar todos los valores del array de un golpe. En el siguiente caso, estamos declarando, creando y asignando todos los valores:

int m[] = { 2, 4, 6, 8 }; // Array de 4 elementos

Para conocer el **número de elementos** de una matriz tenemos la propiedad: **length**.

Los **límites de los índices** válidos de una matriz 'm' son 0 .. (m.length-1). Si intentamos acceder a un elemento fuera de estos límites, se producirá un error en tiempo de ejecución, de tipo ArrayIndexOutOfBoundsException.

4. Consultar valores de los elementos.

Se hace del mismo modo que en la asignación.

En el caso de los **arrays multidimensionales**, prácticamente es lo mismo. Por ejemplo, para un array de 2 dimensiones, tendremos un array de arrays, los cuales no tienen porqué tener la misma longitud.

1. Declarar:

```
<Tipo>[][] nombre;
<Tipo> nombre[][];
```

2. Crear:

```
Nombre = new <Tipo>[<NumeroElementos1>][<NumeroElementos2>];
```

3. Inicializar los elementos:

```
int m[][] = { { 2, 4, 6 }, {1, 3} }       // Array de 3 elementos y otro de 2
```

Para **pasar un array a un método**, el método debe declararse del siguiente modo:

```
OrdenarArray ( int m[] );
```

Y debemos llamarlo: OrdenarArray (m);

Los arrays siempre **se pasan por referencia**, por razones de rendimiento. Si se pasaran por valor, se pasaría una copia de cada elemento. Si el array fuese grande y se llamase continuamente a un método se desperdiciaría mucho tiempo.

Podemos **copiar todo un array** o solo una parte de golpe con el método System.arraycopy, del paquete java.lang.

Java dispone también de la **clase Arrays**, específica para gestionar matrices, dentro del paquete java.util. Por ejemplo, tenemos los métodos:

equals: comparar dos arrays
fill: rellenar una matriz con un valor concreto

Cadenas de caracteres

Son arrays unidimensionales cuyos elementos son de tipo 'char'. Sería la forma de gestionar las cadenas que tiene el lenguaje C.

Podríamos asignar una cadena de la forma que hemos visto de inicialización de elementos de una matriz:

```
char[] cadena = new char[10];   cadena =  "abc";
char[] cadena = { 'a', 'b', 'c' };
char[] cadena = { 97, 98, 99 };
```

En este último caso, estamos asignando el código Unicode de cada uno de los caracteres. Los 128 primeros códigos Unicode coinciden con los 128 primeros códigos ASCII.

En las matrices de caracteres, el **carácter de inicialización** de los elementos es el carácter '\0'. Si hemos asignado menos caracteres que el número máximo de

elementos, el resto tendrá dicho valor. Si utilizamos el método "println" visualizaremos la cadena y todos estos caracteres '\0'.

ATENCION: al utilizar el método "length", nos dice el número de elementos máximo de la cadena de caracteres, no su longitud.

Clase String

Nos facilita el trabajo con cadenas de caracteres, pues ya no tenemos que trabajar con índices de un array, y tiene multitud de métodos. Encapsula una cadena de caracteres.

Métodos importantes:

- concat: Devuelve un nuevo objeto String resultado de concatenar otros dos. También podemos concatenar dos String con los operadores + y +=.
- compareTo: Compara lexicográficamente dos Strings (menor, igual, mayor).
- length: Devuelve el número de caracteres Unicode.
- toLowerCase: Transforma a minúsculas.
- toUpperCase: Transforma a mayúsculas.
- trim: Elimina los espacios en blanco finales.
- indexOf: Obtiene el índice donde empieza una cadena o un carácter.
- substring: Obtiene una subcadena.
- replace: Sustituye todas las ocurrencias de un carácter por otro.
- valueOf: Transforma a un String una variable.

Hay varias formas de crear un objeto String:

- String str = "abc";
- String str = new String("abc");
- String str = new String("abc", offset, length);
- String str = new String("abc", LowHight, offset, length);
- String str = new String("abc", LowHight);

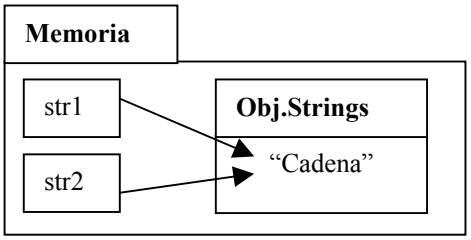
En los parámetros **offset** y **length** especificamos el desplazamiento y el número de caracteres de la cadena del primer parámetro que queremos copiar.

En el parámetro **LowHight** indicamos si queremos utilizar el byte de orden bajo o el byte de orden alto. Como ya hemos comentado, los caracteres en Java son caracteres Unicode de dos bytes. Normalmente especificamos cero como valor del byte alto cuando queremos usar los caracteres del conjunto de caracteres ASCII.

Java transforma los **literales de tipo cadena** (“abc”) en objetos. Estos objetos los almacena en una zona específica de la memoria llamada “Area de memoria de objetos String”. Se trata de un diccionario de palabras, donde no pueden haber dos palabras iguales. Este tipo de objetos no son modificables. Si posteriormente aparece el mismo literal en cualquier otra parte del código, el compilador no añade un nuevo objeto, sino que utiliza el que ya teníamos, es decir, lo referencia. En este caso, el objeto string “abc” tendría dos referencias. Esto se hace así para ahorrar memoria.

String str1 = “Cadena”;
String str2 = “Cadena”;

str1.equals (str2) ⇒ true
str1 == str2 ⇒ true

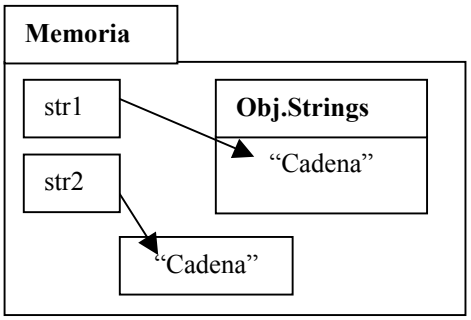


El método “equals” compara el contenido de dos objetos.
El operador “==” compara la referencia.

Si hubiéramos utilizado el operador new, lo que hubiera hecho el compilador es reservar espacio para un nuevo objeto en la memoria.

String str1 = “Cadena”;
String str2 = new String(“Cadena”);

str1.equals (str2) ⇒ true
str1 == str2 ⇒ false



En este caso tendríamos dos veces el literal “Cadena” en la memoria. Sin embargo, podemos forzar a que el objeto creado dinámicamente de tipo String se coloque en la zona de memoria reservada para los objetos String (literales):

```
str2 = str2.intern();
```

Como el literal “Cadena” ya existía en dicha zona, de nuevo los dos objetos apuntarán (referenciarán) al mismo literal.

Para comparar dos objetos String hemos visto que podemos utilizar el método equals y el operador "==", como para cualquier tipo de objeto. Hay además otros métodos específicos para comparar objetos de tipo String:

- **equalsIgnoreCase**: no tiene en cuenta las mayúsculas/minúsculas.
- **compareTo**: Retorna 0 si son iguales, <0 si el String que invoca al método es menor que el String que se pasa como argumento, y >0 si es mayor, según el orden lexicográfico.
- **regionMatches**: Para determinar si subcadenas de dos objetos String son o no iguales.
- **startsWith** y **endsWith**: si una cadena empieza o termina con otra.

La **comparación de objetos String grandes** es una operación relativamente lenta. Con el uso del método "intern()" de la clase String podemos mejorar el rendimiento de las comparaciones, pues las podemos comparar con el operador "==" que es más rápido que el método "equals" ó "equalsIgnoreCase", ya que compara dos referencias, no su contenido, lo cual requiere una comparación de cada uno de los pares de caracteres correspondientes de cada String.

IMPORTANTE: Los objetos String **no son modificables**. ¿Qué es lo que ocurre cuando ejecutamos algún método que modifica su valor?. Se crea un nuevo objeto, en la memoria dinámica. Posteriormente, mediante el uso del método “intern()”, podremos volver a moverlo a la zona de memoria de objetos Strings.

```
str1 = str1.replace (‘c’,’x’);
```

Clase StringBuffer

La clase String nos facilita mucho el trabajo para manejar cadenas de caracteres. Sin embargo, como hemos visto, una vez que se crea un String, su contenido nunca cambia (son constantes). Java dispone de la clase StringBuffer, para manipular cadenas de caracteres de forma dinámica, es decir, modificables.

Java distingue entre las cadenas constantes y las modificables por razones de optimización; en particular, Java puede realizar ciertas optimizaciones al manejar objetos String (como compartir un objeto String entre varias referencias) porque sabe que estos objetos no van a cambiar. Por eso, si sabemos que el objeto no va ha cambiar, conviene elegir la clase String, para obtener un mejor rendimiento.

Constructores:

- StringBuffer str = new StringBuffer(); // Longitud por defecto (16)
- StringBuffer str = new StringBuffer(50); // Longitud 50
- StringBuffer str = new StringBuffer("abc"); // Longitud 3 + 16

Los objetos StringBuffer ajustan su tamaño automáticamente a la cantidad de caracteres que lo componen.

Algunos de los métodos más importantes son:

- **toString**: podemos convertir en cualquier momento un objeto StringBuffer a un objeto String, para poder imprimirlos.
- **length**: devuelve el número de caracteres actualmente (longitud).
- **capacity**: devuelve el número de caracteres que se pueden almacenar sin asignar más memoria.
- **ensureCapacity**: para asegurarnos de que hemos reservado una cantidad suficiente para almacenar un número de caracteres, sin esperar a que se ajuste automáticamente (capacidad mínima).
- **setLength**: para modificar (aumentar o reducir) la longitud. Si la longitud especificada es menor que el número de caracteres que actualmente están en el StringBuffer, los caracteres se truncarán a la longitud especificada. Si, por el contrario, la longitud especificada es mayor, se anexarán caracteres nulos (\0).

- **charAt**: obtener el caracter que está en el índice especificado.
- **setCharAt**: establece un carácter en la posición especificada.
- **getChars**: devuelve un array de caracteres, subcadena del StringBuffer.

- **append**: anexa un String al final del StringBuffer. Si el argumento no es un String, lo convierte previamente. Este método es el que utiliza el compilador para implementar los operadores + y += que concatenan objetos String.
- **insert**: nos permite insertar valores de distintos tipos de datos en cualquier posición de un StringBuffer: tipos de datos primitivos, arrays de caracteres y objetos String. Previamente, el parámetro se convierte a un String, y después se inserta en la posición indicada.

Clase StringTokenizer

Con esta clase podemos extraer unidades lexicográficas (tokens) desde un objeto String. Los tokens se dividen unos de otros mediante delimitadores (espacios en blanco, tabuladores, saltos de línea, retornos de carro, comas, etc.). Dicho de otro modo, muchas veces nos interesa dividir una cadena de caracteres en subcadenas, indicando un caracter o caracteres que actúan de separadores. Por ejemplo, la división de una frase en palabras, donde el separador será el espacio en blanco.

La acción de extraer una secuencia de tokens a partir de la cadena recibe la denominación de análisis lexicográfico.

Tenemos tres contructores:

- StringTokenizer tokens = new StringTokenizer(str);
- StringTokenizer tokens = new StringTokenizer(str, strDelim);
- StringTokenizer tokens = new StringTokenizer(str, strDelim, bInclDel);

En el primer caso, donde no especificamos una cadena con los separadores, se toma como separador el delimitador por defecto " \n\t\r" (cualquiera de ellos).

En el constructor le asignamos la cadena de la que queremos extraer las unidades lexicográficas, y a continuación, podemos ir recorriendo los tokens extraidos con los métodos: countTokens(), hasMoreTokens(), nextToken().

El método nextToken() devuelve un String.

Clase Object

Es la clase raíz de la jerarquía de clases de la biblioteca Java. Pertenece al paquete Java.lang. Cualquier clase que implementemos en nuestras aplicaciones pasará a ser automáticamente una subclase, heredamos sus métodos. Algunos de ellos son:

equals = Comparación del contenido de dos objetos de la misma clase.
clone = copiar un objeto
toString = proporciona una representación en cadena del objeto

Conceptos avanzados de Java

Paquetes

El software se construye a partir de componentes existentes, bien definidos, probados, documentados y reutilizables. Este tipo de reutilización de software acelera el desarrollo de software. El concepto de creación rápida de aplicaciones (RAD, Rapid Applications Development) es muy importante en nuestros días.

Para aprovechar todo el potencial de la **reutilización de código**, necesitamos mejorar los esquemas de organización y catalogación del mismo, mecanismos de protección (para asegurarnos de que no sufren modificaciones por terceras personas), mecanismos de navegación o consulta de las clases existentes.

Podemos agrupar un conjunto de clases e interfaces desarrolladas por nosotros y darle un nombre, formando **paquetes** (packages). Un paquete puede contener además de clases otros paquetes, formando una estructura jerárquica.

A los paquetes les llamamos bibliotecas de clases Java o Interfaz de Programación de Aplicaciones de Java (Java API).

La biblioteca de clases de Java también está estructurada en paquetes de forma jerárquica, siendo el de primer nivel el paquete "**java**". Tan solo incorpora otros paquetes.

Una de las grandes ventajas de Java es el gran número de clases contenidas en los paquetes de la Java API para que los programadores puedan utilizarlas sin tener que volver a implementar.

Los **paquetes de clases** más importantes que contiene Java son:

- **java.lang**: Paquete del lenguaje Java (Java Language Package)
  Elementos básicos del lenguaje como operaciones de entrada/salida (in/out). El compilador lo importa automáticamente en todos los programas.
- **java.util**: Paquete de utilidades de Java (Java Utilities Package)
  Contiene ciertas estructuras de datos (pilas, colas, etc.) y clases generales que pueden ser de utilidad.
- **java.io**: Paquete de entrada/salida de Java (Java Input/Output Package).
  Contiene clases de acceso a ficheros (archivos y flujos).
- **java.applet**: Paquete de applets de Java (Java Applet Package).
  Necesario para realizar applets ejecutables por los navegadores de Internet.
- **java.net**: Paquete de trabajo con redes de Java (Java Networking Package).
  Contiene clases basadas en el protocolo TCP/IP para la comunicación de sistemas.
- **java.awt**: Paquete de herramientas abstractas para trabajar con ventanas Java (Abstract Windows Toolkit Package).
  Contiene una serie de clases para acceder al sistema de ventanas (interfaces gráficas de usuario) de diversas plataformas: Windows, Macintosh, Motif (Unix).
- **Predeterminado (sin nombre)**: Contiene todas las clases que hemos desarrollado nosotros sin incluirlos en un paquete concreto.

Para **acceder a una clase** (y a sus métodos y propiedades) dentro de un paquete tenemos dos opciones:

1. **Referenciar** completamente su nombre, especificando toda la jerarquía de paquetes separados con un punto y el nombre de la clase:

   java.lang.System.out.println("Hola mundo");

2. **Importar** una clase utilizando la sentencia *import*, con lo que nos ahorramos tener que especificar toda su jerarquía cada vez que accedamos a sus métodos y propiedades, es decir, utilizaríamos directamente el nombre de la clase:

   import *java.lang.System;*
   ...
   System.out.println("Hola mundo");

   La única excepción a esto es el paquete de clases '*lang*', que es importado de forma automática. Por eso, no es necesario importar sus clases explícitamente, como puede comprobarse en el ejemplo anterior "HolaMundo.java".

   Podemos importar todas las clases públicas de un paquete en una sola línea, especificando el carácter '*' como nombre de clase a importar. Por ejemplo:

   import java.util.;*

   En este caso, cuando importamos un paquete completo, el compilador sólo importa las clases que se usan en el programa.

La **ventaja** de tener todos estos paquetes incluidos en la librería de Java es que el programador tiene a su disposición una serie de clases ya implementadas y probadas que facilitan el trabajo posterior.

La **desventaja** es que debido a la cantidad de paquetes y clases que incorpora, la curva de aprendizaje es muy larga.

Las **ventajas del uso de paquetes** son:

- Reducir los posibles conflictos de nombres
- Controlar la visibilidad de las clases, interfaces y métodos, así como los datos que se definen en ellos.

Para **especificar un paquete** en Java debemos poner como primera línea de código del fichero la siguiente:

package <NombrePaquete>;

Si no especificamos la sentencia package, las clases que se generen se guardarán dentro del paquete predeterminado (sin nombre).

A continuación podemos importar clases de otros paquetes, con la sentencia **import**, formando así la estructura jerárquica comentada. Del mismo modo, otros paquetes podrán importar clases (e interfaces) de este paquete, siempre y cuando hayan sido definidas como públicas (**public**):

import <NombrePaquete>. ... .<NombreClase>;

A la hora de elegir el **nombre de un paquete**, Sun Microsystems recomienda utilizar nuestro dominio de Internet en orden inverso, seguido de la jerarquía del paquete en sentido descendente.

Por **ejemplo**, suponiendo que los desarrolladores sean Sun Microsystems, cuyo dominio es www.java.sun.com, creasen un paquete llamado *impresora*, dentro del paquete *util*, el nombre recomendado sería:

com.sun.java.util.impresora

El seguir esta recomendación tiene varias **ventajas**:

1. No habrá conflictos, pues el nombre no coincidirá con el elegido por nadie, ya que no es posible encontrar dos nombres de dominio iguales. Recordemos que un programa Java puede ejecutarse a través de Internet en cualquier parte del mundo (applets).
2. Cualquier usuario de un paquete podrá conocer quien lo desarrolló.

Por último, la estructura de paquetes definida se almacena en una estructura de **directorios en el disco**, de tal forma que exista una correspondencia. El directorio raíz donde comienza la estructura jerárquica debemos especificarlo en la variable de ambiente **CLASSPATH**, en el fichero Autoexec.bat. En dicha variable indicamos al compilador todas las rutas donde debe buscar paquetes de clases, separadas por ';'.

Por **ejemplo**, si nuestras clases las agrupamos en un paquete llamado 'libreria', el directorio raíz deberá ser c:\java\jdk1.3\libreria, y la variable CLASSPATH deberá tener:

Set **CLASSPATH** = . ; c:\java\jdk1.3 ; c:\java\jdk1.3\libreria (Windows)

ó

setenv CLASSPATH . : /users/jdk1.3 : /users/jdk1.3/libreria (Unix)

Otra alternativa a esta variable de ambiente sería el uso del parámetro "**-classpath"** que vimos anteriormente.

Podemos guardar toda una estructura de archivos dentro de ficheros *.zip y *.jar. En la variable de ambiente CLASSPATH debemos especificar el nombre del fichero, con la ruta completa.

Interfaces

Una interfaz es una colección de constantes y de métodos abstractos, que serán implementados por una clase concreta.

Los interfaces se utilizan para poder implementar el equivalente a herencia múltiple entre clases que tiene C++, pero eliminando muchos de sus inconvenientes.

Para definir un interface se utiliza la siguiente sintáxis, exáctamente igual que una clase:

```
[public] interface <NombreInterfaz> extends <Lista SuperInterfaces>
{
        // Constantes
        ...
        // Declaraciones de métodos (virtuales)
        ...
}
```

Un interface contiene **declaraciones de constantes y métodos**. La claúsula **extends** tiene el mismo significado que para las clases, es decir, indicamos que el interfaz hereda de otros (en este caso, herencia múltiple): constantes, atributos y métodos públicos.

La definición de una interfaz es exactamente igual que la definición de una clase, salvo que únicamente se permitirá el uso de las constantes y de los métodos abstractos. Por defecto, todas las interfaces son abstractas, y de forma implícita:

Constantes: publicas y estáticas (public final static)
Métodos: abstractos y públicos (public abstract).
No será necesario precederlos con los modificadores: final, static o public.

Al declarar una clase, vimos que existía una claúsula **implements**. Esto indica que la clase va a implementar los métodos de las interfaces especificadas en la lista. Una clase puede implementar cualquier número de interfaces:

[**implements** <Lista Interfaces>]

Una clase que implemente una interfaz, puede acceder a las **constantes del interfaz**, bien mediante el nombre del interfaz, o bien, directamente, pues las ha heredado.

La **implementación de los métodos** de un interfaz, debe estar en cada una de las clases que utilicen dicho interfaz, debiendo cumplir además, dos condiciones:

- Debemos implementar todos los métodos del interfaz, aunque no se vayan a utilizar.
- Si el método era público, la implementación del método también deberá serlo.

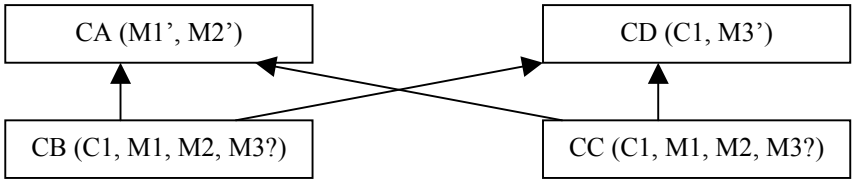
Los métodos que se declaran en las interfaces, son sólo plantillas de comportamiento. Una vez implementados los **métodos de la interfaz** en una clase, ya son métodos en toda regla, con lo cual, cualquier subclase que herede de esta clase los heredará, junto con las constantes definidas en el interfaz.

Aparentemente, el **uso de interfaces** y el de **clases abstractas** es similar, pero no es así. En el caso de que tengamos una serie de clases que sean subclase de otra, y que necesiten todas ellas una sóla interfaz, perfectamente podría sustituirse esta interfaz por una clase abstracta. Con esto queremos decir que, el uso de interfaces nos permite implementar la **herencia múltiple**, pero también la **herencia simple**.

Ejemplo

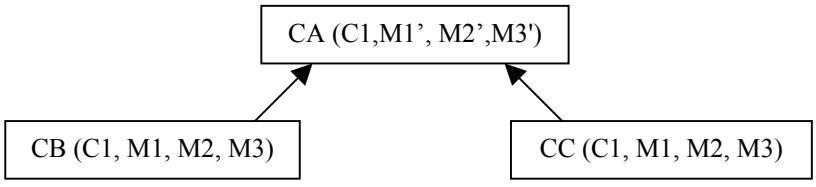
Supongamos una clase A, de la que heredan dos subclases B y C. La superclase A tiene dos métodos virtuales M1 y M2 (abstractos), que deberán ser implementados en las subclases B y C.

Imaginemos ahora que tenemos otra clase D, con una constante C1 y un método virtual M3, y queremos que este método tambien se utilice en las dos subclases B y C.

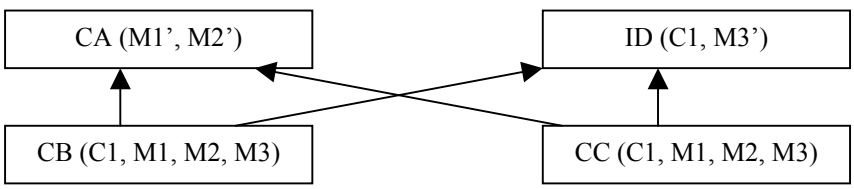


Como Java no permite la herencia múltiple entre clases, tendríamos las siguientes **opciones**.

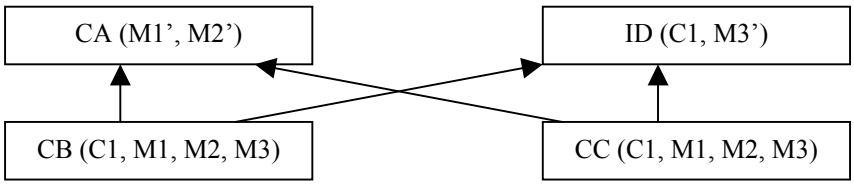
Solución 1: Eliminar la clase D, copiando el método virtual M3 a la superclase A, junto con sus atributos, constantes, etc., pero esto afectaría a la reutilización de código, ya que la clase D podría utilizarse en otros sitios (podrían ser muchos métodos).



Solución 2: Definir la clase D como una interfaz, que implementarían las subclases B y C. En este caso tendríamos una herencia simple entre clases, y una implementación de interfaz.



Solución 3: Definir tanto la superclase A como la D como interfaces. En este caso, no existiría herencia entre clases, tan sólo, la implementación de dos interfaces. Puede que esta opción no sea viable, si la superclase A tuviese algún método no virtual.



Una vez comprendido el ejemplo anterior es el momento de aclarar que el uso de interfaces no es siempre una **alternativa a la herencia múltiple** (opción 3 del ejemplo). En el ejemplo, como la superclase A sólo tenía constantes y métodos virtuales, si se podría haber utilizado como alternativa a la herencia múltiple, pués al implementar una interfaz, se heredan las constantes. Pero en el caso de que la superclase A tuviera algún método no virtual, ya no podríamos haberlo hecho así, pues, como hemos explicado, una interfaz sólo puede tener declaraciones de métodos, no su implementación.

La herencia múltiple y el uso de interfaces es distinto:

- Con interfaces, heredamos constantes y definiciones de métodos, pero no métodos.
- Por otro lado, las **jerarquías de clases y de interfaces** son independientes. Varias clases pueden implementar una interfaz y no pertenecer a la misma jerarquía de clases. En herencia múltiple, todas las clases pertencen a la misma jerarquía.

Usos de las interfaces:

- **Como tipo de datos**. Al definir una variable de tipo interfaz, tendremos que asignarle la referencia a un objeto cuya clase implemente dicha interfaz. El nombre de una interfaz se puede utilizar en cualquier lugar donde se pueda utilizar el nombre de una clase.
- **Forzar a que una clase tenga que implementar ciertos métodos**.
- **Reducción de código**, al compartir todas las clases que utilizan una interfaz las definiciones de los métodos, abstrayendo el comportamiento de los mismos.

Una variable de tipo interfaz espera referenciar un objeto que tenga implementada dicha interfaz. Si no es así, tendremos un error de compilación.

Además, sólo tiene acceso a los métodos y constantes declaradas por dicha Interfaz.

Gestión de errores (Excepciones)

Captura de errores

Cuando se produce algún error durante la ejecución de un programa, el intérprete Java aborta su ejecución, mostrando un mensaje. Java dispone de unas sentencias para poder capturar los errores en tiempo de ejecución, pudiendo nosotros decidir la acción a realizar en el caso de que se produzca un error, y sin que se detenga la ejecución del programa.

Se trata de las sentencias de **manejo de excepciones**: *try ... catch ... finally*.

```
try {
        <Bloque de sentencias a supervisar>
}
catch ( <TipoExcepcion> ) {
        <Bloque de sentencias a ejecutar en caso de error>
}
finally {
        <Bloque de sentencias a ejecutar en caso de error por defecto>
}
```

La **ventaja** de estas sentencias es que separamos de forma explícita el código que gestiona los errores del código del programa propiamente dicho, obteniendo un código más claro que si utilizásemos sentencias de tipo: **if**, **switch**, etc. Además, el código se ejecutará de forma más eficiente, ya que no tendremos que estar continuamente comprobando condiciones que no son muy frecuentes o "imposibles".

Sentencia	Significado
try	Engloba el grupo de sentencias que queremos supervisar durante la ejecución.
catch (<TipoExcepción>)	En el caso de que en el bloque **try** se haya producido una excepción del tipo <TipoExcepción> se ejecutarán ese bloque de sentencias. Después de ejecutar sus sentencias, continúa desde donde termina, pues es posible que se haya producido más de una excepción, de tipos distintos, pudiendo así entrar en varios bloques **catch**.
finally	En el caso de que en el bloque **try** se haya producido cualquier excepción (haya sido capturada previamente o no por un bloque **catch)**, se ejecutará este bloque de sentencias. Se trata del gestor de excepciones por defecto, y nos permite liberar recursos (cerrar ficheros, etc.). También nos permite capturar ciertos errores que no pensamos que puedan producirse y no hemos capturado.

El **<TipoExcepción>** es un objeto que se instancia a partir de la clase correcta (**Throwable** o alguna de sus subclases), justo en el momento de producirse el error, y contiene información de la excepción producida.

En la estructura jerárquica de la librería de clases de Java, tenemos las clases que implementan cada uno de los distintos tipos de error:

Java.lang.**Throwable**
- **Exception**
  - RuntimeException
    - ArithmeticException
    - NullPointerException
    - NumberFormatException
    - ArrayIndexOutOfBoundsException
  - ClassNotFoundException
  - IOException
    - EOFException
- **Error**

La clase raíz es **Throwable**, la cual tiene dos subclases que clasifican los dos grupos de tipos error posibles:

- **Exception**: son los errores habituales, capturables por el usuario. De esta clase se derivan otras subclases, e incluso podemos derivar nosotros nuestras propias clases.

- **Error**: son errores graves, en los que conviene no capturar su excepción y dejar que aborte el programa de forma anormal. Suelen ser errores de ejecución de la JVM.

Cuando se produce un error, se mira si hay una manejador para dicha excepción en el método donde se ha producido. Si no se captura la excepción, se va retornando en la **pila de llamadas,** hasta llegar a la función **main**() si no se hubiese capturado en ninguna las funciones o métodos. Si en esta función tampoco se captura la excepción, se ejecutará el **manejador de excepciones por defecto**, que consiste en abortar el programa mostrando un mensaje significativo del error y el lugar donde se ha producido (clase, método, número de línea, pila de llamadas, etc.).

Cuando utilizamos las clases de Java (o nuestras), hay métodos que lanzan una excepción (throw). Dependiendo del tipo de excepción estaremos obligados a capturar la excepción generada o no. Por esto, debemos distinguir dos **tipos de excepciones**:

- **Excepciones implícitas**: no estamos obligados a capturarlas.
  Son todas las de RuntimeException y Error.

- **Excepciones explícitas**: si estamos obligados a capturarlas.
  Son ClassNotFoundException, IOException y EOFException.

Generación de errores

Otra posibilidad que tenemos en la gestión de errores es provocarlos de forma explícita. Para ello utilizaremos las palabras clave: **throw** y **throws**.

Vimos que la **declaración de un método** tenía la siguiente sintáxis:

```
[<Modif. método>] <TipoRetorno> <NombreMetodo> ( [<ListaParámetros>] ) [throws <Excepciones>]
{
        // Variables locales
        ...
        // Instrucciones o sentencias
        ...
        return (<Expresión>);
}
```

Con la claúsula **throws** indicamos todas las excepciones que no serán manejadas dentro del método, sino por el método que la llama. Estas excepciones podrán ser errores del programa como los vistos anteriormente, o bien, generados de forma explícita con **throw**.

La sintáxis de **throw** es la siguiente:

```
throw <ObjetoThrowable>;
```

El <ObjetoThrowable> es un objeto de clase Throwable o alguna de sus subclases. Tenemos dos opciones de obtener este objeto:

- Si estamos dentro de un bloque **catch**, el parámetro (<TipoExcepción>).
- Si estamos fuera de los bloques catch, dentro del código propiamente dicho de un método, crearlo con el operador **new**.

Al ejecutarse una sentencia **throw**, la ejecución salta al bloque catch correspondiente, de la misma forma que si se hubiera producido el error de forma natural.

Ventajas:

- Una clase informa de las posibles situaciones anómalas de la clase.
- Esas excepciones tendrá que redefinir el método que llame a este método de forma obligatoria. Esto nos permite decidir las acciones a tomar en caso de que se produzcan. El método que llama, o cualquier otro por encima en la pila de llamadas.

Por ejemplo, el método "FileWriter" se ha definido:

```
public FileWriter (String name) throws IOException {
        // ...
}
```

Si utilizamos el método "FileWriter" definido así, deberemos capturar la excepción "IOException" de forma obligatoria. Así podemos decidir lo que hacer en el caso de que se produzca un error de ese tipo.

Manejo de ficheros

Un **fichero** es un conjunto de datos almacenado en cualquier dispositivo informático. Los datos suelen ser un conjunto de registros. Un registro es un conjunto de campos con información individual.

Todos los sistemas operativos tienen varios tipos de ficheros diferentes.

Sistemas de ficheros (fs)

Son una estructura jerárquica organizada. Se componen de directorios que a su vez pueden contener ficheros y otros directorios (árbol de directorios).

Los fs son la parte más visible de un sistema operativo:

- **Visión de un usuario**: apariencia externa (árbol de directorios, nombres de los ficheros, operaciones que se pueden realizar (copiar, cambiar atributos, etc.)
- **Visión del diseñador de sistemas operativos**: Número de sectores de los bloques lógicos, cómo saber los bloques libres, detalles sobre el almacenamiento de ficheros (lista enlazada de bloques), etc. Todos estos detalles carecen de importancia para el usuario.

Algunas plataformas pueden tener 0 o más fs. El conjunto de fs variará según insertemos o extraigamos algún disco removible (Diskette, CD-Rom, Zip, etc.), o si montamos o desmontamos alguna unidad física o lógica (mount, umount, etc.).

Raíz de un sistema de ficheros

Cada sistema de ficheros tiene un directorio raíz, desde donde podremos almacenar ficheros y otros subdirectorios.

- Windows: tendremos un directorio raíz por cada unidad lógica (C:\, D:\, ...).
- Unix: tan sólo tendremos un directorio raíz denotado por "/". Un fs se puede montar en cualquier parte del árbol de ficheros, lo que permite acceder a todos los ficheros por su nombre de ruta, independientemente del dispositivo en donde residan.

Todos los ficheros de un sistema de ficheros comenzarán siempre por la cadena que denota la raíz de dicho sistema de ficheros.

Rutas

Cuando necesitamos utilizar **rutas** para especificar el directorio de un fichero, normalmente debemos tener en cuenta que se utiliza una técnica distinta dependiendo del sistema operativo en el que trabajemos.

Siempre tendremos una secuencia de nombres de directorio separados por un carácter separador que especifican la posición del fichero dentro de la estructura jerárquica de directorios del disco. Todos son nombres de directorio, excepto el último que será otro subdirectorio o el nombre de un fichero.

Lo que cambia de un sistema a otro será el carácter separador de nombres:

- Windows: '\' (Barra invertida) (NOTA: deberemos poner '\\' en las rutas).
- Unix: '/' (Barra normal)

Pero en Java nos es indiferente; podemos utilizar ambas formas, y el programa se ejecutará correctamente en cualquier plataforma. Es esta plataforma que la que define el carácter separador. Podemos acceder a las propiedades "separator" de la clase File para consultarlo.

LLamamos "**rutas abstractas**" a cualquier ruta expresada de cualquiera de las dos formas vistas (Windows o Unix), independientemente de la plataforma donde vayamos a ejecutar nuestra aplicación.

Podríamos utilizar indistintamente los siguientes nombres de fichero:

"\\Java\\Documentos\\Leeme.txt" ó "/Java/Documentos/Leeme.txt"

Podemos referenciar un fichero especificando **rutas absolutas o relativas**:

- **Absolutas**: son rutas completas, ya que incluyen la raíz del sistema de ficheros, y toda la jerarquía de directorios hasta llegar al fichero en cuestión, es decir, desde la raíz hasta el fichero.

- **Relativas**: son rutas que toman otra como referencia (p.e. el directorio actual), y se concatenan al final de ésta. Java considera como ruta de referencia el directorio actual o directorio de trabajo, que normalmente será el directorio desde donde invocamos la JVM para ejecutar un programa. Este directorio podremos consultarlo con la propiedad "user.dir". Todos los nombres de fichero que no comiencen por el directorio raíz son relativos al directorio de trabajo.

Flujos (streams)

Vamos a ver una serie de clases relacionadas con el manejo de ficheros. Todas ellas están contenidas en el paquete java.io.

La forma de acceder a un fichero es mediante el uso de streams ó flujos de información. Actúan de intermediarios entre el programa el origen (en lectura) o destino (en escritura) de la información, sin importarle de donde vienen (se leen) o a donde van los datos (se escriben). Esto nos proporciona independencia del dispositivo de almacenamiento, lo que se traduce en facilidad a la hora de escribir los programas (nivel alto de abstracción). Un programa podrá leer o escribir información indistintamente en diversos medios: memoria, fichero en el disco, impresoras o una tubería.

Un fichero deberá ser leido con el mismo formato con el que se escribió, de lo contrario obtendremos resultados inesperados.

Tuberías o conductos

Una **tubería** es un flujo que permite comunicar dos subprocesos (threads) de un programa para transferencia de información entre uno y otro. También permiten la comunicación de dos programas (procesos) distintos.

Clasificación de los flujos

Podemos **clasificar los flujos** desde distintos puntos de vista:

- Si son de entrada (lectura) o salida (escritura).
- Si trabajan con caracteres ASCII o Unicode (modo texto) o bytes (modo binario).
- Si procesan la información o no: algunos flujos procesan los datos de E/S, añadiendo un buffer, un filtro, realizando conversiones, etc.; otros leen/escriben sin realizar ningún otro proceso.

Flujos estándars

Cada vez que llamamos al método "System.out.println" estamos utilizando un flujo de salida. Hacemos referencia al método "println" de la propiedad "out" de la clase "System". La clase "System" tiene tres propiedades que representan flujos:

- **out**: es el flujo de salida por defecto (estándar) de la clase "PrintStream". Este flujo siempre está abierto y listo para aceptar datos de salida. Este flujo corresponde a la pantalla.
- **in**: es el flujo de entrada por defecto (estándar) de la clase "InputStream". Este flujo corresponde al teclado.
- **err**: es el flujo de salida para los errores estándar, de la clase "PrintStream". Normalmente representa al mismo dispositivo que el "out". Se utiliza para visualizar mensajes de error y otras informaciones importantes.

En todos los casos, podemos especificar **otro dispositivo** distinto, como por ejemplo, redirigir la salida a un fichero. Esto se hace con un método para cada uno de esos flujos:

- System.setIn (InputStream in)
- System.setOut (PrintStream out)
- System.setErr (PrintStream err)

Clase File

Con esta clase podremos acceder a las **propiedades de los ficheros**, pero no leer ni escribir en ellos. En algunos casos hablaremos de "rutas abstractas", es decir, rutas independientes de la plataforma.

Los **constructores** de esta clase son:

Constructor	Uso
File (String sPath)	Crear un directorio
File (String sPath, String sFileName)	Crear un fichero en el directorio especificado
File (File oFile, String sFileName)	Crear un fichero en el mismo directorio que el fichero especificado

Tenemos las siguientes **propiedades**:

Propiedad	Uso
separatorChar, pathSeparatorChar	Obtener el carácter separador en las rutas, dependiendo del sistema operativo, como un carácter (char). • Windows: '\' • Unix: '/'
separator, pathSeparator	Idem a los dos anteriores, pero lo retorna en forma de cadena (su único carácter es el anterior)

Los **métodos** más importantes de esta clase son:

Método	Uso
canRead()	Averiguar si podemos leer el fichero
canWrite()	Averiguar si podemos escribir en el fichero
delete()	Borrar el fichero del disco
deleteOnExit()	Borrar el fichero cuando la máquina virtual termine
renameTo()	Cambiar el nombre del fichero
exists()	Averiguar si el fichero existe
isAbsolute()	Averiguar si la ruta del fichero es absoluta, es decir, si va precedido de la raíz del sistema de ficheros: • Windows: letra de unidad y '\\' (C:\) • Unix: '/' (/dev/ttyS0)
getAbsoluteFile()	Obtener el nombre absoluto del fichero
getAbsolutePath()	Obtener la ruta absoluta del fichero
getName()	Obtener el nombre del fichero
getPath()	Obtener la ruta relativa del fichero, incluyendo su nombre
getParent()	Obtener la ruta del directorio padre, el que contiene al fichero
isFile()	Averiguar si el objeto es un fichero normal

isDirectory()	Averiguar si el objeto es un directorio
length()	Obtener el tamaño del fichero
lastModified()	Obtener la fecha/hora de la última modificación
setLastModified()	Establecer la fecha/hora de la última modificación
setReadOnly()	Establecer el fichero como de solo lectura
isHidden()	Averiguar si el fichero es un fichero oculto
mkdir()	Crea un directorio bajo el directorio actual
mkdirs()	Crea un directorio, incluyendo todos los subdirectorio inexistentes
String[] list()	Devuelve un array de cadenas con los nombres de ficheros y de directorios contenidos denotados por la ruta abstracta del objeto File.
String[] list(FilenameFilter)	Idem al anterior, pero que satisfacen el filtro especificado. "FilenameFilter" es una interfaz con un solo método "accept". Ver más detalles abajo.
File[] listFiles()	Idem al método "list", pero solo incluye los ficheros, descartando los directorios.
File[] listFiles(FileFilter)	Idem al anterior, pero que satisfacen el filtro especificado. "FileFilter" es una interfaz con un solo método "accept". Ver más detalles abajo.
File[] listFiles (FileNameFilter)	Idem al anterior, utilizando como filtro la intefaz FileNameFilter.
File[] listRoots()	Devuelve un array con la raíz de todos los sistemas de ficheros. Ver más detalles abajo.

listRoots

El método "listRoots" devuelve un conjunto de objetos "File" que denotan la raíz de sistemas de ficheros. En este conjunto se incluye siempre la raíz de los sistemas de ficheros locales en nuestra máquina. En cuanto a los sistemas de ficheros remotos, puede que sí o puede que no, dependiendo de si el nombre completo de algún fichero remoto coincide con algún fichero local:

- Windows: todas las unidades remotas "mapeadas", es decir, aquellas a las que les hemos asignado una letra lógica, si formarán parte del conjunto devuelto, ya que siempre podremos distinguir los ficheros remotos de los locales. En cambio, las raíces de sistemas de ficheros que contengan nombres UNC no se incluirán en la lista.
- Unix: las unidades accedidas por SMB o NFS puede que si y puede que no, según la condición anterior.

NFS (Network Filing System): Es el sistema de ficheros utilizado en Unix. Fué desarrollado por Sun Microsystems y se caracteriza por ser compartible y distribuido.

Los **nombres UNC** son aquellos que tienen el siguiente formato:

"\\ Hostname \ SharedDevice \ Subdirectorios"

Interfaz FilenameFilter

Este interfaz se utiliza para fijar unos criterios de aceptación o rechazo a la hora de obtener los nombres de los ficheros de un directorio concreto (métodos list() de la clase File). Es decir, establecemos un filtro.

Tan sólo declara el método "**accept**()", que recibe dos parámetros:

- El objeto File con el que se llamó al método "list()", que en este caso representará un directorio.
- Un objeto String, que indica el nombre de uno de los ficheros (o subdirectorios) que incluye el directorio especificado en el objeto File.

Este método se llama para cada uno de los ficheros (o directorios) de dicho directorio. Lo que debemos hacer en este método es retornar true o false, en función de si aceptamos o rechazamos el fichero, bajo cualquier criterio nuestro. Sólo los ficheros que aceptemos se incluirán en la lista de ficheros que retornan los métodos "list()" del objeto File.

Esta interfaz también se utiliza por las ventanas de diálogo de selección de ficheros en el interfaz gráfico AWT (Abstract Window Toolkit).

Interfaz FileFilter

Es similar al interfaz FilenameFilter. Declara también un método "accept()", con la diferencia de que en este caso tan sólo recibe un parámetro de tipo File, que contiene el nombre del fichero que vamos a chequear.

Al disponer del objeto File para cada uno de los ficheros a chequear en vez de su nombre, nos permite acceder a todos los métodos de la clase File, y establecer criterios más complejos: por fecha, por el nombre, por el tamaño, etc. En el caso de la interfaz FilenameFilter también podríamos hacerlo, pero creándonos un objeto File a partir del nombre.

Ficheros de acceso secuencial

Los ficheros secuenciales son ficheros que se escriben y se leen desde el principio hasta el final. Tenemos la posibilidad de crearlos y manejarlos de forma binaria, más apropiada para almacenar valores numéricos, o en modo texto, que tan sólo contendrán caracteres ASCII o Unicode.

Superclases Writer / Reader (Modo texto)

Son clases abstractas para escribir y leer flujos de caracteres (char). Son las superclases de todas las clases que manejan flujos de caracteres. Los pocos métodos que tienen lo pueden sobreescribir otras subclases.

Clase Writer	
Constructor	**Uso**
Writer ()	Abrir un flujo de escritura, donde sus secciones críticas se ejecutarán de forma sincronizada consigo mismo
Writer (Object lock)	Abrir un flujo de escritura, donde sus secciones críticas se ejecutarán de forma sincronizada con el objeto especificado

Clase Writer	
Método	**Uso**
close	Cierra el flujo
flush	Confirma los cambios
write (int c)	Escribe un carácter
write (char[] c)	Escribe un número determinado de caracteres en un array (todo lo que quepa en el array: c.length)
write (char[] c, int off, int len)	Escribe el número especificado en 'len' de caracteres en un array, comenzando en la posición 'off'
write (String s)	Escribe una cadena
write (String s, int off, int len)	Escribe una subcadena

Clase Reader	
Constructor	**Uso**
Reader ()	Abrir un flujo de lectura, donde sus secciones críticas se ejecutarán de forma sincronizada consigo mismo
Reader (Object lock)	Abrir un flujo de lectura, donde sus secciones críticas se ejecutarán de forma sincronizada con el objeto especificado

Clase Reader	
Método	**Uso**
close	Cerrar el flujo y liberar recursos
mark	Establer la marca de posición en el stream
markSupported	Averiguar si el stream actual soporta mark/reset
read ()	Leer el siguiente carácter
read (char[] c)	Leer un número determinado de caracteres en un array (todo lo que quepa en el array: c.length)
read (char[] c, int off, int len)	Leer 'len' caracteres desde el flujo en el array 'c', colocándolos a partir de la posición 'off'
ready	Averiguar si el flujo está listo para lectura
reset	Retorna el buffer hasta la posición previamente marcada con 'mark'
skip (long n)	Saltar 'n' bytes desde el flujo de entrada

Superclases OutputStream / InputStream (Modo binario)

Son clases abstractas para escribir y leer flujos de bytes. Son las superclases de todas las clases que manejan flujos de bytes. La clase "InputStream" es la clase para los flujos de entrada estándard del sistema (teclado).

Clase OutputStream	
Constructor	**Uso**
OutputStream ()	Abrir un flujo de escritura de bytes

Clase OutputStream	
Método	**Uso**
close	Cierra el flujo
flush	Confirma los cambios
write (int b)	Escribe un byte
write (byte[] c)	Escribe un número determinado de bytes en un array (todo lo que quepa en el array: c.length)
write (byte[] c, int off, int len)	Escribe el número especificado en 'len' de bytes en un array, comenzando en la posición 'off'

Clase InputStream	
Constructor	**Uso**
InputStream ()	Abrir un flujo de lectura de bytes

Clase InputStream	
Método	**Uso**
available	Averiguar el número de bytes que se pueden leer desde el flujo sin que se bloque.
close	Cerrar el flujo y liberar recursos
mark	Establer la marca de posición en el stream
markSupported	Averiguar si el stream actual soporta mark/reset
read ()	Leer el siguiente carácter
read (byte[] c)	Leer un número determinado de bytes en un array (todo lo que quepa en el array: c.length)
read (byte[] c, int off, int len)	Leer 'len' bytes desde el flujo en el array 'c', colocándolos a partir de la posición 'off'
reset	Retorna el buffer hasta la posición previamente marcada con 'mark'
skip (long n)	Saltar 'n' bytes desde el flujo de entrada

Clases FileWriter / FileReader (Modo texto)

Como hemos visto, FileWriter y FileReader eran dos clases que trabajan con flujos de caracteres (char). Los métodos que tienen las clases FileWriter y FileReader son los que heredan de las clases Writer y Reader respectivamente, y los métodos de la clase Object que siempre se suponen. Además, estas clases implementan los métodos de las interfaces OutputStreamReader y InputStreamReader.

Los **constructores** son los siguientes:

Clase FileWriter	
Constructor	**Uso**
FileWriter (String fileName)	Abrir un flujo de escritura hacia el fichero especificado (sobreescribiendo si ya existe).
FileWriter (String fileName, boolean bAppend)	Abrir un flujo de escritura hacia el fichero especificado (añadiendo al final si ya existe).
FileWriter (File oFile)	Abrir un flujo de escritura a partir de un objeto File ya creado.

Clase FileReader	
Constructor	**Uso**
FileReader (String fileName)	Abrir un flujo de lectura desde el fichero especificado.
FileReader (File oFile)	Abrir un flujo de lectura a partir de un objeto File ya creado.

Clases FileOutputStream / FileInputStream (Modo binario)

La clase FileOutputStream nos permite acceder a los flujos de salida para escribir bytes a un fichero o "FileDescriptor". En determinados entornos sólo es posible abrir una vez un fichero para escritura; en estos casos, si se intenta abrir de nuevo obtendremos un error.

La clase FileInputStream nos permite acceder a los flujos de entrada para leer bytes desde un fichero.

Los **constructores** y métodos más importantes de la clase FileOutputStream son:

Clase FileOutputStream	
Constructor	**Uso**
FileOutputStream(File oFile)	Abre un flujo de información de salida de un fichero
FileOutputStream (FileDescriptor fdObj)	Obtiene el flujo de salida desde un flujo ya abierto. En ciertos entorno podrá dar error.
FileOutputStream (String sName)	Abre un flujo de información de escritura en un fichero
FileOutputStream (String sName, boolean bAppend)	Idem al anterior, especificando si añadimos al final en caso de que ya exista o lo sobreescribimos.

Clase FileOutputStream	
Método	**Uso**
close ()	Cerrar el flujo y liberar todos los recursos utilizados
finalize ()	Asegurarse de que se llamará al método close() cuando ya no hayan más referencias al flujo.
getFD ()	Obtiene el "FileDescriptor" asociado con el flujo
write (int b)	Escribe el byte especificado
write (byte[] b)	Escribe todos los bytes de un array (b.length)
write (byte[] b, int off, int len)	Escribe el número especificado en 'len' de bytes de un array, comenzando desde la posición 'off'

Los **constructores** y métodos más importantes de la clase FileInputStream son:

Clase FileInputStream	
Constructor	**Uso**
FileInputStream (File oFile)	Abre un flujo de información de entrada de un fichero
FileInputStream (FileDescriptor fdObj)	Obtiene el flujo de entrada desde un flujo ya abierto
FileInputStream (String sName)	Abre un flujo de información de entrada de un fichero

Clase FileInputStream	
Método	**Uso**
available ()	Averiguar el número de bytes pendientes de ser leidos
close ()	Cerrar el flujo y liberar todos los recursos utilizados
finalize ()	Asegurarse de que se llamará al método close() cuando ya no hayan más referencias al flujo.
getFD ()	Obtiene el "FileDescriptor" asociado con el flujo
read ()	Lee un byte desde el flujo
read (byte[] b)	Lee un número determinado de bytes en un array (todo lo que quepa en el array: b.length)
read (byte[] b, int off, int len)	Lee el número especificado en 'len' de bytes en un array, comenzando en la posición 'off'
skip (long n)	Descartar los próximos 'n' bytes del flujo

Interfaces DataOutput / DataInput

Estos interfaces contienen declaraciones de métodos para escribir y leer flujos binarios.

Los implementan las clases:

- DataOutput: DataOutputStream, RandomAccessFile, ObjectOutput
- DataInput: DataInputStream, RandomAccessFile, ObjectInput

Esto quiere decir, que dichas clases están obligadas a implementar (definir) todos los métodos declarados en las interfaces que implementan. Con lo cual, vamos a explicar estos métodos, y ya no será necesario volver a explicar lo mismo cuando veamos dichas clases.

En la siguiente tabla aparecen estos métodos; todos los métodos write pertenecen a DataOutput y los métodos read a DataInput.

Interfaces DataOutput / DataInput	
Método	**Uso**
write / read (byte[] b)	Escribe/lee un número determinado de bytes en un array (todo lo que quepa en el array: b.length)
write / read (byte[] b, int off, int len)	Escribe/lee el número especificado en 'len' de bytes en un array, comenzando en la posición 'off'
writeBoolean / readBoolean	Escribe/lee un dato de tipo boolean
writeChar / readChar	Escribe/lee un dato de tipo char
writeChars / readChars	Escribe/lee un dato de tipo String como caracteres
writeByte / readByte	Escribe/lee un dato de tipo byte
writeBytes / readBytes	Escribe/lee un dato de tipo String como bytes
writeShort / readShort	Escribe/lee un dato de tipo short
writeInt / readInt	Escribe/lee un dato de tipo int
writeLong / readLong	Escribe/lee un dato de tipo long
writeFloat / readFloat	Escribe/lee un dato de tipo float
writeDouble / readDouble	Escribe/lee un dato de tipo double
writeUTF / readUTF	Escribe/lee una cadena de caracteres en formato UTF-8, donde los 2 primeros bytes especifican el número de bytes de datos.

El interfaz DataInput tiene otros métodos específicos:

Interfaz DataInput	
Método	**Uso**
skipBytes (int n)	Nos saltamos n bytes del flujo de entrada

Clases DataOutputStream / DataInputStream (Modo binario)

Se **utilizan** para escribir y leer datos de tipos primitivos (boolean, char, byte, short, int, long, float, double).

La **ventaja** de estos ficheros es que son independientes de la plataforma (recordar los conceptos de representación big-endian y little-endian).

El **inconveniente** de estas clases es que no pueden utilizarse para leer ficheros en modo texto o binario creados con otras clases, es decir, un flujo DataInputStream sólo podrá leer ficheros creados con DataOutputStream.

La forma de escribir o leer un fichero de este tipo es mediante el uso de un filtro. Los pasos son los siguientes:

- Creamos un flujo de salida / entrada (FileOutputStream / FileInputStream)
- Creamos un filtro a partir de ese flujo, es decir, un objeto DataOutputStream ó DataInputStream.
- Ya podemos leer o escribir datos primitivos a través de ese filtro.
- Al finalizar, cerramos el flujo.

Estas clases implementan los métodos definidos en por las interfaces DataOutput y DataInput respectivamente. En las siguientes tablas aparecen los métodos específicos que añaden estas clases, o bien, han heredado. Habría que añadir los métodos de las interfaces que implementan.

Clases DataOutputStream / DataInputStream	
Constructor	**Uso**
DataOutputStream (FileOutputStream fos)	Crea un filtro de salida a partir de un flujo de salida.
DataInputStream (FileInputStream fis)	Crea un filtro de entrada a partir de un flujo de entrada.

Clases DataOutputStream / DataInputStream	
Método	**Uso**
write / read	Escribe/lee un byte (heredado)
close	Cierra el flujo (heredado)

Los siguientes métodos ya son específicos para cada clase:

Clase DataOutputStream	
Método	**Uso**
flush	Confirma las escrituras realizadas
size	Retorna el número de bytes escritos en el fichero
mark	Pone una marca en el flujo de entrada (heredado)
markSupported	Chequeamos si el flujo de entrada soporta los métodos: mark y reset (heredado)
reset	Nos posicionamos en la marca establecida con "mark" (heredado)

Clase DataInputStream	
Método	**Uso**
available	Retorna el número de bytes que se pueden leer (heredado)
skip	Nos saltamos un byte del flujo de entrada (heredado)

Los métodos marcados como "heredados" son heredados de las clases FilterOutputStream ó FilterInputStream (según el caso).

Interfaces ObjectOutput / ObjectInput

Hemos visto que estas interfaces implementan, a su vez, las interfaces DataOutput y DataInput respectivamente. Cuando veamos las clases ObjectOutputStream y ObjectInputStream veremos que implementan las interfaces ObjectOutput y ObjectInput. Por tanto, estas últimas clases implementarán los métodos de dos interfaces cada una. Vamos a ver ahora los métodos concretos de estas interfaces nuevas:

Interfaz ObjectOutput	
Método	**Uso**
close	Cerrar el flujo
flush	Confirmar los cambios
write (byte[] b)	Escribir un número determinado de bytes en un array (todo lo que quepa en el array: b.length)
write (byte[] b, int off, int len)	Escribir el número especificado en 'len' de bytes en un array, comenzando en la posición 'off'
write (int b)	Escribir un byte
writeObject (Object obj)	Escribir el objeto especificado de cualquier clase

Interfaz ObjectInput	
Método	**Uso**
close	Cerrar el flujo
available	Averiguar el número de bytes que se pueden leer
int read ()	Leer un byte
read (byte[] b)	Leer un número determinado de bytes en un array (todo lo que quepa en el array: b.length)
read (byte[] b, int off, int len)	Leer el número especificado en 'len' de bytes en un array, comenzando en la posición 'off'
readObject (Object obj)	Leer un objeto previamente almacenado
skip (long n)	Saltarse n bytes de la entrada

Clases ObjectOutputStream / ObjectInputStream (Escritura/Lectura de objetos)

La clase ObjectOutputStream no permite escribir en un fichero tanto tipos de datos primitivos, como objetos (Strings, arrays y objetos de cualquier clase), con lo que conseguimos objetos persistentes que más tarde se leerán con la clase ObjectInputStream. Lo que realmente se almacena serán los atributos del objeto (su estado).

Al proceso de escritura se le llama **seriación**, y al de lectura, **deseriación**. La única condición que deben cumplir las clases de las que vayamos a serializar sus objetos es que deben implementar la interfaz **java.io.Serializable**, la cual es una interfaz vacía (sin métodos) y cuyo único propósito es el de informar al compilador las clases de las que nos interesa serializar sus objetos. Es decir, si queremos que una de nuestras clases pueda ser serializada, debemos utilizar la claúsula "implements Serializable" en la declaración de la clase.

Cuando se **serializa** un objeto, al final se escribe utilizando la clase FileOutputStream. Del mismo modo, antes de **deserializar** un objeto almacenado, éste se lee desde el fichero con la clase FileInputStream. Es decir, debemos proceder del mismo modo que con las clases DataOutputStream/DataInputStream, creando un filtro.

Si se han almacenado objetos y tipos primitivos, se deben recuperar en el mismo orden en el que se almacenaron.

Los **constructores** de la clase ObjectOutputStream son:

ObjectOutputStream	
Constructor	**Uso**
ObjectOutputStream (OutputStream os)	Construye un objeto para escribir en el Stream de salida especificado. Puede ser OutputStream o alguna de sus subclases.

Los **métodos** son los que implementa de la interfaz DataOutput y la interfaz ObjectOutput. Para recordar, el más importante era:

ObjectOutputStream	
Método	**Uso**
writeObject (Object obj)	Escribir el objeto especificado de cualquier clase

Los **constructores** de la clase ObjectInputStream son:

ObjectInputStream	
Constructor	**Uso**
ObjectInputStream (InputStream is)	Construye un objeto para leer del Stream de entrada especificado. Puede ser de la clase InputStream o alguna de sus subclases.

Los **métodos** son los que implementa de la interfaz DataInput y la interfaz ObjectInput. Para recordar, el más importante era:

ObjectInputStream	
Método	**Uso**
readObject ()	Lee un objeto almacenado

Cuando escribimos un objeto que tiene referencias a otros objetos (de otros tipos o del mismo), todos estos objetos también se escriben, de forma recursiva.

Del mismo modo, cuando leamos un objeto, recuperaremos también todas estas referencias, para mantener la relación que existía entre ellos cuando se almacenaron.

Ficheros de acceso directo

Clase RandomAccessFile

Nos permite el acceso aleatorio (o directo) para escritura y para lectura. Tenemos un **puntero** de fichero que apunta a la posición donde vamos a leer o escribir, sin necesidad de tener que leer desde el principio hasta el final. Este puntero se coloca al principio del fichero cuando lo abrimos. Una vez situado el puntero en una posición determinada, cualquier operación de lectura/escritura se realizará en dicha posición, y el puntero se quedará apuntando a la siguiente posición del último byte escrito o leido.

Esta clase se utiliza tanto para operaciones de lectura como de escritura. Implementa las interfaces DataInput y DataOutput, que ya hemos comentado anteriormente. Esto quiere decir que tendremos a nuestra disposición una serie de métodos para escritura / lectura de datos de tipos primitivos (read, readBoolean, write, writeFloat, etc.).

Los **constructores** de esta clase son:

Constructor	Uso
RandomAccessFile (File oFile, String sMode)	Abre un flujo de lectura de un fichero y, opcionalmente, también de escritura
RandomAccessFile (String sFileName, String sMode)	Idem, especificando el nombre del fichero

NOTA: El modo en estos casos puede ser “r” = Sólo lectura; “rw” = Lectura/Escritura.

Los **métodos** más importantes de esta clase son:

Método	Uso
close ()	Cierra el fichero y libera todos los recursos
getFD ()	Obtiene el “FileDescriptor” asociado con el flujo
length ()	Averiguar la longitud del fichero
setLength (long nNewLong)	Establece la longitud del fichero
getFilePointer ()	Obtiene el desplazamiento del puntero de fichero
seek (long nPos)	Establece el desplazamiento del puntero de fichero, contando desde el principio del fichero
skipBytes (int n)	Se salta el número de bytes especificado de la entrada
read (int b)	Lee el byte especificado
read (byte[] b)	Lee todos los bytes de un array (b.length)
read (byte[] b, int off, int len)	Lee el número especificado en ‘len’ de bytes de un array, comenzando desde la posición ‘off’
readBoolean, readByte, readBytes, readChar, readChars, readDouble, readFloat,	Lee un dato de tipo primitivo del fichero. El número de bytes dependerá del tipo de dato.

readInt, readLong, readShort, readUTF	
write (int b)	Escribe el byte especificado
write (byte[] b)	Escribe todos los bytes de un array (b.length)
write (byte[] b, int off, int len)	Escribe el número especificado en 'len' de bytes de un array, comenzando desde la posición 'off'
writeBoolean, writeByte, writeBytes, writeChar, writeChars, writeDouble, writeFloat, writeInt, writeLong, writeShort, writeUTF	Escribe un dato al fichero. El número de bytes dependerá del tipo de dato.

Normalmente se utilizan para almacenar **registros de tamaño constante**, debido a que es muy fácil realizar los cálculos para posicionarnos en el byte correspondiente al principio de un registro concreto, aunque esto no tiene por qué ser así.

Supongamos que tenemos un tamaño de registro fijo de 50 bytes. Si quisiéramos colocarnos al principio del registro 27 haríamos lo siguiente:

```
seek ( 27 * 50 )
```

El primer registro es el 0. Conviene utilizar constantes para indicar el tamaño del registro, para conseguir que nuestros programas sean más legibles.

El problema de no tener un tamaño fijo es calcular el inicio de un registro concreto. Se podría almacenar una matriz, donde sus elementos nos indicarían la posición relativa de cada uno de los registros, aunque esto es bastante peligroso, sobre todo si eliminamos registros.

La forma de **eliminar registros** en estos ficheros es poniendo alguna marca para indicar que dicho registro está eliminado, pero sin eliminarlo físicamente (algún campo específico (eliminado) o poniendo un valor concreto en algún campo (0). Esto es muy rápido. Periódicamente podríamos hacer un proceso de eliminación definitiva de todos los registros eliminados, para ahorrar espacio.

El proceso consistirá en crear un fichero temporal para almacenar todos los registros no borrados. Una vez almacenados, se cerrarán los dos ficheros, eliminaremos el fichero actual (File) y renombraremos el fichero temporal con el nombre del fichero original.

Conversión de flujos

Podemos convertir flujos de bytes (modo binario) a flujos de caracteres (modo texto) y viceversa.

Para ello utilizamos las clases: InputStreamReader e OutputStreamWriter.

Flujos para acceso a memoria

Podemos crear flujos de salida, para escribir en un array de la memoria, y de entrada, para leer desde un array de la memoria.

Modo texto: CharArrayWriter y CharArrayReader.
Modo binario: ByteArrayOutputStream y ByteArrayInputStream.

Clases CharArrayWriter / CharArrayReader

Clases ByteArrayOutputStream / ByteArrayInputStream

Los **constructores** de la clase ByteArrayOutputStream son:

Constructor	Uso
ByteArrayOutputStream ()	Creamos un buffer de salida.
ByteArrayOutputStream (int size)	En este caso especificamos el tamaño del buffer (en bytes).

Tenemos las siguientes **propiedades** ocultas:

Propiedad	Uso
byte[] buf	Buffer donde almacenamos los datos
int count	Número de bytes del buffer

Los **métodos** más importantes de esta clase son:

Método	Uso
close	Cierra el flujo de salida y libera recursos
reset	Inicializamos el contador del array, con lo que ignoramos toda la salida actual
size	Retorna el tamaño actual del buffer
byte[] toByteArray	Asignar el contenido del flujo a un nuevo array de bytes
toString () / toString (String enc)	Convierte el contenido del buffer en una cadena, traduciendo bytes a caracteres, en función de la codificación de caracteres por defecto de la plataforma. En el segundo caso, le pasamos como argumento el nombre de la codificación
write (byte[] b, int off, int len)	Escribe 'len' bytes desde el array especificado 'b' comenzando en la posición 'off' a este flujo
write (int b)	Escribe el byte especificado a este flujo
writeTo (OutputStream out)	Escribe el flujo actual al flujo de salida especificado (p.e. un fichero). Es equivalente a "out.write(buf, 0, count)"

Básicamente para escribir en un array utilizando este flujo haríamos lo siguiente. Abrir el flujo de salida ByteArrayOutputStream, ir escribiendo con los métodos 'write', y llamar al método 'toByteArray' para copiar el contenido actual del buffer del flujo a un array de bytes.

Los **constructores** de la clase ByteArrayInputStream son:

Constructor	Uso
ByteArrayInputStream(byte buf[])	Creamos un buffer de entrada, usando 'buf' como buffer
ByteArrayInputStream (byte[] buf, int off, int len)	Creamos un buffer de entrada, usando 'buf' como buffer

Tenemos las siguientes **propiedades** ocultas:

Propiedad	Uso
byte[] buf	Buffer que especificamos en los constructores
int count	Número de bytes del buffer (indice del último byte + 1)
int mark	Posición actualmente que señala en el buffer
int pos	Indice del siguiente byte a leer; buf[pos] contiene dicho byte

Los **métodos** más importantes de esta clase son:

Método	Uso
available	Averiguar el número de bytes que se pueden leer sin bloquear
close	Cerrar el flujo y liberar recursos
mark	Establer la marca de posición en el stream
markSupported	Averiguar si el stream actual soporta mark/reset
read ()	Leer el siguiente byte
read (byte[] b, int off, int len)	Leer 'len' bytes desde el flujo en el array 'b', colocándolos a partir de la posición 'off'
reset	Retorna el buffer hasta la posición previamente marcada con 'mark'
skip (long n)	Saltarse 'n' bytes desde el flujo de entrada

Otros dispositivos

Podemos utilizar las clases anteriores imprimir por una impresora o para comunicar dos procesos, en vez de leer o escribir en el disco. Lo único que tenemos que hacer es vincular un flujo estándar (cualquiera de los que hemos visto hasta ahora) con el dispositivo concreto.

- Flujos de salida: FileWriter, FileOutputStream, DataOutputStream, ObjectOutputStream, RandomAccessFile.
- Flujos de entrada: FileReader, FileInputStream, DataInputStream, ObjectInputStream, RandomAccessFile.

Aunque podemos utilizar cualquiera de estas clases, si utilizamos alguna en modo binario e imprimimos por impresora puede que obtengamos caracteres extraños, ya que las impresoras se consideran dispositivos ASCII.

Podemos utilizar las clases de **flujos de salida** (escritura) para imprimir por una impresora conectada a un puerto paralelo o serie. En MS-DOS, existe una serie de nombres de dispositivos de impresión predefinidos: PRN (predeterminado: LPT1), LPT1, LPT2, LPT3, COM1 y COM2. Los cuatro primeros corresponden a impresoras conectadas a puertos paralelo, y los dos últimos (COM?), a impresoras conectadas a los puertos serie. En Unix, los dispositivos son los ficheros /dev/lp0, /dev/lp1, /dev/ttyS0, etc.

Del mismo modo, podemos utilizar las clases de **flujos de entrada** (lectura) para leer desde el teclado. Tenemos los siguientes nombres de dispositivo para la consola:

- Windows: CON y AUX.
- Unix: /dev/xxx

Conductos o tuberías (pipeline)

Podemos comunicar dos subprocesos (hilos) mediante el uso de conductos, sin tener que recurrir a otros elementos: ficheros temporales, matrices, etc.

También permiten la comunicación de dos programas (procesos) distintos.

Para ello tenemos definidos los siguientes flujos:

- Modo texto: PipedWriter y PipedReader
- Modo binario: PipedOutputStream y PipedInputStream

Para todos ellos, debemos crear los dos extremos de la tubería: emisor y receptor (no importa el orden en el que los creemos):

```
PipedWriter emisor = new PipedWriter();
PipedReader receptor = new PipedReader(emisor);
```

O también podríamos haber hecho:

```
PipedReader receptor = new PipedReader();
PipedWriter emisor = new PipedWriter(receptor);
```

Clases PipedWriter / PipedReader

Los **constructores** de la clase PipedWriter son:

Clase PipedWriter	
Constructor	**Uso**
PipedWriter ()	Crea una tubería de escritura que aun no se ha conectado a una tubería de lectura (lector)
PipedWriter (PipedReader pr)	Crea una tubería de escritura y la conecta a una tubería de lectura (lector) previamente creada

Los **métodos** de la clase PipedWriter son los heredados de la clase Writer (close, flush, write's) y los siguientes:

Clase PipedWriter	
Método	**Uso**
connect (PipedReader pr)	Conecta la tubería de escritura a un lector

Los **constructores** de la clase PipedReader son:

Clase PipedReader	
Constructor	**Uso**
PipedReader ()	Crea una tubería de lectura que aun no se ha conectado a una tubería de escritura (emisor)
PipedReader (PipedWriter pw)	Crea una tubería de lectura y la conecta a una tubería de escritura (emisor) previamente creada

Los **métodos** de la clase PipedReader son los heredados de la clase Reader (close, mark, reset, skip, ready, read's) y los siguientes:

Clase PipedReader	
Método	**Uso**
connect (PipedWriter pr)	Conecta la tubería de lectura a un emisor

Una vez conectados un emisor y un receptor, todo lo que envíe el emisor lo recibirá el receptor. Si utilizamos el constructor sin parámetro, hasta que nosotros no nos conectemos a un lector con el método "connect", o un lector no se conecte a nosotros, o no habrá comunicación.

La comunicación con estas clases se realiza en modo texto (caracteres).

Clases PipedOutputStream / PipedInputStream

Los **constructores** de la clase PipepOutputStream son:

Clase PipedOutputStream	
Constructor	**Uso**
PipedOutputStream ()	Crea una tubería de escritura que aun no se ha conectado a una tubería de lectura (lector)
PipedOutputStream (PipedInputStream pis)	Crea una tubería de escritura y la conecta a una tubería de lectura (lector) previamente creada

Los **métodos** de esta clase son los heredados de la clase OutputStream (close, flush, write's) y los siguientes:

Clase PipedOutputStream	
Método	**Uso**
connect (PipedInputStream pis)	Conecta la tubería de escritura a un receptor

Los **constructores** de la clase PipepInputStream son:

Clase PipedInputStream	
Constructor	**Uso**
PipedInputStream ()	Crea una tubería de lectura que aun no se ha conectado a una tubería de escritura (emisor)
PipedInputStream (PipedOutputStream pos)	Crea una tubería de escritura y la conecta a una tubería de lectura (emisor) previamente creada

Los **métodos** de esta clase son los heredados de la clase InputStream (close, mark, reset, skip, ready, read's) y los siguientes:

Clase PipedInputStream	
Método	**Uso**
connect (PipedOutputStream pos)	Conecta la tubería de lectura a un receptor

Estas clases se utilizan del mismo modo que las anteriores, pero trabajan a nivel binario (bytes).

IMPORTANTE: Debemos tener en cuenta que si dentro de un programa se van a realizar comunicaciones por tuberías (piped), es muy recomendable crear el flujo de salida en un hilo (thread) y el flujo de entrada en otro hilo distinto. Si lo hacemos todo en un único hilo, se puede producir un bloqueo (abrazo mortal).

Interfaces gráficos (ventanas)

Introducción

Una aplicación ventana consta de un contenedor y componentes.

Para desarrollar aplicaciones con un entorno gráfico (ventanas y controles gráficos) tenemos dos posibilidades:

- **AWT (Abstract Window Toolkit):** Los controles son específicos de la plataforma.
- **Swing:** Los interfaces gráficos son compatibles con todas las plataformas (Windows, Motif, MAC, etc.), y tiene más controles que los AWT.

Ambos son un conjunto de herramientas para diseñar interfaces gráficos, pero son diferentes. Están dentro de la biblioteca de clases JFC (Java Foundation Classes).

Componente

Un **componente** es un objeto que tiene una representación gráfica, y que puede interactuar con el usuario.

Componentes peso pesado / peso ligero

Cada componente AWT tiene su propia ventana de una plataforma. En programas extensos, todas estas ventanas ralentizan el rendimiento y consumen gran cantidad de recursos. En cambio, los componentes Swing son dibujados como imágenes dentro de sus contenedores y no tienen una ventana de la plataforma propia, por lo que usan bastantes menos recursos del sistema. Por esto, a los componentes AWT se les llama componentes **peso pesado**, y a los componente Swing, componentes **peso ligero**.

Todos los componentes Swing se derivan de la clase JComponent, la cual se deriva de la clase AWT Container. Por tanto, todos los componentes Swing son componentes AWT. Podemos mezclar componentes AWT y Swing en un mismo programa. La tendencia es ampliar el conjunto de componentes Swing en futuras versiones del lenguaje, aunque AWT seguirá siendo necesario.

Para utilizar uno de ellos debemos importar el conjunto de clases correspondiente al **paquete** en el que está definido.

Paquete	AWT	Swing
Entorno gráfico	Java.awt	Javax.swing

En ambos casos tenemos los **controles o componentes** más comunes, aunque Swing, como hemos dicho, tiene muchos más. Algunos de ellos son:

Componente	AWT	Swing
Etiquetas	Label	JLabel
Cajas de texto	TextField / TextArea	JTextField / JTextArea
Casillas de selección	CheckBox	JCheckBox
Casillas de opción	Choice	JRadioButton
Listas de selección	List	JComboBox
Botones	Button	JButton
Barras de desplazamiento	ScrollBar	JScrollBar
Cuadro de dibujo	Canvas	No es necesario (dentro de JPanel)
Arbol	No disponible	JTree
Contenedores	Container: Panel, Applet, Window (Dialog, File-Dialog, Frame), ScrollBar.	JFrame, JDialog, JApplet, JPanel.

AWT (Abstract Windowing Toolkit)

Nos permite generar aplicaciones gráficas o "ventana", la cual contendrá otros componentes o controles. Las aplicaciones de tipo ventana se basan en la clase Frame de AWT: java.awt.Frame.

```
java.lang.Object
        java.awt.Component
                java.awt.Container
                        java.awt.Window
                                java.awt.Frame
```

Eventos

Cada vez que el usuario pulsa una tecla, mueve el ratón o hace click sobre un componente, se genera un evento que se envía al sistema de ventanas.

En Java, la gestión de eventos ha cambiado sustancialmente en las sucesivas versiones. Desde la versión 1.1 tenemos un modelo llamado **gestión de eventos delegado**, que consiste en decidir que eventos deben ser capturados, con el objetivo de mejorar la ejecución.

En este modelo, para que estos componentes respondan a las acciones del usuario, deben tener asociado un **manejador de eventos**. Una vez hecho esto, debemos definir un método para cada uno de los eventos en los que nos interese ejecutar alguna acción.

Resumiendo, para trabajar con interfaces gráficos debemos capturar y **gestionar los eventos** que nos interesen de los componentes que forman la ventana.

Para ello debemos seguir los siguientes **pasos**:

1. Identificar los eventos que nos interesa capturar.
2. Crear un manejador de eventos para cada uno.
3. Registrar el manejador de eventos.

Con un poco más de detalle sería:

1. **Identificar los eventos** que nos interesa capturar para cada componente, puesto que no interesa capturar eventos en los que no se vaya hacer nada, para mejorar la ejecución.
2. Crear un **manejador de eventos**, es decir, una clase construida por nosotros, que implementa alguna de las subinterfaces de la interfaz EventListener (superinterface), como por ejemplo: ActionListener, MouseListener, WindowListener, etc. En esta clase, definiremos los métodos de la interfaz como respuesta al evento. Como parámetro tendrá el evento producido (ActionEvent, MouseEvent, WindowEvent, etc.)
3. **Registrar el manejador de eventos**, es decir, asociar un manejador de eventos al componente que genera el evento. En función del tipo de eventos, tendremos un método "add...Listener" concreto, que reciben como parámetro el objeto de la clase que implementa los métodos de la interfaz.
4. Finalmente, puede que nos sea de utilidad **eliminar el registro** de un objeto listener, con el método "removeListener", para asignarle otro posteriormente (registrar).

Si hubiésemos asociado un mismo manejador de eventos a varios componentes (por ejemplo, varios botones es un formulario), para saber cual de los componentes ha generado el evento llamaremos al método: "**getSource**" del evento recibido como parámetro, el cual nos devolverá un objeto de tipo de "Object". Una vez obtenido este objeto, podremos conocer su nombre o de que clase es.

Tipos de manejadores de eventos

Existen muchos **tipos de manejadores de eventos**, en función de los eventos que nos interese capturar. Los más interesantes son:

Manejador (Interfaz)	Método para asociar el manejador a un componente	Eventos
ActionListener	addActionListener	Acciones sobre un componente.
AdjustementListener	addAdjustementListener	Acciones sobre componentes de tipo ScrollBar, o cuando un componente se redimensiona, se oculta, se mueve, etc.
FocusListener	addFocusListener	Acciones cuando un componente obtiene o pierde el foco.
ItemListener	addItemListener	Acciones al cambiar de valor.
KeyListener	addKeyListener	Acciones de pulsación de teclas.
MouseListener	addMouseListener	Acciones de pulsación de botones del ratón.
MouseMotionListener	addMouseMotionListener	Acciones de movimiento del ratón.
WindowListener	addWindowListener	Acciones sobre una ventana: maximizar, minimizar, cerrar, mover, etc.

En cuanto a los métodos que implementamos de estas interfaces, siempre reciben como parámetro un objeto con la información del evento. La clase de este objeto se corresponde con el tipo de evento generado.

Manejador (Interfaz)	Métodos	Clase del parámetro de los métodos
ActionListener	actionPerformed	ActionEvent
AdjustementListener	adjustementValueChanged	AdjustementEvent
FocusListener	focusGained, focusLost	FocusEvent
ItemListener	itemStateChanged	ItemEvent
KeyListener	keyPressed, keyReleased, keyTyped	KeyEvent
MouseListener	mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased	MouseEvent
MouseMotionListener	mouseDragged, mouseMoved	MouseEvent
WindowListener	windowActivated, windowClosed, windowClosing, windowDeactivated, windowDeiconified, windowIconified, windowOpened	WindowEvent

Cuando se produce un evento, podremos acceder a las propiedades y métodos del objeto pasado como parámetro. Vamos a ver algunos ejemplos:

- Cuando **pulsamos un botón del ratón**, se genera un evento que nos ejecuta el método "mousePressed" del interfaz "MouseListener". Este método recibe un parámetro de la clase "MouseEvent e". Podremos acceder al método "e.getModifiers()" y comparar el resultado con unas constantes definidas en la clase "InputEvent" para saber qué botón hemos presionado.
- Cuando **pulsamos o soltamos una tecla del teclado**, se genera un evento que nos ejecuta el método "keyTyped", "keyPressed" ó "keyReleased" del interfaz "KeyListener". Estos métodos reciben un parámetro de la clase "KeyEvent e". Podremos acceder al método "e.getKeyChar()" para obtener el carácter Unicode de la tecla pulsada, o "e.getKeyCode()" para obtener el código de la tecla pulsada o soltada. Del mismo modo que antes, con el método "e.getModifiers()" y comparando el resultado con unas constantes definidas en la clase "InputEvent", podremos saber si hemos pulsado la tecla junto con las teclas modificadoras (MAYS, ALT, CTRL, etc.).

Un ejemplo de aplicación gráfica sencilla sería la siguiente:

```
/* ================================================================
   Titulo :    Hola Mundo (Interfaz gráfica AWT)
   Fichero:    HolaMundoAWT.java
   ================================================================ */

import java.awt.*;
import java.awt.event.*;

public class HolaMundoAWT extends Frame
{
   // Constructor
   private HolaMundoAWT (String sTitulo) {
      super(sTitulo);      // Establecemos el título del formulario
      WindowListener oWindowListener;
      oWindowListener = new WindowAdapter() {    // Atender eventos
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
      }
      this.addWindowListener(oWindowListener);
   }

   public void paint(Graphics g)
   {
      g.drawString("Aplicación ventana", 50, 50);
   }

   // Función principal
   public static void main(String[] args) {
      // Objetos
      HolaMundoAWT oApp = new HolaMundoAWT("Hola Mundo (AWT)");
      oApp.setSize(300,100);
      oApp.show();
   }
}

/* ================================================================ */
```

Ejemplo:

Vamos a escribir una pequeña aplicación Java en modo gráfico con un botón "Ok" y otro botón "Cancelar". Al pulsar "Ok", escribiremos "Botón Ok pulsado" en algún lugar de la ventana, y al pulsar "Cancel", "Botón Cancel pulsado". Además, como en todas las aplicaciones de tipo ventana, deberemos capturar el evento "windowClosing" para que podamos cerrar la ventana. En este evento podemos hacer dos cosas (por ejemplo):

- System.exit(0); // Cerrar la ventana
- setVisible(false); // Ocultar la ventana o hide() o dispose()

En el segundo caso, con "hide" nos puede servir para ocultar la ventana y poder acceder a sus métodos y atributos. Por ejemplo, al cerrar un diálogo, para poder recuperar los datos que ha introducido un usuario. Con "dispose" se descarga.

```
/* ================================================================
   Titulo :    Introducción a AWT - Version 0 (Interfaz gráfica AWT)
   Fichero:    IntroAWT0.java       IMPLEMENTANDO INTERFACES
                                    CLASES INTERNAS ANONIMAS
   ================================================================ */

import java.awt.*;
import java.awt.event.*;

class FrameAWT0 extends Frame implements ActionListener
{
   // Objetos
   TextField text;
   Button    buttonOk, buttonCancel;

   // Constructor
   public FrameAWT0 (String sTitulo) {
      // Establecemos el título del formulario
      super(sTitulo);
      // Objetos
      WindowListener oWindowListener;

      // Atender eventos del sistema
      oWindowListener = new WindowListener() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
         public void windowActivated(WindowEvent e) {};
         public void windowClosed(WindowEvent e) {};
         public void windowDeactivated(WindowEvent e) {};
         public void windowDeiconified(WindowEvent e) {};
         public void windowIconified(WindowEvent e) {};
         public void windowOpened(WindowEvent e) {};
      };

      // Construir ventana
      this.addWindowListener(oWindowListener);
      // Añadimos controles
      this.setLayout(new FlowLayout());
      // Texto
      text = new TextField(40);
      //text.setLocation (50,50);
      this.add(text);
```

```
        // Botón Ok
        buttonOk = new Button("Ok");
        buttonOk.setSize (50,50);
        //buttonOk.setLocation (50,250);
        this.add(buttonOk);
        buttonOk.addActionListener(this);
        // Botón Cancelar
        buttonCancel = new Button("Cancel");
        buttonCancel.setSize (50,50);
        //buttonCancel.setLocation (250,250);
        this.add(buttonCancel);
        buttonCancel.addActionListener(this);
    }

    // Interfaz: ActionListener (solo 1 método)
    public void actionPerformed(ActionEvent event)
    {
        if (event.getSource() == buttonOk) {
            text.setText("Botón <Ok> pulsado");
        } else if (event.getSource() == buttonCancel) {
            text.setText("Botón <Cancel> pulsado");
        }
    }
}

public class IntroAWT0
{
    // Función principal
    public static void main(String[] args) {
        // Objetos
        FrameAWT0 f;

        f = new FrameAWT0("Intro AWT (0)");
        f.setSize(300,100);
        f.show();
    }
}

/* ============================================================= */
```

Clases delegadas

Al crear los manejadores de eventos, hemos dicho que debemos tener una clase que implemente los métodos de la interfaz, según el tipo de eventos a capturar. Esta clase puede ser la clase principal de la aplicación, o bien, construir una clase explícitamente para cada tipo de eventos, que implemente todos los métodos necesarios. A esto se le llama uso de "clases delegadas".

Esto es muy útil, sobre todo cuando se tienen muchos eventos que gestionar, para no abarrotar la clase principal de nuestra aplicación.

Como buena conducta de programación, el uso de clases delegadas es recomendado, pues se favorece la reutilización de código.

Ejemplo:

Vamos a ver el mismo ejemplo de antes, utilizando esta vez clases delegadas.

```
/* ================================================================
   Titulo :    Introducción a AWT - Version 1 (Interfaz gráfica AWT)
   Fichero:    IntroAWT1.java    CLASES DELEGADAS
   ================================================================ */

import java.awt.*;
import java.awt.event.*;

class FrameAWT1 extends Frame
{
   // Objetos
   TextField text;
   Button    buttonOk, buttonCancel;

   // Constructor
   public FrameAWT1 (String sTitulo) {
      // Establecemos el título del formulario
      super(sTitulo);
      // Objetos
      ActionListenerDelegado al;
      WindowListenerDelegado wl;
      // Atender eventos del sistema
      wl = new WindowListenerDelegado(this);
      this.addWindowListener(wl);
      // Construir ventana
      // Añadimos controles
      this.setLayout(new FlowLayout());
      // Texto
      text = new TextField(40);
      //text.setLocation (50,50);
      this.add(text);
      // Botón Ok
      buttonOk = new Button("Ok");
      buttonOk.setSize (50,50);
      //buttonOk.setLocation (50,250);
      this.add(buttonOk);
      al = new ActionListenerDelegado(this,buttonOk);
      buttonOk.addActionListener(al);
      // Botón Cancelar
      buttonCancel = new Button("Cancel");
      buttonCancel.setSize (50,50);
      //buttonCancel.setLocation (250,250);
```

```
      this.add(buttonCancel);
      al = new ActionListenerDelegado(this,buttonCancel);
      buttonCancel.addActionListener(al);
   }
}

// Clase delegada: WindowListenerDelegado
class WindowListenerDelegado implements WindowListener
{
   Frame objFrame;

   WindowListenerDelegado (Frame obj) {
      objFrame = obj;
   }
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
   public void windowActivated(WindowEvent e) {};
   public void windowClosed(WindowEvent e) {};
   public void windowDeactivated(WindowEvent e) {};
   public void windowDeiconified(WindowEvent e) {};
   public void windowIconified(WindowEvent e) {};
   public void windowOpened(WindowEvent e) {};
}

// Clase delegada: ActionListenerDelegado
class ActionListenerDelegado implements ActionListener
{
   Button objButton;
   FrameAWT1 f;

   ActionListenerDelegado (FrameAWT1 f, Button obj) {
      this.f = f;
      objButton = obj;
   }
   public void actionPerformed(ActionEvent event)
   {
      if (event.getSource() == objButton) {
         f.text.setText("Botón <" + objButton.getLabel() + ">
pulsado");
      }
   }
}

public class IntroAWT1
{
   // Función principal
   public static void main(String[] args) {
      // Objetos
      FrameAWT1 f;

      f = new FrameAWT1("Intro AWT (1)");
      f.setSize(300,100);
      f.show();
   }
}

/* ============================================================= */
```

Clases adaptadoras

Hemos dicho que para crear un manejador de eventos debemos implementar una subinterfaz de la superinterfaz "ActionListener". Algunas de estas interfaces tienen un solo método, pero en cambio, hay otras que tienen algunos más. Como para implementar una interfaz debemos implementar todos y cada uno de los métodos que declara, esto, en ocasiones, puede ser molesto si tan solo nos interesa capturar uno de los posibles eventos.

Para evitar tener que hacer esto, Java tiene las "clases adaptadoras". Estas clases son clases que ya han implementado todos los métodos de sus respectivas interfaces, pero sin código, es decir, son métodos vacíos. Por tanto, en vez de usar las interfaces mencionadas, podemos heredar de estas clases adaptadoras, sobreescribiendo tan sólo el método o los métodos que nos interesen, pero ya no tenemos la obligación de definirlos todos, aunque sea con código vacío; este trabajo ya está hecho en las clases adaptadoras.

Por ejemplo, en vez de implementar el interfaz "MouseListener", la cual tiene cinco métodos, podemos heredar de la clase "MouseAdapter", redefiniendo sólo los métodos que nos interesen.

Manejador (Interfaz)	Clase adaptadora
ActionListener	ActionAdapter
AdjustementListener	(No disponible)
FocusListener	FocusAdapter
ItemListener	(No disponible)
KeyListener	KeyAdapter
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
WindowListener	WindowAdapter

Como vemos, no existen clases adaptadoras para todas las interfaces "Listener".

Ejemplo:

Vamos a ver el mismo ejemplo de los dos casos anteriores, utilizando esta vez clases adaptadoras.

```
/* ================================================================
   Titulo :    Introducción a AWT - Version 2 (Interfaz gráfica AWT)
   Fichero:    IntroAWT2.java    CLASES ADAPTADORAS
   ================================================================ */

import java.awt.*;
import java.awt.event.*;

class FrameAWT2 extends Frame
{
   // Objetos
   TextField text;
   Button    buttonOk, buttonCancel;

   // Constructor
```

```
   public FrameAWT2 (String sTitulo) {
      // Establecemos el título del formulario
      super(sTitulo);
      // Objetos
      ActionListenerDelegada al;
      WindowListenerAdaptadora wl;
      // Atender eventos del sistema
      wl = new WindowListenerAdaptadora(this);
      this.addWindowListener(wl);
      // Construir ventana
      // Añadimos controles
      this.setLayout(new FlowLayout());
      // Texto
      text = new TextField(40);
      //text.setLocation (50,50);
      this.add(text);
      // Botón Ok
      buttonOk = new Button("Ok");
         buttonOk.setSize (50,50);
         //buttonOk.setLocation (50,250);
      this.add(buttonOk);
      al = new ActionListenerDelegada(this,buttonOk);
      buttonOk.addActionListener(al);
      // Botón Cancelar
      buttonCancel = new Button("Cancel");
      buttonCancel.setSize (50,50);
      //buttonCancel.setLocation (250,250);
      this.add(buttonCancel);
      al = new ActionListenerDelegada(this,buttonCancel);
      buttonCancel.addActionListener(al);
   }
}

// Clase adaptadora: WindowListenerAdaptadora
class WindowListenerAdaptadora extends WindowAdapter {
   Frame objFrame;

   WindowListenerAdaptadora (Frame obj) {
      objFrame = obj;
   }
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
}

// Clase delegada: cActionListenerDelegada
class ActionListenerDelegada implements ActionListener
{
   Button objButton;
   FrameAWT2 f;

   ActionListenerDelegada (FrameAWT2 f, Button obj) {
      this.f = f;
      objButton = obj;
   }
   public void actionPerformed(ActionEvent event)
   {
      if (event.getSource() == objButton) {
         f.text.setText("Botón <" + objButton.getLabel() + ">
pulsado");
      }
```

```
    }
}

public class IntroAWT2
{
    // Función principal
    public static void main(String[] args) {
        // Objetos
        FrameAWT2 f;

        f = new FrameAWT2("Intro AWT (2)");
        f.setSize(300,100);
        f.show();
    }
}

/* ============================================================== */
```

Ejecución de aplicaciones

Para **ejecutar aplicaciones de tipo ventana** podemos invocar a la máquina virtual JVM ejecutando el fichero "java", o bien con "javaw" desde la consola del DOS. En el primer caso, no se nos devolverá el control hasta que la aplicación termine; en el segundo, el control se nos devuelve inmediatamente, lo cual es más conveniente porque la aplicación se ejecuta en otra ventana.

Clase Component

La clase más básica de AWT es java.awt.Component. Todos los componentes AWT derivan de ella, directa o indirectamente. Esta clase, es una clase abstracta que, a su vez, deriva de la clase Object. Los componentes Swing derivan también de ella.

Los **métodos** más importantes son:

Clase Component	
Método	**Uso**
add / remove	Añadir / eliminar un menú emergente
getBackground / setBackground	Obtener / establecer el color del fondo
getBounds / setBounds	Obtener / establecer los límites del componente en un rectángulo
getComponenteAt	Obtener el componente de unas coordenadas
getFont / setFont	Obtener / establecer la fuente del componente
getForeground / setForeground	Obtener / establecer el color de primer plano
getHeight / getWidth / getSize / setSize	Obtener / establecer alto y ancho
getMaximumSize / getMinimumSize	Obtener al tamaño máximo / mínimo
getName / setName	Obtener / establecer el nombre
getParent	Obtener el padre
getX / getY	Obtener coordenadas
hasFocus	Averiguar si tiene el foco del teclado
isEnabled	Averiguar si está habilitado
isLightweight	Averiguar si es peso ligero (Swing)
paint / paintAll	Pinta el componente (y todos los subcomponentes)
print / printAll	Imprime el componente (y todos los subcomponentes)
requestFocus	Solicita el foco de entrada
transferFocus	Transfiere el foco al siguiente componente
update	Actualiza el componente

Clase Container

Un contenedor es un objeto genérico AWT que puede contener otros componentes AWT.

Esta clase deriva de la clase Component y es la superclase para las clases contenedoras de AWT: Applet, Dialog, FileDialog, Panel y Window.

Clase Container	
Método	**Uso**
add	Añadir componentes a la lista de componentes del contenedor.
getComponent	Obtener el componente i-esimo de la lista, o bien, por posición (x,y).
getMaximumSize / getMinimumSize	Obtener el tamaño máximo / mínimo del contenedor
remove	Eliminar componentes del contenedor
getLayout / setLayout	Obtener / establecer el gestor de esquemas

Clase Frame

Es el marco de nuestra aplicación. Es la ventana que contiene la barra de título y el borde. Este borde cuenta cuando le pedimos las dimensiones de la ventana con el método "getInsets". Además estas dimensiones dependen de la plataforma.

Justo después de crear un objeto frame, debemos establecer un tamaño con "setSize", pues si no lo hacemos, no tendremos área de trabajo, es decir, nos aparecerá una ventana en la que sólo veremos la barra de título. La ventana no se visualizará hasta que llamemos al método "show".

El objeto frame creado puede ser de la clase "Frame", o de otra subclase nuestra que herede de la clase "Frame".

Un objeto frame, puede generar los eventos del tipo "WindowListener":

- windowOpened, windowClosed, windowClosing
- windowActivated, windowDeactivated
- windowIconified, windowDeiconified

Como mínimo debemos capturar los eventos de la ventana para cerrar la aplicación, pues si no lo hacemos, al pulsar el botón cerrar (X) no ocurrirá nada, y tendremos que salir abortando la tarea. Debemos llamar al método "System.exit" cuando se produzca el evento "windowClosing", que se genera al pulsar el botón cerrar (X).

Clase Frame	
Método	**Uso**
getFrames	Retorna un array con todos los frames creados por la aplicación
getIconImage / setIconImage	Obtiene / establece el icono de la ventana
getMenuBar / setMenuBar	Obtiene / establece la barra de menu
getState / setState	Obtiene / establece el estado
getTitle / setTitle	Obtiene / establece el título
isResizable / setResizable	Obtiene / establece si el usuario puede cambiar el tamaño de la ventana

Componentes

Vamos a estudiar los componentes básicos, y cómo podemos incorporarlos a nuestra aplicación gráfica.

Básicamente, el proceso de **incorporar un componente** a nuestra aplicación es siempre el mismo:

- Crear el nuevo control de la clase concreta (new)
- Añadirlo a un contenedor (Applet, Dialog, etc.), con el método "add"
- Opcionalmente, ejecutar métodos del nuevo control (asignarle valores, etc.)
- Opcionalmente, asociar gestores de eventos (listener)

Todo esto conviene realizarlo en el método "init()" si se trata de un Applet, o en el método "main()" si es una aplicación estándar.

Los más importantes son:

Componente	Descripción
Button	Botones
Canvas	Cuadro de dibujo
CheckBox	Casillas de selección
Choice	Casillas de opción
ComboBox	Listas desplegables
Label	Etiquetas
List	Listas de selección
Menu	Menu
PopupMenu	Menú contextual
RadioButton	Botones de opción
ScrollBar	Barra de desplazamiento
ScrollPane	Cuadro de desplazamiento
TextField / TextArea	Cajas de texto (1D:una línea / 2D:multilínea)
TextPane	Area de texto

Algunos componentes, pueden contener a otros. Se les llama **contenedores**:

Componentes contenedores	Descripción
Dialog	Cuadro de diálogo
FileDialog	Cuadro de diálogo de selección de ficheros (deriva de Dialog)
Panel	Recuadro para colocar otros controles dentro
Applet	Applet de tipo ventana AWT (deriva de Panel)
Window	Ventana
Frame	Marco ventana (deriva de Window)

Cuadros de texto (clase TextField)

Nos permiten introducir texto en campos, con la posibilidad de poner máscaras mientras escribimos, etc. Tan sólo permiten una línea.

Areas de texto (clase TextArea)

Son similares a los cuadros de texto pero permiten escribir texto en varias líneas. Debemos especificar su tamaño en caracteres (filas y columnas).

Etiquetas (clase Label)

Son cuadros de texto que no podemos modificar, simplemente se utilizan como títulos para indicar al usuario el uso de otros controles.

Botones (clase Button)

Nos permiten ejecutar alguna acción cuando pulsamos sobre ellos. Debemos capturar los eventos, mediante la interfaz ActionListener. Esta interfaz tiene tan solo un método: actionPerformed (ActionEvent e).

Casillas de selección ó activación (clase Checkbox)

Nos permite seleccionar o no varias opciones.

Botones de opción (clase CheckboxGroup)

Nos permiten seleccionar una de las opciones. En AWT, cuando agrupamos varias casillas de selección sólo podremos seleccionar una de ellas. Los métodos que se pueden utilizar son los mismos que para la clase Checkbox.

Listas (clase List)

Son listas de elementos de tipo texto, que nos permiten seleccionar uno o varios de ellos. Nos podremos desplazar con barras de scroll si tenemos más elementos de los que nos caben en la zona del componente.

Listas desplegables (clase Combobox)

Son listas en las que sólo podremos seleccionar un elemento. La lista permanece oculta hasta que la abrimos pulsando un botón a la derecha. Una vez seleccionado el elemento, la lista se vuelve a cerrar. Igual que antes, si tenemos muchos elementos podremos despalazarnos mediante barras de desplazamiento (scroll).

Barras de desplazamiento (clase Scrollbar)

Nos sirven para desplazar la información de otros controles en horizontal, en vertical o ambos. Disponemos de un indicador o cuadro de desplazamiento para desplazarnos de forma rápida, que al mismo tiempo nos informa de la posición en la que nos encontramos visualizando la información. También tenemos dos botones en los dos extremos de la barra para hacer desplazamientos pequeños. Así mismo, podemos hacer click sobre la barra, entre el cuadro de desplazamiento y los dos botones de los extremos, para hacer desplazamientos grandes en los dos sentidos. La cantidad de información que nos desplazaremos en los desplazamientos pequeños y en los grandes es una cantidad fija que habremos establecido previamente.

Cuadros o paneles de desplazamiento (clase ScrollPane)

Nos permiten desplazarnos con barras de desplazamiento por cualquier componente. Lo que hacemos es añadir componentes al área de desplazamiento, y el cuadro de desplazamiento ya se encarga de visualizar barras de desplazamiento y una parte de los controles que se han agregado al panel de desplazamiento. Podremos después movernos a las partes ocultas mediante las barras de desplazamiento.

Menús

Podemos añadir menús AWT a ventanas de la clase Frame mediante las clases:

- MenuBar: Añade una barra de menú.
- Menu: Los distintos menús de una barra de menú, con opciones y submenús.
- MenuItem: Las opciones de cada uno de los menús.
- MenuShortcut: Teclas de modo abreviado para las opciones de menús.
- CheckboxMenuItem: Opciones de menú con o sin marca.
- PopupMenu: Menús emergentes (no tienen que estar unidos a una barra)

Todas estas clases derivan directa o indirectamente de la clase abstracta "MenuComponent". La clase MenuShortcut deriva directamente de la clase "Object".

Clase MenuBar	
Constructor	**Uso**
MenuBar()	Crear una barra de menu

Clase MenuBar	
Método	**Uso**
add(Menu m)	Añadir un nuevo menú a la barra
deleteShortCut	Borra la tecla de acceso rápida del menú
getHelpMenu() / setHelpMenu(Menu m)	Obtiene / Establece el menú de ayuda de la barra
getMenu(int i)	Obtiene el menú i-ésimo
getMenuCount()	Obtiene el número de menús
remove(int i)	Elimina el menú i-ésimo
Enumeration shortcuts()	Obtener una enumeración de las teclas abreviadas

Clase Menu	
Constructor	**Uso**
Menu()	Crear un menu
Menu(String nombre)	Crear un menu con el nombre especificado

Clase Menu	
Método	**Uso**
add(MenuItem o) insert(MenuItem o, int i)	Añadir una opción al menú al final o en la posición dada
add(String nombre)	Añadir una opción con la etiqueta dada
addSeparator(), insertSeparator(int i)	Añadir un separador (línea), al final o en la posición dada
getItem(int i)	Obtener la opción i-ésima
remove(int i)	Elimina el elemento i-ésimo del menú
removeAll()	Eliminar todos los elementos del menú

Clase MenuItem	
Constructor	**Uso**
MenuItem()	Crear una opción de menú
MenuItem(String nombre)	Crear una opción de menú con la etiqueta dada
MenuItem(String nombre, MenuShortcut s)	Crear una opción de menú con la etiqueta dada, y con la tecla abreviada

Clase MenuItem	
Método	**Uso**
addActionListener(ActionListener l), removeActionListener(ActionListener l)	Añade / elimina el gestor de eventos de la opción de menú
deleteShortcut()	Elimina la tecla rápida asociada
disableEvents(long eventsToDisable), enableEvents(long eventsToEnable)	Deshabilita / habilita la gestión de eventos de esta opción
getActionCommand, setActionCommand	Obtener / establecer el nombre del comando (etiqueta) que ha generado
getLabel(), setLabel()	Obtener / establecer la etiqueta de esta opción de menú
getShortCut(), setShortCut()	Obtener / establecer la tecla rápida asociada con esta opción (objeto MenuShortcut)
isEnabled(), setEnabled(boolean b)	Averiguar / establecer si este elemento de menú está habilitado

Clase MenuShortcut	
Constructor	**Uso**
MenuShortcut(int key)	Crear una tecla abreviada (sin SHIFT)
MenuShortcut(int key, boolean useShift)	Crear una tecla abreviada (con SHIFT)

Clase MenuShortcut	
Método	**Uso**
equals	Averiguar si dos objetos MenuShortcut son iguales.
getKey()	Obtener la tecla asociada
useShiftModifier()	Averiguar si debemos pulsar SHIFT

El **código de la tecla** es una de las constantes estáticas definidas en la clase "KeyEvent". Tendremos que pulsar CTRL y la tecla que especifiquemos:

- VK_0, ..., VK_9
- VK_A, ..., VK_Z
- VK_ALT, VK_ALT_GRAPH
- VK_F1, ..., VK_F24

Clase CheckboxMenuItem	
Constructor	**Uso**
CheckboxMenuItem()	Crear una opción de menú con/sin marca
CheckboxMenuItem(String sLabel)	Crear una opción de menú con/sin marca, con la etiqueta especificada
CheckboxMenuItem(String sLabel, boolean state)	Crear una opción de menú con/sin marca, según especifiquemos, y con la etiqueta dada

Clase CheckboxMenuItem	
Método	**Uso**
addItemListener / removeItemListener	Añade / elimina el gestor de eventos
getState / setState(boolean state)	Obtiene / establece el estado (marcado o no marcado)

NOTA: Con la clase CheckboxMenuItem utilizamos el gestor de eventos "ItemListener", en vez del "ActionListener". Así mismo, debemos escribir el código de captura del evento en el método "itemStateChanged", en vez de en "actionPerformed".

Ejemplo:

```
import java.awt.*;
import java.awt.event.*;

// ActionListener: para las opciones de menús y submenús
// ItemListener: para las opciones de menú de tipo Checkbox.
class MiFrame extends Frame implements ActionListener, ItemListener
    {
        MenuBar menubar;
        Menu menu, submenu;
        MenuItem menuitem1, menuitem2;
        MenuItem subitem1, subitem2;
        MenuShortcut shortcut;
        CheckboxMenuItem cbsubitem;
        Label label;

        public MiFrame(String sTitle)
        {
            super(sTitle);
            label = new Label("Hola mundo");
            setLayout(new GridLayout(1,1));
            add(label);
            menubar = new MenuBar();

            menu = new Menu("Archivo");
            shortcut = new MenuShortcut(KeyEvent.VK_1); // CTRL+1
            menuitem1 = new MenuItem("Opción 1", shortcut);
            menu.add(menuitem1);
            menuitem1.addActionListener(this);
            shortcut = new MenuShortcut(KeyEvent.VK_2); // CTRL+2
            menuitem2 = new MenuItem("Opción 2", shortcut);
            menu.add(menuitem2);
            menuitem2.addActionListener(this);
```

```
            menu.addSeparator();

            submenu = new Menu("Submenu");
            menu.add(submenu);
            subitem1 = new MenuItem("Subopcion 1");
            submenu.add(subitem1);
            subitem1.addActionListener(this);
            subitem2 = new MenuItem("Subopcion 2");
            submenu.add(subitem2);
            subitem2.addActionListener(this);
            subitem2.setEnabled(false); // Deshabilitamos

            cbsubitem = new CheckboxMenuItem("Subopcion Checkbox");
            submenu.add(cbsubitem);
            cbsubitem.addItemListener(this);

            menubar.add(menu);
            setMenuBar(menubar);

            addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    //setVisible(false);
                    System.exit(0); }});
        }

        public void actionPerformed (ActionEvent event) {
            if (event.getSource() == menuitem1) {
                label.setText("Opción 1");
            } else if (event.getSource() == menuitem2) {
                label.setText("Opción 2");
            } else if (event.getSource() == subitem1) {
                label.setText("Subopcion 1");
            } else if (event.getSource() == subitem2) {
                label.setText("Subopcion 2");
            }
        }

        // Opciones de menu de tipo CheckBox
        public void itemStateChanged(ItemEvent event) {
            boolean bActivated;

            if (event.getSource() == cbsubitem) {
                bActivated = ((CheckboxMenuItem)event.getItemSelectable()).getState();
                if (bActivated) {
                    label.setText("Subopcion 3 (Checkbox): Activado");
                }else {
                    label.setText("Subopcion 3 (Checkbox): Desactivado");
                }
            }
        }
}

public class Menus {
    public static void main(String Args[]) {
        MiFrame f = new MiFrame("Menus");
        f.setSize (200,100);
        f.show();
    }
}
```

Menus emergentes (clase PopupMenu).

La clase PopupMenu deriva directamente de la clase "Menu", por lo que se puede utilizar en cualquier sitio donde se pueda utilizar un objeto de tipo "Menu".

Clase PopupMenu	
Constructor	**Uso**
PopupMenu ()	Crear un menú emergente
PopupMenu (String sLabel)	Crear un menú emergente con el nombre especificado

Clase PopupMenu	
Método	**Uso**
show(Component origen, int x, int y)	Mostrar el menú emergente en las coordenadas (x,y) relativas al componente

Ejemplo:

```
/* ================================================================
   Titulo :    Menus emergentes (Interfaz gráfica AWT)
   Fichero:    MenusEmergentesAWT.java
   ================================================================ */

import java.awt.*;
import java.awt.event.*;

// ActionListener: para las opciones de menús y submenús
class FramePopupMenusAWT extends Frame implements ActionListener
{
   Panel       panel;                  // Utilizado como contenerdor
   PopupMenu   popup;                  // Menu contextual
   MenuItem    menuitem1, menuitem2;   // Elementos del menu
   Label       label;                  // Una etiqueta

   public FramePopupMenusAWT(String sTitle)
   {
      // Establecemos el título
      super(sTitle);
      // Creamos el panel contenedor
      // (necesario para capturar eventos del ratón)
      panel = new Panel();
      this.add("Center", panel);

      // Añadimos componentes al contenedor
      label = new Label("Hola mundo");
      setLayout(new GridLayout(1,1));
      panel.add(label);

      // Creamos el menu contextual y le añadimos las opciones
      popup = new PopupMenu("Menu emergente");
      //
      menuitem1 = new MenuItem("Opcion 1");
      popup.add(menuitem1);
      menuitem1.addActionListener(this);
      //
```

```
        menuitem2 = new MenuItem("Opcion 2");
        popup.add(menuitem2);
        menuitem2.addActionListener(this);
        // Añadimos el popup menu a la aplicación
        this.add(popup);
        // Captura eventos de ratón
        // ATENCION: El manejador de eventos se asocia al panel,
        // Si se asocia al "frame" no funciona.
        panel.addMouseListener(new ControladorRaton(this, popup));
        // Captura eventos de la ventana
        this.addWindowListener(new ControladorVentana());
    }

    // Seleccion de opciones
    public void actionPerformed (ActionEvent event) {
        if (event.getSource() == this.menuitem1) {
            this.label.setText("Opción 1");
        } else if (event.getSource() == this.menuitem2) {
            this.label.setText("Opción 2");
        }
    }
}

// Eventos de la ventana
class ControladorVentana extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}

// Eventos del ratón
class ControladorRaton implements MouseListener { // extends
MouseAdapter {
    private FramePopupMenusAWT oFrame;
    private PopupMenu oPopupMenu;

    ControladorRaton ( FramePopupMenusAWT oFrame, PopupMenu oPopupMenu)
{
        this.oFrame = oFrame;
        this.oPopupMenu = oPopupMenu;
    }

    public void mousePressed(MouseEvent event) {
        // Pulsación de teclas del ratón
        if (botonPulsado(event)) {
            // Averiguamos si hemos pulsado el botón derecho
            if (botonPulsado(event,InputEvent.BUTTON3_MASK)) {
                // Si hacemos click sobre la barra de título (coordenadas
negativas) no hacemos nada
                if ( event.getY()>0 ) {
                    oPopupMenu.show(oFrame, event.getX(), event.getY());
                }
            }
        }
    }
    public void mouseClicked (MouseEvent event) {};
    public void mouseEntered (MouseEvent event) {};
    public void mouseExited  (MouseEvent event) {};
    public void mouseReleased(MouseEvent event) {};

    private boolean botonPulsado (MouseEvent event) {
```

```
      // Retorna true si hemos pulsado algún botón del ratón.
      boolean bOk = (event.getModifiers()!=0);
      return (bOk);
   }

   private boolean botonPulsado (MouseEvent event, int boton) {
      // Retorna true si hemos pulsado un botón del ratón.
      // El parámetro "boton" puede ser:
      //    InputEvent.BUTTON1_MASK = Botón izquierdo
      //    InputEvent.BUTTON2_MASK = Botón central
      //    InputEvent.BUTTON3_MASK = Botón derecho
      boolean bOk;

      bOk = ((event.getModifiers() & boton) == boton);
      return (bOk);
   }
}

public class MenusEmergentesAWT {
   public static void main(String Args[]) {
      FramePopupMenusAWT f;

      f = new FramePopupMenusAWT("Menus emergentes (AWT)");
      f.setSize (300,100);
      f.show();
   }
}

/* ================================================================ */
```

Cuadros de diálogo

AWT permite la creación de cuadros de diálogo personalizados mediante la clase Dialog, que deriva de la clase Window.

Las ventanas que se crean con esta clase se parecen más a los cuadros de diálogo estándar que los que se crean con las ventanas frame; por ejemplo, las ventanas de diálogo no tienen un botón minimizar ni maximizar. Sólo tiene la barra de título y un borde, y normalmente se usa para pedir algún valor del usuario.

Un diálogo siempre debe tener un objeto **ventana padre**, que podrá ser un Frame u otro Dialog. Por tanto, al ocultar o minimizar la ventana padre, el cuadro de diálogo no estará visible. Al volver a mostrar la ventana padre, se volverá a mostrar.

Un diálogo se puede abrir en modo "**modal**" o en modo "**no modal**". Un diálogo "modal" es uno que bloquea las acciones sobre cualquier otra ventana, hasta que el usuario cierre la ventana; en cambio, un diálogo "no modal", si lo permite .

Al derivar de Window, la clase Dialog genera los eventos de la Interfaz "WindowListener": WindowOpened, WindowClosing, WindowClosed, WindowActivated, WindowDeactivated.

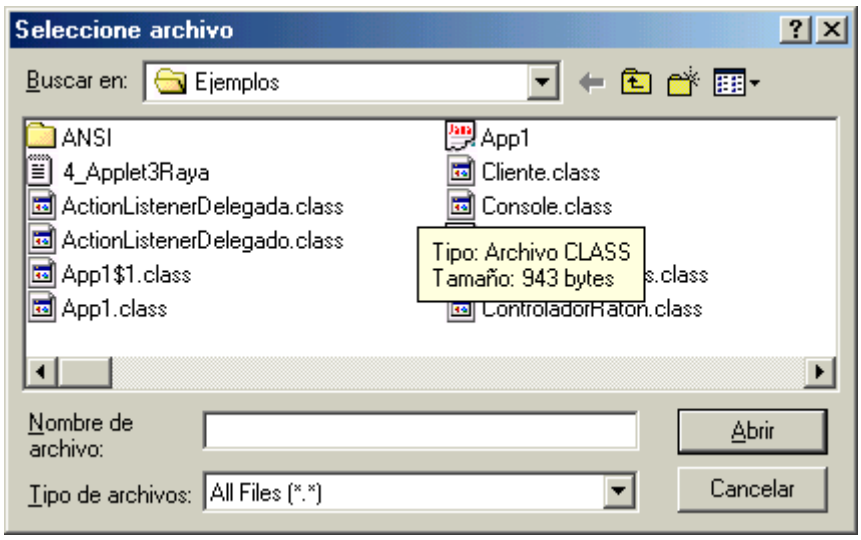
Clase Dialog	
Constructor	**Uso**
Dialog (Dialog/Frame owner)	Crear un diálogo, indicando el padre (Dialog o Frame)
Dialog (Dialog/Frame owner, String sTitle)	Idem, indicando además el título
Dialog (Dialog/Frame owner, String sTitle, Boolean bModal)	Idem, indicando además si lo mostramos en modo modal o no modal.

Clase Dialog	
Método	**Uso**
show()	Muestra el diálogo
hide()	Oculta el diálogo, pero no lo descarga
dispose()	Oculta y descarga el diálogo
isModal() / setModal()	Obtiene / establece si se ha mostrado en modo modal o no modal
isResizable() / setResizable()	Obtiene / establece si podemos cambiar el tamaño del diálogo
getTitle() / setTitle()	Obtiene / establece el título

Por supuesto, para crear un diálogo personalizado debemos escribir una clase que hereda de esta clase “Dialog”. Como en cualquier clase, podemos poner todos los constructores que queramos, y desde ellos llamar al constructor de “Dialog” mediante “super()”.

Clase FileDialog

Nos muestra una ventana de diálogo en la cual el usuario puede seleccionar un fichero, para leer o escribir. Al mostrar este diálogo, podremos seleccionar el nombre del fichero, pero la operación de lectura o escritura en concreto la tendremos que hacer nosotros.



Disponemos de dos constantes para especificar si vamos a seleccionar un fichero para leer (LOAD) o para escribir (SAVE).

Clase FileDialog	
Constructor	**Uso**
FileDialog (Dialog/Frame owner)	Crear un diálogo, indicando el padre (Dialog o Frame)
FileDialog (Dialog/Frame owner, String sTitle)	Idem, indicando además el título
FileDialog (Dialog/Frame owner, String sTitle, int Mode)	Idem, indicando además si vamos a leer un fichero (LOAD) o escribir en él (SAVE).

Clase FileDialog	
Método	**Uso**
getDirectory()	Obtiene el directorio del fichero seleccionado
getFile()	Obtiene el nombre seleccionado
getMode() / setMode(int Mode)	Obtiene / establece el modo (lectura/escritura), si no lo habíamos hecho ya en el constructor.
setDirectory()	Establece el directorio por defecto.
setFile()	Establece el nombre del fichero por defecto. También nos sirve para establecer una máscara o filtro de los ficheros a mostrar.

Gestores de esquemas o plantillas (Flow layout)

Los componentes se distribuyen por sus contenedores de forma automática, aunque también podemos hacerlo de forma manual. En este caso, es más complicado pues somos nosotros los responsables de mover los controles cuando redimensionamos las ventanas contenedoras.

Lo más sencillo es dejar que el **gestor de esquemas** por defecto (flow layout manager) distribuya los controles en el contenedor, o utilizar alguno predefinido.

setLayout(new FlowLayout());

En el caso de que queramos controlar nosotros la posición de los controles, debemos hacer:

- Anular el gestor de esquemas: setLayout(null);
- Añadir los controles (add)
- Posicionar los controles de forma manual: setLocation
- Ajustar el tamaño de los controles: setSize

Java incluye varios **gestores de esquema predefinidos**:

- Flow Manager (gestor por defecto): ordena los componentes como si estuviéramos escribiendo (de izquierda a derecha y de arriba a abajo).
- CENTER: Cada fila de componentes se centra en la ventana.
- LEFT: Cada fila de componentes se ajusta a la izquierda.
- RIGHT: Cada fila de componentes se ajusta a la derecha.
- LEADING: Cada fila de componentes se ajusta a la primera esquina del contenedor.
- TRAILING: Cada fila de componentes se ajusta a la siguiente esquina del contenedor.
- GRID LAYOUTS: Posiciona los componentes en una cuadrícula. Previamente le tenemos que asignar las dimensiones de dicha cuadrícula. Instanciando un objeto de la clase GridLayout nos sirve como parámetro para el método “setLayout”.
- BORDER LAYOUTS: Podemos colocar los componentes cerca de los bordes del contenedor, independientemente del tamaño del mismo. Esto se realiza con la clase BorderLayout. Igual que antes, podemos crearnos un objeto de la clase BorderLayout y pasárselo al método “setLayout”. Posteriormente, cuando añadamos componentes con el método “add”, podremos especificar la posición (“North”, “South”, “East”, “West”, “Center”).
- CARD LAYOUTS: Son solapas; cada uno de los contenedores que le pasamos se le da un nombre y lo considera como una solapa distinta. El mecanismo es crear un objeto de la clase CardLayout, y establecerlo como gestor de esquema (“setLayout”). A continuación, creamos los distintos paneles, de la clase “cardPanel”, con una etiqueta, y finalmente los añadimos asignánoles un nombre (“add”). Posteriormente podremos movernos de una a otra con el método “show”, “first”, “last”, “next” y “previous”.
- GRID BAG LAYOUTS: Son los más complejos pero los más versátiles. Se usan de la misma forma que los GridLayout, pero ahora disponemos de más opciones. En este caso, una vez creado un objeto GridBagLayout y establecido como gestor de esquema, antes de añadir un componente

tenemos que establecer la posición relativa y tamaño de éste respecto a los componentes ya insertados. para configurar los componentes se utiliza la clase GridBagConstraints. Por tanto, nos crearemos un objeto de esta clase, modificaremos las propiedades que nos interesan, ejecutamos el método "setConstraints" del objeto "GridBagLayout", pasándo como parámetros el componente creado y el objeto "GridBagConstraints" creado. Finalmente, añadimos el componente con "add".

Paneles (Panel)

Puede que en alguna parte de la ventana nos interese una distribución de componentes y en otra parte nos interese otra distinta, es decir, especificar varios gestores de esquema.

Para tener un mayor control sobre los gestores de esquemas, la mejor forma de trabajar es incluir los componentes en **paneles**, ya que éstos son contenedores, y añadir posteriormente dichos paneles a la ventana. De esta forma podremos especificar un gestor de esquema diferente para cada panel (setLayout).

Intercalados y rellenos (Insets)

Con los **intercalados** podemos añadir espacio entre los bordes del contenedor y los componentes. Se crean con la clase "Insets", especificando los márgenes a los cuatro lados (en píxeles).

Con los **rellenos** podemos separar horizontal o verticalmente los componentes. Esto se consigue con los métodos "setHgap" y "setVgap" de las clases "CardLayout", "FlowLayout" y "GridLayout". La forma de hacer esto es ejecutar alguno de estos dos métodos antes de establecer el esquema (setLayout) de alguna de las clases anteriores.

Gestores de esquemas personalizados

Aparte de poder eliminar el gestor de esquema por defecto (setLayout(null)) para colocar los controles especificando la posición y el tamaño de forma manual, podemos crear nuestros propios gestores de esquemas.

Para ello deberemos implementar la interfaz "LayoutManager2", sobreescribiendo los siguientes métodos (todos, ya que se trata de una interfaz):

- addLayoutComponent (String sName, Component oComponent)
- removeLayoutComponet (Component oComponent)
- getLayoutAlignmentX y getLayoutAlignmentY (Container target)
- invalidateLayout (Container target)
- preferredLayoutSize (Container parent)
- maximumLayoutSize y minimumLayoutSize (Container target)
- layoutContainer (Container parent)

Gráficos

Clase Graphics

Es una superclase abstracta para todos los contextos gráficos que permiten a las aplicaciones dibujar de forma gráfica.

Tanto AWT como Swing se basan en la clase Graphics. Se utiliza para dibujar cualquier tipo de figuras, puntos, polígonos, rectángulos, etc., así como rellenarlas de color.

Cuando especificamos coordenadas, dependerán del dispositivo de salida. Además, son relativas a la última posición donde se dibujó, es decir, trasladamos el origen en cada operación.

Los **constructores** de la clase Graphics son:

Constructor	Uso
Graphics()	Construir un nuevo objeto Graphics

Los **métodos** más importantes de esta clase son:

Método	Uso
clearRect	Limpiar el rectángulo especificado
clipRect	Copia el portapapeles en el rectángulo especificado
copyArea	Copia un rectángulo de un componente
create	Crea un nuevo objeto Graphics copia del actual
dispose	Libera un contexto de dibujo, liberando recursos
draw3DRect / fill3DRect	Dibuja un rectángulo en 3D
drawArc / fillArc	Dibujar un arco
drawBytes	Dibujar un texto, dando un array de bytes
drawChars	Dibujar un texto, dando un array de caracteres
drawImage (varios)	Dibujar una imagen
drawLine	Dibujar una recta
drawOval / fillOval	Dibujar elipses y círculos
drawPolygon / fillPolygon	Dibujar un polígono
drawPolyline	Dibujar una línea con varios tramos o segmentos
drawRect / fillRect	Dibujar un rectángulo
drawRoundRect / fillRoundRect	Dibujar un rectángulo con los puntas redondeadas
drawString	Escribe un texto en las coordenadas especificadas
finalize	Elimina un contexto gráfico cuando ya no se referencie
getClip	Obtiene el área de recorte actual
getClipBounds	Obtiene el rectángulo del área de recorte actual
getColor	Obtiene el color actual del contexto gráfico
getFont	Obtiene la fuente del contexto gráfico
getFontMetrics	Obtiene las medidas de la fuente actual del contexto gráfico

hitClip	
setClip	
setColor	
setFont	
setPaintMode	
setXORMode	
toString	
translate	

Clase Color

Se usa para encapsular los colores especificados en el formato RGB o en cualquier otro espacio de color arbitrario (ColorSpace). Cada color tiene un valor "alfa" de 0.0 a 1.0 (ó 0..255) que nos indica el nivel de transparencia. Un valor mínimo indica que es transparente; un valor máximo indica que es opaco.

Clase Font

Se utiliza para representar fuentes de caracteres. Debemos distinguir los conceptos de "Caracter" y "Glyph":

Caracter: es un símbolo que representa letras y números.

Glyph: forma de un caracter una vez dibujado.

Una codificación de caracteres es una correspondencia entre los caracteres y su imagen. Java utiliza la codifiación Unicode.

Una fuente es una colección de imágenes, las cuales pueden tener variaciones: negrita, cursiva, etc., manteniendo el mismo estilo tipográfico en todas ellas.

Clase FontMetrics

Internet - Comunicaciones

Comunicaciones: java.net

Introducción – Conceptos básicos

Vamos a definir unos conceptos básicos para comunicaciones en red, y posteriormente veremos las dos posibles formas que tiene Java para realizarlas:

- Uso de Sockets (TCP y UDP)
- Uso de URL

Hosts / Terminales

En Internet, cada ordenador conectado es un HOST o nodo.

En una red local, los clientes se llaman TERMINALES, y se conectan a una máquina principal o servidor llamado HOST.

TCP/IP

Es el nombre común de una familia de más de 100 protocolos que permiten la interconexión no sólo de computadoras locales, sino también, unas redes locales con otras.

El nombre viene de los dos protocolos más importantes:

- TCP (Transmission Control Protocol): se encarga de dividir los mensajes a enviar en PAQUETES o pequeños trozos de información, y de reconstruirlos en la recepción, pidiendo el reenvío de los paquetes erróneos al HOST de origen.
- IP (Internet Protocol): se encarga de transportar esos paquetes hasta un HOST remoto.

Otros protocolos son el HTTP, FTP, SNMT, POP, NNTP, etc.

Direcciones IP

Las direcciones IP son una serie de cuatro cifras 0..255 (4 bytes, 32 bits) separadas por puntos.

Cada Host u ordenador conectado a Internet tiene asignada una dirección IP fija; este es el caso de los servidores y algunos organismos oficiales, Universidades, etc., que mantienen una conexión permanente con la red. Por otro lado, la inmensa mayoría de los usuarios deben establecer una conexión para acceder a Internet, haciendo una llamada telefónica al servidor que nos da el acceso a Internet; en este caso, cada vez que establecemos una conexión, nuestro servidor nos asigna una dirección IP que varía cada vez.

Existe una dirección IP especial: 127.0.0.1 que referencia siempre nuestra máquina local. Por ello, tiene asociado el nombre "localhost".

Número de puerto

En muchos casos se especifica un número de puerto detrás de la dirección IP o nombre de host, separado por ":", el cual identifica una petición de un tipo particular.

Este número sólo será imprescindible especificarlo si no es el número de puerto predeterminado para el servicio al que accedemos mediante una dirección URL.

Servicio	Número de puerto
http	80
ftp	21
telnet	23

Nombre de Dominio

Un nombre de dominio es una dirección IP estándar. Para evitar que el usuario trabaje con direcciones IP directamente, nuestro servidor de acceso a Internet actúa también como servidor de nombres (Domain Name System ó DNS), que traduce los nombres de dominio que teclea el usuario a direcciones IP.

Esto es equivalente al fichero <Hosts> del sistema UNIX o incluso Windows, que consiste en un fichero de texto ASCII, en el que se especifica cada equivalencia en una línea diferente de la forma:

Dirección IP <Tabulador> Nombre

Ejemplo – Fichero C:\Windows\Hosts.sam

```
# Copyright (c) 1998 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP stack for Windows98
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#      102.54.94.97     rhino.acme.com          # source server
#       38.25.63.10     x.acme.com              # x client host

127.0.0.1       localhost
```

URL (Uniform Resorce Locator)

Se puede acceder a todos los recursos de Internet por medio de una dirección estándar especial llamada dirección URL (localizador uniforme de recursos), es decir, cada URL apunta a un recurso concreto de Internet.

El **formato** básico es:

Servicio://*NombreHost*/*RutaRecurso*

Servicio: es uno de los servicios que ofrece internet (http, ftp, news, ...)
NombreHost: nombre de la computadora
RutaRecurso: lugar dentro del Host donde reside el recurso, también llamado nombre de encaminamiento.

Por **ejemplo**:

http://www.microsoft.com/WindowsNT.htm

Servicio: http o página Web de hipertexto
NombreHost: www.microsoft.com
RutaRecurso: página con información relacionada con Windows NT

Es un tipo especial de dirección que nos permite acceder a cualquier tipo de información en Internet, mediante cualquier tipo de servicio de Internet (http, ftp, news, mailto, telnet, etc.). Las direcciones URL nos ofrecen un modo estándar de especificar el nombre y la situación exacta de cualquier recurso.

No siempre especificamos el nombre del host, como el caso de los servicios mailto y news.

Datagramas

En Internet, la información se transmite mediante datagramas. La información se divide en paquetes, denominados datagramas IP. Cada paquete puede seguir un camino diferente desde el origen hasta el destino, pueden llegar en distinto orden de cuando se enviaron, o puede que alguno no llegue al destino.

Tenemos dos posibilidades:

- Protocolo UDP (User Datagram Protocol), para manejar directamente datagramas IP.
- Protocolo TCP (Transmission Control Protocol), que se basa en el uso de "Sockets". Es una capa de software por encima que nos permite que nos permite que las transmisiones sean fiables.

Antes de ver las clases correspondientes a estos dos protocolos, vamos a estudiar la clase InetAddress, ya que tanto para uno como para otro la necesitaremos.

Clase InetAddress

Esta clase nos permite encapsular una dirección IP y su nombre asociado. Esta clase carece de contructores; a cambio, disponemos de unos métodos estáticos especiales denominados "métodos factoría".

Clase InetAddress	
Método factoría	**Uso**
getLocalHost()	Obtener un objeto de esta clase, el cual contiene la dirección IP de la máquina (host) donde se ejecuta nuestro programa
getByName(String sHostName)	Obtener un objeto, el cual contiene la dirección IP de la máquina (host) cuyo nombre hemos especificado
getAllByName(String sHostName)	Obtener un array de objetos, que contienen las direcciones IP de todos los hosts que tienen el nombre especificado.

Clase InetAddress	
Método	**Uso**
String getHostName()	Devuelve el nombre de la máquina que corresponde a la dirección IP del objeto
getAddress()	Devuelve un array de 4 bytes con la dirección IP
toString()	Devuelve una cadena con el nombre y la dirección IP.

Protocolo UDP

Clase DatagramPacket (Protocolo UDP)

Define la información que contendrán los paquetes a enviar o recibir, dependiendo de si vamos a enviar o recibir. Para establecer una comunicación entre dos ordenadores es necesario:

- Que ambos tengan asignada una dirección IP (32 bits)
- Que el servicio IP por el que se comunican tenga asignado un número de puerto (32 bits)

Como número de puerto, podemos utilizar cualquiera que esté libre, pero por convenio, se utilizan los números de puerto predeterminados.

Clase DatagramPacket	
Constructor	**Uso**
DatagramPacket (byte buffer[], int length, InetAddress Address)	Construir un objeto para enviar. En el buffer colocaremos los bytes a transmitir, y además deberemos, indicamos la dirección IP de destino y el número de puerto.
DatagramPacket (byte buffer[], int length)	Construir un objeto para recibir. Los paquetes recibidos se almacenan en un buffer de tamaño fijo.

Clase DatagramPacket	
Método	**Uso**
InetAddress getAddress()	Retorna la dirección de destino
getPort()	Retorna el número de puerto de destino
getData()	Retorna un array de bytes con los datos recibidos
getLength()	Retorna el número de bytes válidos del array obtenido con getData

Clase DatagramSocket (Protocolo UDP)

Nos permite enviar o recibir los datos encapsulados con la clase DatagramPacket.

Clase DatagramSocket	
Constructor	**Uso**
DatagramSocket ()	Construir un objeto para enviar
DatagramSocket (int nPort)	Construir un objeto para recibir por el puerto especificado

Clase DatagramSocket	
Método	**Uso**
send (DatagramPacket datagram)	Enviar un paquete
receive (DatagramPacket datagram)	Recibir un paquete

Protocolo TCP

Se basa en el uso de "sockets" para establecer comunicaciones fiables entre dos procesos. Estos procesos pueden residir en la misma máquina o máquinas remotas conectadas en red.

Un socket es un canal de comunicación, basado normalmente en el modelo cliente-servidor. Los clientes deben comprobar la disponibilidad del servidor para realizar la conexión. Por otro lado, el servidor debe esperar a que los clientes intenten la conexión.

Una vez realizada la conexión mediante un socket, para enviar o recibir información lo haremos del mismo modo que para escribir o leer de un fichero respectivamente, es decir, flujos de salida o de entrada.

Clase Socket (Protocolo TCP)

Clase Socket	
Constructor	**Uso**
Socket (String sHostName, int nPort, [, InetAddress localAddress, int localPort])	Construir un objeto para enlazar nuestra máquina con una máquina remota por un puerto especificado
Socket (InetAddress Address, int nPort, [, InetAddress localAddress, int localPort])	Idem, pero especificando en este caso la dirección IP de la máquina remota

Clase Socket	
Método	**Uso**
getInetAddress()	Nos retorna un objeto InetAddress que contiene la dirección IP del host remoto al que nos hemos conetado con el socket
getLocalAddress()	Obtener la dirección local a la que está enlazado el socket.
getPort()	Nos retorna el número de puerto al que nos hemos conectado en el host remoto
getLocalPort()	Idem para el host local
getInputStream()	Nos retorna una referencia al objeto InputStream asociado al socket
getOutputStream()	Nos retorna una referencia al objeto OutputStream asociado al socket
close()	Cierra los flujos de entrada y salida del socket

Clase ServerSocket (Protocolo TCP)

Esta clase nos permite la creación de aplicaciones servidores, que esperen la conexión de clientes de forma indefinida.

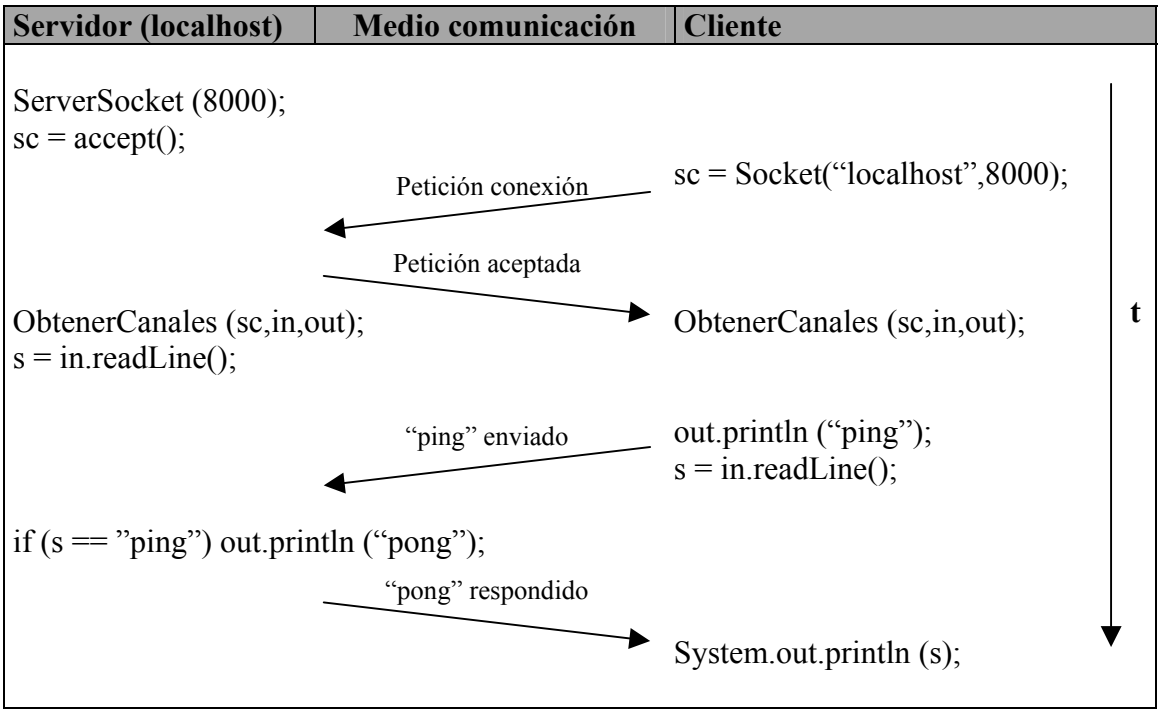
Podemos crear servidores que atiendan a un solo cliente o a muchos. En este último caso tendremos que hacer uso de la técnica de multihilos o threading. Tendremos un hilo principal para el servidor, y crearemos nuevos hilos hijos por cada nuevo cliente que se conecte; cuando un cliente se desconecte, el hilo correspondiente se destruirá.

Clase ServerSocket	
Constructor	**Uso**
ServerSocket (int nPort)	Construir un objeto servidor en el puerto especificado
ServerSocket (int nPort, int nTimeout)	Idem, solo que además, en el caso en que el puerto especificado esté ocupado, esperaremos a que se libere; si transcurrido el tiempo especificado no se ha liberado se producirá un error IOException.

Clase ServerSocket	
Método	**Uso**
accept()	Para que la aplicación servidor se quede esperando la conexión de un cliente debemos llamar a este método. El hilo principal de la aplicación quedará bloqueado hasta la llegada de un cliente.

Ejemplo de sesión de comunicaciones entre un cliente y un servidor: Ping/Pong

El servidor reside en la misma máquina que el cliente, es decir, ambos se ejecutan en la misma máquina, por lo que el nombre del host será "localhost". El servidor escucha por el puerto 8000, esperando que se conecte algún cliente. En ese momento, tanto el cliente como el servidor obtienen los canales de entrada y salida para la comunicación en ambos sentidos (full-duplex). Cuando el servidor recibe la cadena "ping" desde un cliente, el servidor le devuelve la cadena "pong".



En ambos casos, "**ObtenerCanales**" nos retorna los canales de entrada y salida, y se realiza de la misma forma, salvo que el servidor le pasa como parámetro el objeto socket cliente que retorna "accept()" cuando se conecta un cliente.

El código de esta función sería:

```
private void ObtenerCanales (Socket sc, BufferedReader in, PrintWriter out) {
        InputStream is = sc.getInputStream ();
        OutputStream os = sc.getOutputStream ();
        in = new BufferedReader (new InputStreamReader(in));
        out = new PrintWriter (new OutputStreamWriter(out));
        return;
}
```

Clases URL / URLConnection

Estas clases tienen la misma funcionalidad que las clases Socket y ServerSocket, pero son más sencillas de utilizar.

Clase URL	
Constructor	**Uso**
URL (String sURL)	Construye un objeto con una dirección URL
URL (String sProtocol, String sHostName, int nPort, String sFileName)	Nos permite acceder a un fichero remoto, especificando el protocol (http, ftp, ...), servidor, puerto y nombre del fichero
URL (String sProtocol, String sHostName, String sFileName)	Idem al anterior, utilizando el puerto por defecto en función del protocolo utilizado

Clase URL	
Método	**Uso**
URLConnection openConection ()	Nos retorna una referencia a un objeto que nos permite acceder a los atributos del objeto URL creado

La clase URLConection no tiene constructores; debemos utilizar el método openConnection() de la clase URL para obtener una referencia.

Clase URLConnection	
Método	**Uso**
getContentLength ()	Obtener la longitud del contenido
getContentType ()	Obtener el tipo del contenido
getDate ()	Obtener la fecha del contenido
getExpiration ()	Obtener la fecha de caducidad del contenido
getLastModified ()	Obtener la fecha de la última modificación
getInputStream ()	Obtener un objeto InputStream para leer el contenido

Applets Java

Como dijimos, Java nos permite crear tanto **aplicaciones convencionales** (ejecutables de forma autónoma), como **aplicaciones de Internet ó applets** (aplicaciones incluidas en páginas Web, ejecutables desde los navegadores de Internet).

El uso de Applets permite que tengamos páginas Web activas o dinámicas, es decir, no sólo muestren información estática.

Crear un applet

Para **crear** un applet Java necesitamos hacer los siguientes pasos:

- **Escribir el código** de nuestro programa, dentro de una clase derivada de la clase Applet. A diferencia de una aplicación estándar, no se necesita un método *main*. Como mínimo, debemos importar el paquete java.applet, y si nuestro applet trabaja en modo gráfico tendremos que importar otros paquetes.

```
import java.applet;

Public class <NombreApplet> extends Applet
{
        // Código de nuestro programa
}
```

- **Compilar el programa**

<NombreApplet.java> → <NombreApplet.class>

Debemos asegurarnos que cualquiera podrá acceder al fichero. Si estamos en un sistema Unix, deberemos asignarle los permisos: 644 o 777 (chmod).

- Insertar en una página Web en formato HTML una **referencia** al programa compilado. La forma más simple es:

```
<APPLET CODE = <NombreApplet.class>
                WIDTH = (Anchura utilizada en píxeles)
                HEIGHT = (Altura utilizada en píxeles) ... >
</APPLET>
```

En esta referencia podemos especificar otras etiquetas (tags) opcionales:

Etiqueta	Significado
[CODEBASE]	Es la dirección URL donde reside el applet (servidor y ruta). Si no se especifica se utiliza el de la página HTML que lo contiene.
CODE	Especifica el fichero .class del applet compilado, (incluyendo la extensión). La ruta es siempre realtiva a la página HTML que lo contiene o a la dirección especificada en el tag CODEBASE.
WIDTH / HEIGHT	➢ Especifican el ancho / alto de la ventana donde se visualizará el applet dentro de la página HTML que lo contiene. Se especifican en pixels. Esta ventana será de E/S. ➢ Estos valores no se utilizan para redimensionar la salida del applet. Si especificamos unos valores pequeños, tan sólo veremos una parte de la salida del applet.
[ALT]	➢ Especifica un texto alternativo, que se mostrará en el caso de que el navegador del cliente no pueda ejecutar el applet, pero si pueda interpretar el tag APPLET (por ejemplo, podría estar desactivado en las opciones la ejecución de applets). ➢ Si algún navegador no es capaz de interpretar el tag APPLET, se comportará como con cualquier otro tag desconocido, esto es, mostrará el texto tal cual, sin modificarlo. Esto se puede utilizar para mostrar un mensaje, en el caso de que un cliente utilice un navegador que no utilice Java.
[NAME]	Permite asignar un nombre al applet, pues es posible referenciar más de una vez un mismo applet en una página web. Además, este nombre permitiría identificarse a los applets que se comunican entre sí.
[ALIGN]	Especifica la posición de la ventana de visualización del applet respecto a la línea actual de la página HTML: left, right, top, bottom, middle, etc.
[VSPACE] / [HSPACE]	Especifican el espacio libre por arriba y por abajo (VSPACE) y por los lados (HSPACE) en píxeles.
[PARAM NAME= <Nombre> VALUE=<Valor>]	➢ Son parámetros opcionales que se pasan al applet, de la misma forma que los argumentos en la línea de comandos para ejecutar programas. ➢ Tanto el nombre como el valor pueden ir entre comillas dobles, aunque no es obligatorio, y los valores siempre se consideran cadenas de caracteres (String). ➢ El applet, para leer estos parámetros necesitar llamar a la función “getParameter(<Nombre>)”. Si se especifica un nombre de parámetro inexistente, la función retornará el valor <**null**>. ➢ Podemos insertar tantos como queramos, fuera de <APPLET CODE ...> y antes del marcador </APPLET>.

Ejecutar un applet

Para **ejecutar una aplicación convencional** necesitamos el intérprete Java, que sabemos que es la Máquina Virtual Java.

Para **ejecutar el applet** necesitamos un navegador que soporte Java o la utilidad <appletviewer> que incorpora el JDK de Sun, que se le considera como el navegador mínimo, debido a que sólo reconoce la etiqueta HTML <applet>, ignorando el resto.

En cualquier caso, necesitaremos un fichero html con el siguiente código como mínimo:

```
<html>
<applet code="applet.class" width=500 height=500>
/* El siguiente mensaje se mostrará si no podemos ejecutar applets Java */
Imposible ejecutar applets. Su navegador no soporta Java.
</applet>
</html>
```

Existe otra posibilidad más cómoda, que nos evita tener que crear una página HTML con estas líneas para poder probar un applet, sobre todo para depuración durante su desarrollo. Se trata de insertar justo las líneas anteriores entre comentarios /* ... */ al principio del código del applet, dentro del mismo fichero fuente *.java, y justo después de las sentencias "import".

Esto nos permite ejecutar el programa de la forma habitual, como con cualquier otro programa.

NOTA: Si a un applet, que no necesita un método main(), le añadimos uno, éste podremos ejecutarlo como applet y **como aplicación estándar**. Cuando se ejecute como un applet desde un navegador, el método main() se ignorará.

NOTA: Para activar/desactivar la **ejecución de applets en Internet Explorer**, entramos en la ventana de opciones, solapa "Opciones avanzadas", y en el árbol de opciones abrimos el nodo "Java VM". En ese grupo de opciones, debemos tener activa la opción "Compilador Java JIT activado".

Ciclo de vida de un applet

Como hemos dicho, todos los applets se derivan de la clase **Applet**, que en la jerarquía de clases de Java está en la siguiente posición:

Java.lang.Object
Java.awt.Component
Java.awt.Container
Java.awt.Panel
Java.applet.Applet

Esta clase tiene una serie de métodos que se ejecutan en momentos concretos del ciclo de vida de un applet. Podemos redefinirlos en la clase de nuestro applet:

Método	Momento de ejecución	Ejemplos
init	Cuando se carga el applet. Es el equivalente a la función *main*(). Es aquí donde podremos leer los parámetros pasados al applet con la función <**getParameter**>.	➢ Al visitar una página Web con el navegador, se carga el applet (Init) y luego se ejecuta (Start). ➢ Al pulsar el botón de <**Actualizar**>, se descarga y se vuelve a cargar.
start	Cuando el applet comienza su ejecución, y siempre después de cargarse, es decir, después de haberse ejecutado init.	➢ Al visitar otra página y volver a la página actual, estando el applet parado (Stop).
stop	Cuando el applet detiene su ejecución, y siempre antes de descargarse.	➢ Al visitar otra página Web, abandonando la página actual
destroy	Cuando se descarga el applet, liberando todos los recursos utilizados	➢ Al cerrar el navegador. ➢ Al pulsar el botón de <**Actualizar**>, se descarga y se vuelve a cargar.
update	LLena el área del applet con el color del fondo y después llama a "paint". Se puede forzar su ejecución llamando al método **repaint**().	➢ Cada vez que se necesita "refrescar" la imagen de la ventana del applet.
paint	Cada vez que se necesita "refrescar" la imagen de la ventana del applet, después de ejecutar el método Update(). Podemos utilizar el objeto Graphics del argumento para utilizar sus métodos gráficos para dibujar.	➢ Para evitar parpadeos, conviene que el único código que se ejecute en "paint" sea llamar al método "update".

Otros **métodos** más de esta clase son:

Clase Applet	
Método	**Uso**
repaint (int nTimeout)	Fuerza un repintado sólo si es posible realizarlo antes de los milisegundos especificados
repaint (int x, int y, int width, int height)	Fuerza un repintado, limitándolo al rectángulo especificado
repaint (int nTimeout, int x, int y, int width, int height)	Es la combinación de las dos anteriores
getCodeBase ()	Retorna un objeto URL con el directorio desde donde se cargó el applet (fichero *.class)
getDocumentBase ()	Retorna un objeto URL con el directorio donde reside la página Web wue hacía referencia al applet (*.htm o *.html)
showDocument (String sURL, String sWindow)	Hace que nuestro navegador cargue otra página Web. La nueva página puede mostrarse en la ventana actual ("_self"), en la ventana padre ("_parent"), en la ventana más visible ("_top") o en una nueva ventana en blanco ("_blank")

Plug-in de Java

Debido a que los desarrolladores de navegadores tardan un tiempo en actualizar la última versión de entorno de ejecución de Java en sus navegadores, Sun ha desarrollado el **Java Plug-in**, que consiste en un plug-in para Netscape y un control ActiveX para el Microsoft Internet Explorer. Esto nos permite ejecutar applets en los navegadores, teniendo siempre actualizada la última versión de Java de forma sencilla, pasando por alto la máquina virtual por defecto integrada con el navegador.

Este Plug-in se instala automáticamente al instalar el JDK, pero si no lo tenemos instalado, se puede descargar desde la dirección URL:

http://java.sun.com/products/plugin

El plug-in está incluido en la runtime (JRE)

Una vez instalado nos aparece un icono nuevo en el panel de control.

Para poder utilizar páginas Web con el plug-in, necesitamos convertir primero su código HTML, usando el **convertidor HTML** de Sun, que podemos descargar desde la misma dirección que el Plug-in anterior. Consiste en una herramienta (un fichero java compilado (*.class)) que se ejecuta sobre las páginas que le especifiquemos (filtro) para convertir la etiqueta <APPLET> de forma que pueda usarse por el plug-in, en vez de utilizar la máquina virtual integrada con el navegador.

Applets con interfaz gráfica

Un applet gráfico es un fichero de clase que nos permite visualizar gráficos en una página Web. El applet se realiza de la misma forma que cualquier aplicación gráfica estándar. Podemos utilizat AWT o Swing.

Cuando trabajamos con AWT, debemos importar las clases del paquete java.awt, pues el applet se basa en la clase Component de AWT.

Ejemplo:

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

/*
<APPLET CODE=HolaMundoApplet.class
        WIDTH = 500
        HEIGHT = 500>
</APPLET>
*/

public class HolaMundoApplet extends Applet
{
        public void paint(Graphics g)
        {
                g.drawString("Hola mundo", 50, 100);
        }
}
```

Los applets con interface gráfica con la librería Swing, deben ser una subclase de la clase JApplet, la cual es un contenedor en el que podremos colocar componentes (de tipo Swing). Salvo esto, todo es idéntico a lo explicado para interfaces gráficos en las aplicaciones estándar.

Consola Java

Si en el ejemplo anterior, hubiésemos utilizado el método "System.out.println" en vez de "drawString", la salida se mostrará en una ventana aparte. Dependiendo de que navegador estemos utilizando para ejecutar el applet tendremos:

- Appletviewer: se abrirá una ventana de línea de órdenes (consola) aparte.
- Internet Explorer: dispone de una consola Java que tenemos que habilitar de forma manual antes de utilizarla. Esto se hace con Menú Ver > Opciones de Internet > Opciones avanzadas > Habilitar "Consola Java activada" (necesitaremos reiniciar).
- Netscape Navigator: en este caso podemos abrir la consola Java con la opción Menú Communicator > Java Console.

Trabajo con múltiples ficheros

Los ficheros JAR (Java Archive) son un formato de ficheros multiplataforma que nos permite fusionar varios ficheros en uno.

Su principal utilidad es en el desarrollo de applets, pues podemos fusionar varios applets y sus recursos (ficheros compilados *.class, iconos y sonidos) en un único fichero *.JAR, y por tanto, se descargaría desde un navegador con una única transacción, lo cual siempre es más rápido. Además, estos ficheros pueden comprimirse, con lo que el tiempo de descarga será aun mucho menor. Además, el autor de los applets puede incluir una "firma digital" para comprobar la autenticidad del fichero.

Seguridad de un applet

Cuando un applet se carga en el ordenador de un cliente, antes de ejecutarse se comprueba las llamadas a funciones del sistema que realiza. Esto es así para evitar que un applet malintencionado "ataque" el ordenador del cliente. Este sistema de seguridad recibe el nombre de "**SandBox**".

El problema que tiene este sistema es que se limita la funcionalidad de los applets. Por ejemplo, un applet no puede nunca acceder al sistema de ficheros de la máquina que ejecuta dicho applet.

Por eso, en versiones posteriores de Java, además de este sistema se implementó otro sistema en el que los applets vienen con una **firma digital**, y se pide la confirmación del cliente para tener acceso a los recursos de su ordenador antes de su ejecución.

Por otro lado, los applets incorporan unos **bytes de comprobación** (checksum) para saber si un applet se ha modificado por terceras personas. Esto permitiría la detección de virus informáticos.

Apéndice A: Secuencias de escape: ANSI.SYS

El objetivo de este apéndice es conocer unas secuencias de escape para poder actuar sobre la pantalla en modo texto (consola). Java, a pesar de la gran cantidad de paquetes y clases que lo componen, no dispone de unos métodos tan básicos como borrar la pantalla o colocar el cursor en una posición concreta (fila, columna).

En MS-DOS, existen varios manejadores de dispositivo instalables: ANSI.SYS, DRIVER.SYS, RAMDRIVE.SYS, etc. Todos ellos se instalan poniendo una línea DEVICE en el fichero CONFIG.SYS.

En el caso del ANSI.SYS, debemos insertar en el fichero CONFIG.SYS una de las líneas siguientes, dependiendo de si vamos a trabajar en MS-DOS o Windows. Debemos reiniciar el sistema para que los cambios tengan efecto:

- MS-DOS: DEVICE=C:\DOS\ANSI.SYS o
- Windows: DEVICE=C:\WINDOWS\COMMAND\ANSI.SYS

ATENCION: A partir de la versión Windows 98 2nd Edition, aunque el fichero ANSI.SYS se encuentra instalado, aunque intentemos cargar este manejador de dispositivo insertando la línea anterior en el fichero CONFIG.SYS, Windows la ignorará. Por tanto, todo lo que viene a continuación sólo funciona bajo Windows 98 y versiones anteriores. Sin embargo, no funciona en Windows 98 2nd Edition, Windows NT ni en Windows 2000.

Una secuencia de escape ANSI es una serie de caracteres, comenzando con un carácter de escape (ESC = 27_{10} = 33_8 = $1B_{16}$) y un corchete abierto "[" que podemos utilizar para definir funciones para MS-DOS. Específicamente, podemos cambiar funciones gráficas, mover el cursor, borrar la pantalla, etc.

Para hacer uso de las siguientes secuencias de escape, lo único que debemos hacer es imprimir por pantalla una cadena de caracteres formada por dichas secuencias.

Java

Esto se puede utilizar con cualquier lenguaje de programación. En el caso del lenguaje Java, utilizaremos el método:

System.out.print (<*SecuenciaEscape*>);

La <*SecuenciaEscape*>, como hemos dicho debe comenzar por un carácter de Escape ESC. En Java, este carácter se puede expresar de varias formas:

- En unicode: "\u001B [..."
- En octal: "\033 [..."

Por ejemplo, un método para borrar la pantalla sería:

```
// Borrar pantalla
public static void ClrScr()
{
   System.out.print("\033[2J");
   return;
}
```

Funciones del cursor

Las siguientes secuencias de escape afectan a la posición del cursor en la pantalla. Consideramos que la pantalla es de 25 líneas y 80 columnas (25 x 80).

CUP - Posición del cursor

Uso: Mover el cursor a la posición especificada por los parámetros.
El valor predeterminado para <linea> y <col> es 1.
Secuencia: ESC [<linea>;<col> H

HVP - Posición horizontal y vertical

Uso: Mover el cursor a la posición especificada por los parámetros.
El valor predeterminado para <linea> y <col> es 1.
Secuencia: ESC [<linea>;<col> F

CUU - Cursor hacia arriba

Uso: Mover el cursor hacia arriba una línea sin cambiar de columna.
Podemos movernos un número de líneas (1 por defecto).
Si el cursor ya está en la primera línea se ignora la secuencia.
Secuencia: ESC [<num.lineas> A

CUD - Cursor hacia abajo

Uso: Mover el cursor hacia abajo una línea sin cambiar de columna.
Podemos movernos un número de líneas (1 por defecto).
Si el cursor ya está en la última línea se ignora la secuencia.
Secuencia: ESC [<num.lineas> B

CUF - Cursor hacia adelante

Uso: Mover el cursor hacia la derecha una col. sin cambiar de línea.
Podemos movernos un número de columnas (1 por defecto).
Si el cursor ya está en la última columna se ignora la secuencia.
Secuencia: ESC [<num.cols> C

CUB - Cursor hacia atrás

Uso: Mover el cursor hacia la izquierda una col. sin cambiar de línea.
Podemos movernos un número de columnas (1 por defecto).
Si el cursor ya está en la primera col. se ignora la secuencia.
Secuencia: ESC [<num.cols> D

DSR - Informe de estado del dispositivo

Uso: Obtener una secuencia RCP (ver abajo).
Secuencia: ESC [6 n

SCP - Guardar la posición del cursor

Uso: Almacenar la posición actual del cursor, la cual podrá ser restaurada mediante la secuencia RCP.
Secuencia: ESC [s

RCP - Restaurar la posición del cursor

Uso: Restaurar la posición del cursor, almacenada con SCP.
Secuencia: ESC [u

Funciones de borrado

Las siguientes secuencias de escape afectan a funciones de borrado.

ED - Borrar la pantalla completa

Uso: Borrar la pantalla y colocar el cursor en la posición inicial (1,1).
Secuencia: ESC [2 J

EL - Borrar hasta fin de línea

Uso: Borrar hasta el fin de la línea actual.
Secuencia: ESC [K

Modos de operación

Las siguientes secuencias de escape afectan el aspecto de los caracteres en la pantalla.

SGR - Encender la interpretación gráfica

Uso: Cambiar los atributos de los caracteres de la pantalla.
Estos atributos permanecen hasta la siguiente ocurrencia de una secuencia de escape SGR.
Secuencia: ESC [<atributo1>; <atributo2>; ... m

Como atributos podemos especificar los de la siguiente tabla:

Parámetro	Descripción
0	Apagar todos los atributos (caracteres normales)
1	Encender negrita
2	Encender tenue
3	Encender itálicas
4	Encender subrayado
5	Encender parpadeo
6	Encender parpadeo rápido
7	Encender video inverso
8	Encender invisible
30	Color negro
31	Color rojo
32	Color verde
33	Color amarillo
34	Color azul
35	Color magenta
36	Color cian
37	Color blanco
40	Fondo en negro
41	Fondo en rojo
42	Fondo en verde
43	Fondo en amarillo
44	Fondo en azul
45	Fondo en magenta
46	Fondo en cian
47	Fondo en blanco
48	Subíndice
49	Indice

Los parámetros del 30 al 47 cumplen con el estándar ISO 6429.

SM - Establecer modo

Uso: Cambiar el modo de la pantalla: modo texto o gráfico.
Secuencia de escape: ESC [= <modo> ESC [= h ESC [= 0 h ESC [? 7 h

El parámetro <modo> puede ser uno de los siguientes:

Parámetro	Descripción
0	Modo texto: 40 x 25 blanco y negro
1	Modo texto: 40 x 25 color
2	Modo texto: 80 x 25 blanco y negro
3	Modo texto: 80 x 25 color
4	Modo gráfico: 320 x 200 color
5	Modo gráfico: 320 x 200 blanco y negro
6	Modo gráfico: 640 x 200 blanco y negro
7	Avanza línea al final de cada línea

RM - Desactivar modo

Uso: Anular el modo de pantalla establecido con SM
Secuencia de escape: ESC [= <modo> ESC [= 1 ESC [= 0 1 ESC [? 7 1

El parámetro <modo> puede ser cualquiera de los que hemos visto antes. En el caso del parámetro 7, se reinicializa el modo que ocasiona avance de línea al final de cada línea.

Apéndice B: Enlaces de interés

Descripción	URLs
Servidor Obelix de la UMH	http://obelix.umh.es/
Sun Microsystems: Java Development Kit	http://java.sun.com/j2se/ http://obelix.umh.es/pub/programas/win/programacion/java/
Sun Microsystems: Java Runtime (JRE)	http://java.sun.com/products/jdk/1.2/runtime.html
Sun Microsystems: Java Forte	http://www.sun.com/forte/ffj/index.html
Sun Microsystems. Proyectos con código fuente	http://www.sunsource.net
Borland: JBuilder Foundation	http://www.borland.com/jbuilder/foundation/windows.html http://obelix.umh.es/pub/programas/win/programacion/java/
Thinking In Java (Eckel Bruce)	http://www.mindview.net/ http://www.yoprogramo.com/
Applets Java	http://softwaredev.earthweb.com/java
Aprenda Informática como si estuviera en primero	http://www.tayuda.com/ayudainf/index.htm
Varios tutoriales	http://programacion.net/java/
Mas tutoriales	http://www.infotutoriales.com/
Juego de caracteres Unicode	http://www.unicode.org/

Apéndice C: Referencias